Enabling Synergy in IoT: Platform to Service and Beyond

Michael P Andersen[1], Gabe Fierro[2], David E. Culler[3]

Electrical Engineering and Computer Science, UC Berkeley

Abstract

To enable a prosperous Internet of Things (IoT), 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 implementing and deploying a full-stack Synergistic IoT platform. We address hardware design concerns and present a reference platform, Firestorm. Upon this platform, we demonstrate firmware and personal-area networking concerns and solutions. Moving out towards larger scales we address local service discovery and syndication, and show how these principles carry through to global operation where security concerns dominate.

1. Introduction

The Internet of Things that we imagine involves far more than the 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 delineate 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

[1]m.andersen@cs.berkeley.edu
[2]gtfierro@cs.berkeley.edu
[3]culler@cs.berkeley.edu

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 (Bluetooth Low Energy) 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.

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 [4] to include the interactions and services that occur in those domains.

[4] Personal-Area, Local-Area and Wide-Area networks, respectively

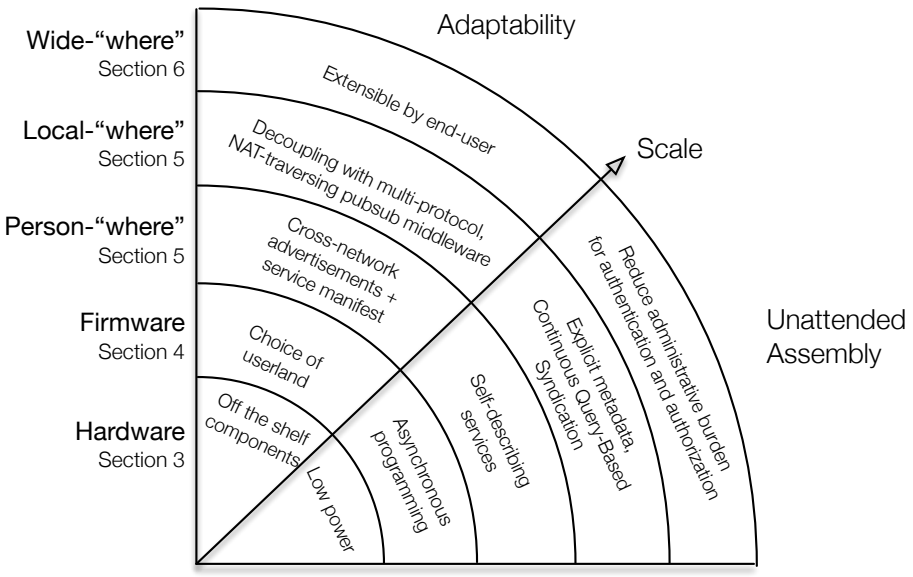


Figure 1: An effective IoT solution requires addressing adaptability and unattended assembly and operation at every tier. *Adaptability* means the ability of a device, service or application to a) detect and react to change in its environment, and b) support changes in its use case. *Unattended assembly* means the ability of a collection of devices, services and applications to discover each other, identify relevant resources, and operate at length without constant administration by a human supervisor.

2. 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 [1] 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 [5], 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 [6] and 802.15.4 or Bluetooth radios. Storm is extremely low power (2.3 μ in sleep with RTC [7] active) and supports many peripherals (63 GPIO pins). This adds to the commercial ecosystem of wearables, embedded Linux boxes (Raspberry

[5] SoC: system on chip
[6] MCU: microcontroller unit
[7] RTC: real-time clock

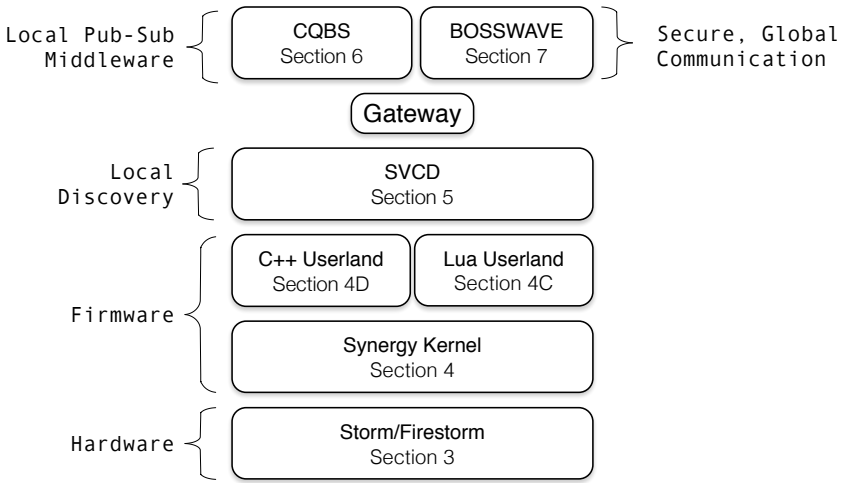


Figure 2: An IoT system architecture

Pi's and Beaglebones), BLE tags, and such.

Firmware: An extension of TinyOS [2] 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 with fewer computational resources than a Raspberry Pi or Beaglebone. 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.

To demonstrate the power of a dynamic Lua userland, we explore the creation of an Active Networks-inspired platform for wireless network measurement. The platform extends the syscall interface for visibility into the networking and radio stacks, and leverages an embedded Lua interpreter to introduce new logic and replace symbols in running code without the need to re-flash. From our experiences building production-ready firmware using a dynamic userland, we develop a C++11 userland that also provides efficient event-driven execution of asynchronous code but in a static, resource-efficient language.

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: self-assembly in this setting cannot assume the existence of coordinating infrastructure, so discovery and interaction must be able to perform in a "peer-to-peer" manner. We describe *SVCD*, a brokerless,

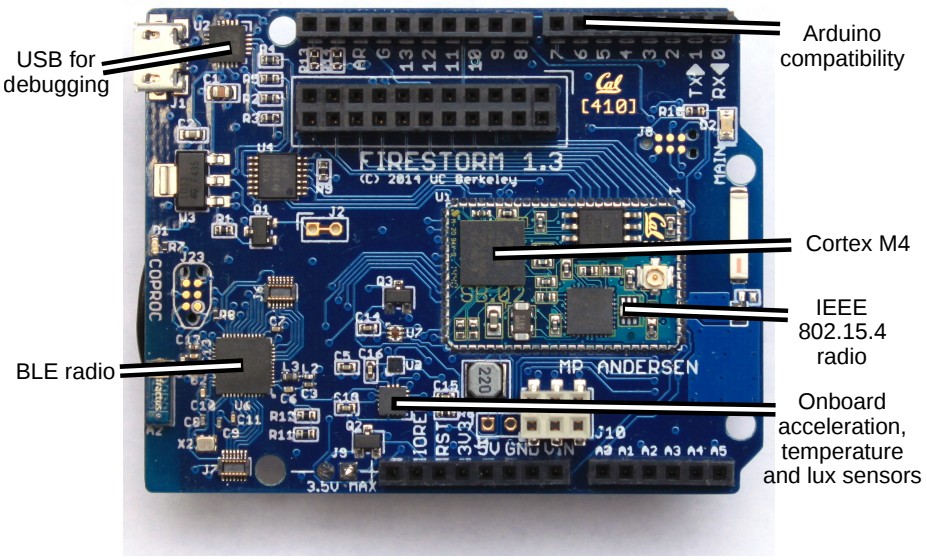


Figure 3: The Firestorm platform

local publish-subscribe mechanism that is symmetric over both IPv6/802.15.4 and BLE. We construct a simple, asynchronous API for SVCD and use this to construct a self-assembling "smart space heater" ensemble.

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. We explain how a new "flavor" of publish-subscribe — continuous query-based syndication — can enable applications to adapt to changing contexts and devices without the need for reconfiguration or reprogramming. To demonstrate, we integrate a building's HVAC system with a collection of networked chairs with heating and cooling capabilities, using the syndication mechanism to keep the operation of the application consistent even as the set of resources changes.

Wide-where: Beyond simply tunneling local interactions to resources hosted in the cloud, broader-scale ensembles containing devices and services 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. Such an infrastructure supports connecting devices and services into a global publish/subscribe network, utilizing an Ethereum-like block chain to provide secure cooperative action among agents that do not trust each other.

2.1. Related Work

There is a substantial body of work covering the areas of low-power, ubiquitous sensing, service discovery and composition, context-aware computing and preserving security in a distributed setting. Here, we consider this work in the context of the Internet of Things, specifically in relation to the principles of unattended assembly and adaptability.

Low-power sensing platforms such as the TelosB [3] and Mica [4] predate the Firestorm, but lack flexible communications (802.15.4 and BLE), higher computing power (the Firestorm obtains 20x more DMIPs in a Dhrystone benchmark) and a syscall ABI that enables a userland programming environment. The Firestorm achieves these features without compromising on low-power operation, meaning the Firestorm is much more amenable to rapid prototyping as well as unattended operation in long-term deployments.

Systems such as Phidgets [5] and Atlas [6] provide extensible hardware platforms that can adapt to many different deployment scenarios from assisted living to building automation. However, in doing so such systems sacrifice the ability to operate unattended at scale, and/or the ability to adapt their operation as their use case or environment changes.

Phidgets are powered embedded devices intended to be composed into physical user interfaces. Phidgets constitute a very extensible platform: the physical components are modular, and the Phidget software encourages rapid development of devices that can combine communication (X10, Ethernet, Bluetooth) with sensing and actuation. The software-based PhidgetManager discovers and interacts remotely with Phidget devices, enabling the creation of smart ensembles that react to devices entering and leaving a space. While at a high level the Phidget project shares the motivations of adaptability and ease-of-use with the Firestorm platform, the Phidget's lack of low power management, lack of an RTOS and reliance on a USB connection means it is unsuited to long-term embedded deployments.

The Atlas sensor platform also provides a modular hardware/software solution for programmable environments. The core Atlas board, based on an Atmel ATmega128L, contains contains headers for an array of extension shields for RF communication (Bluetooth and 802.11b WiFi) and various sensors and actuators. Each Atlas node uses the uIP stack for communication and for the main loop of the application, restricting choice of language and programming model. In contrast, the syscall ABI offered by the Synergy stack on the Firestorm allows the expression of both asynchronous and pseudo-synchronous applications in a family of languages (C, C++ and Lua). All Atlas nodes export a virtual representation to an OSGi-based [7] middleware server which is required for all node-to-node and node-to-application interactions. The middleware server stores "contexts", which are directed graphs of intentions and actions describing how sensors and actuators interact within a space. These tend to be hardcoded for a particular deployment, making it difficult to write the kinds of portable applications required in an Internet of Things.

The "person-where"' tier of interaction (explored in Section 5) concerns the

formation of ad-hoc ensembles of sensors and actuators in which nodes can independently discover and make use of resources around them. On the Firestorm platform makes use of both 802.15.4 and BLE radios for service discovery and invocation in node-to-node and node-to-persion interactions, respectively. The Firestorm is not the first BLE-enabled mote platform, and while prior efforts such as EcoBT [8] and BTnode [9] make effective use of BLE for discovering local sensors and actuators, the lack of a dynamic userland means it is difficult to expand the capabilities of a node after it has been deployed. Furthermore, only possessing a BLE radio means these platforms cannot take advantage of distributed resources outside of a single broadcast domain. Conversely, the Firestorm can advertise and consume services over both BLE and 802.15.4.

Recent research [10, 11, 12] has explored integrations of sensors across multiple physical layer technologies using BLE-enabled smartphones as gateways tying together BLE and NFC networks and using WiFi to pull in other sources such as 802.15.4 networks. The architecture proposed in this paper also permits the incorporation of smartphone gateways using the Firestorm's BLE radio, but reliance on such a gateway for cross-network communication is not compatible with the goal of unattended assembly. Large networks of embedded devices may use, but must not require the presence and intervention of an administrated smartphone in order to share data and communicate; otherwise, without a smartphone present, smart devices and services in a space are "dead in the water" without a means to coordinate. For this reason, the Firestorm contains an 802.15.4 radio so that it can communicate with other IP-based networks independent of a smartphone interlocutor.

Expanding the scope of service discovery and invocation from a broadcast domain to a local-area network usually requires some entity to act as coordinator and service registry. This decouples applications and services from the particular physical transport used by the distributed sensors and actuators in a space. Because these devices are typically battery powered and resource-constrained, the inclusion of a coordinator allows applications to not only discover duty-cycled devices that may not be active at the time of discovery, but discover these devices using rich, descriptive metadata. Interactions between devices, services and applications usually occur either through an RPC or publish-subscribe mechanism.

Because publish-subscribe (or *pub-sub*) brokers serve as both a discovery service and a message bus for all device/service/application interactions, they must strike a balance between expressiveness – the power of the data model offered to publishers and subscribers – and scalability. Common sense dictates that the more sophisticated a discovery system is, the less scalable it becomes. Within an IoT context, the choice of tradeoff is not obvious: rich and expressive descriptions are necessary for capturing the wide array of possible devices and contexts, but at the projected scale of the IoT, applications must be able to find and use relevant devices and services from the hundreds or thousands available at a given time.

Hierarchical topic-based publish-subscribe systems such as MQTT [13] and MQTT-SN [14] scale very well because topic-matching is a fast operation which can be easily sharded over multiple servers. By themselves, topics cannot carry

much descriptive information; real-world devices may exist under multiple hierarchies, and must advertise themselves under every possible topic in order to be reliably discovered. BOSSWAVE adopts the topic-based model, but persists metadata as descriptive key-value pairs on special topics their extend the expressive power. This augmentation allows the implementation of continuous query-based syndication (discussed in Section 6).

On the other side of the scalability/expressiveness tradeoff, lie content-based publish-subscribe systems. Systems such as SIENA [15], Gryphon [16], Java Message Service (JMS) [17], Elvin [18] and Jedi [19] leverage subscription schemes based on filtering the actual content or properties of received messages. These approaches are much more expressive because they operate over complex message structures, but come at the cost of performance. This is evidenced by the body of work surrounding content-based subscription over XML documents: each effort contains its own approach on reducing the computational overhead of filtering XML documents [20][21][22]. Simpler query mechanisms like JMS's SQL filters [17] or template-matching approaches for tuplespaces [23] offer arguably similar levels of expressiveness but without requiring a complex representation. The CQBS archiver (Section 6) and BOSSWAVE (Section 7) both use simple key-value structures for metadata, which enables both discovery and syndication through efficient, expressive queries.

This paper is an extension of a prior publication, which can be found at [24].

3. 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 – a requirement of unattended battery powered devices.

3.1. 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 [25][26].

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.

3.2. 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.

3.3. 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 $9.6\ \mu A$ 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.

4. 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.

4.1. 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, as the application must be programmed in a compatible language. 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.

4.2. 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. Sequential, synchronous logic is the most straightforward to program and understand, but

at the embedded tier, it becomes difficult to reconcile this approach with the asynchronous nature of 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. This logical separation of the definition an event or interrupt handler from the context of its invocation means that the handler must be able to *demultiplex* the received signal. Upon receiving an interrupt, platforms with many peripherals — radios, GPIO pins, I2C and SPI devices — must juggle state machines for the user application, the hardware peripherals and the communication interface between them. This decreases readability and increases complexity, making it difficult to reason about the control flow of an application and raising more possibilities for bugs.

This problem of demultiplexing asynchronous operations is not new. 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 [27]. 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.

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 has the potential to keep only 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.

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.

4.3. A Lua Userland

The Lua userland sacrifices memory usage for readability and dynamic code loading. Though Lua bytecode resides in ROM, which is typically more plentiful on embedded platforms, the language has first class functions, so the code in ROM serves to create the functions that then reside in and are executed from RAM. This means that RAM is still the limiting resource. The upshot is that Lua symbols — variables and functions — can be replaced at runtime without rebooting the application. The userland passes buffers of Lua code, which may be received over the network or loaded from ROM, to the Lua-C API `loadbuffer` function. If the buffer contains a complete piece of Lua code, the API returns the compiled chunk as an anonymous function which is executed in the global Lua environment, where it can override or define new symbols; this also empties the buffer. If the Lua code contained in the buffer is incomplete, the API does not compile the chunk or empty the buffer until it is. This behavior allows larger programs which may consist of a couple thousand bytes to be delivered over bandwidth constrained networks in the form of many smaller chunks of Lua code.

In addition to closures, Lua also includes support for concurrent, cooperatively scheduled coroutines. CORoutine Daemon, or CORD, is a 78-line Lua mini-scheduler for allowing multiple fibers of execution (Lua coroutines) to operate in parallel without the syntactic overhead of many nested callbacks. CORD adopts the `await` approach (found in C#) to express chains of nested, asynchronous logic in a sequential, pseudo-synchronous manner. Each asynchronous function takes a callback as its last argument which is invoked upon the completion of the asynchronous operation. CORD's `await` takes as arguments the

```lua
function sleep (t)
  cord.await(storm.os.invokeLater, t*storm.os.MILLISECOND)
end
-- usage inside a new CORD "fiber"
cord.new(function()
  print("Sleeping for 1 Second")
  sleep(1000)
  print("hello")
end)
```

Figure 4: How to implement a simple `sleep` in CORD

```
task next_state() {
  switch(state)
  case par_a_write_addr:
    call I2C.write(...);
    state = par_a_write_val;
    break;
  case par_a_write_val:
    call Timer.startOneShot(200)
    state = par_b_write_addr;
    break;
  case par_b_write_addr:
    call I2C.write(...);
    state = par_b_write_val;
    break;
  case par_b_write_val:
    call Timer.startOneShot(50);
    state = ...
}
event void I2C.writeDone(...) {
  post next_state();
}
event void Timer.fired() {
  post next_state()
}
```

Figure 5: TinyOS NesC code for writing a series of I2C registers

asynchronous function and its parameters, lacking the final callback. It then invokes the function with its own callback that receives the parameters and stores them to be passed as return values from the await invocation via Lua's `coroutine.resume()`. In this way, CORD mitigates the syntactic complexity of long chains of asynchronous operations.

A simple example is the implementation of an efficient `sleep()` that suspends only the *invoking* fiber, allowing the rest of the system to do useful work. The Synergy kernel provides a one-shot timer syscall — `storm.os.invokeLater( delay, fn, arg1, arg2, ...)` — which invokes the given function with its arguments after `delay` has passed. Using CORD's `await`, we can define a function that effectively blocks until `storm.os.invokeLater` completes, as seen in Figure 4.

A more thorough example of the power of callbacks in expressing chains of asynchronous logic is the common pattern of interleaving writes to I2C registers with short sleeps. These types of state machines are commonly used for interacting with peripherals such as LCD screens and LED strips. First, in Figure 5, we present how this would be accomplished using the TinyOS model, which uses split-phase pairs (an invocation and a completion handler) to han-

```
function sleep(t, cb)
  storm.os.invokeLater(
    t*storm.os.MILLISECOND, cb)
end
function write_i2c_reg(addr, val, cb)
  storm.i2c.write(..., function()
    storm.i2c.write(..., cb)
  end)
end
write_i2c_reg(PAR_A, VAL_A, function()
  sleep(200, function()
    write_i2c_reg(PAR_B, VAL_B, function()
      sleep(50, function()
        ...
          ...
        ...
      end)
    end)
  end)
end)
```

Figure 6: Synergy Lua userland code for writing a series of I2C registers using Lua's closures

dle asynchrony. A global `state` variable records the position within the state machine, which quickly becomes untenable as the state machine grows in complexity. Furthermore, if the application in Figure 5 were to be interacting with multiple I2C peripherals, then the `I2C.writeDone()` event handler would have to demultiplex which write operation had completed.

Using callbacks, it is possible to cleanly associate the completion logic for each timer firing and I2C operation with which logic invoked it, removing most of the demultiplexing logic. Figure 6 illustrates how this looks using the Lua userland (without CORD).

Finally, Figure 7 shows how CORD expresses the same logic in a synchronous manner. Each `cord.await` call blocks until the operation is finished, but does so without wastefully spinning the processor.

4.3.1. Lua Example: WSN Network Measurement Platform

The dynamic Lua 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 [28], code dissemination [29], and network testbeds [30]. Traditionally, the storage constraints of embedded platforms have forced an opacity onto embedded networking stacks, making it difficult to add monitoring logic without removing functionality from the stack itself. The iterative process of altering the networking stack for any sort of parametric study is intractable for more than a few parameters. Usually, these

```
function sleep (t)
  cord.await(storm.os.invokeLater,
     t*storm.os.MILLISECOND)
end
function write_i2c_reg(addr, val)
  cord.await(storm.i2c.write, ...)
  cord.await(storm.i2c.write, ...)
end

write_i2c_reg(PAR_A, VAL_A)
sleep(200)
write_i2c_reg(PAR_B, VAL_B)
sleep(50)
...
```

Figure 7: Synergy Lua userland code for writing a series of I2C registers using CORD.

parameter spaces are explored with the help of a network simulator such as COOJA [31] or NS2 [32]. While helpful for establishing estimates of how a protocol under a set of parameters might perform, it is difficult to evaluate the practicality of a protocol without having implemented it in a real system. 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.

Figure 8 illustrates the architecture of the platform. The network measurement platform provides a set of libraries and utility functions so that code capsules do not have to replicate basic functionality. The capsules, which define a network experiment, make use of a number of syscalls added to the Synergy kernel for visibility into the networking and radio stacks. This was greatly facilitated by Lua's C API. The added syscalls retrieve transmission and retransmission counts, maintain logs of which routing protocol control messages were sent, and can list, add and clear entries from the routing and neighbor tables.

Each experiment is started and coordinated by either a laptop or desktop computer attached to the network. This coordinator disseminates code, triggers the beginning of the experiment and retrieves the reports from the nodes after the experiment. The coordinator does not participate in the experiment, but it can serve as a "data drop" for a measurement application such as measuring the packet reception ratio. These applications separate the traffic used to measure the network from the reporting of those measurements by storing gathered data on the Storm's ample external flash memory. When the experiment concludes, the application transmits the data to the coordinating server.

An issue that can arise is if the routing protocol under study does not form a

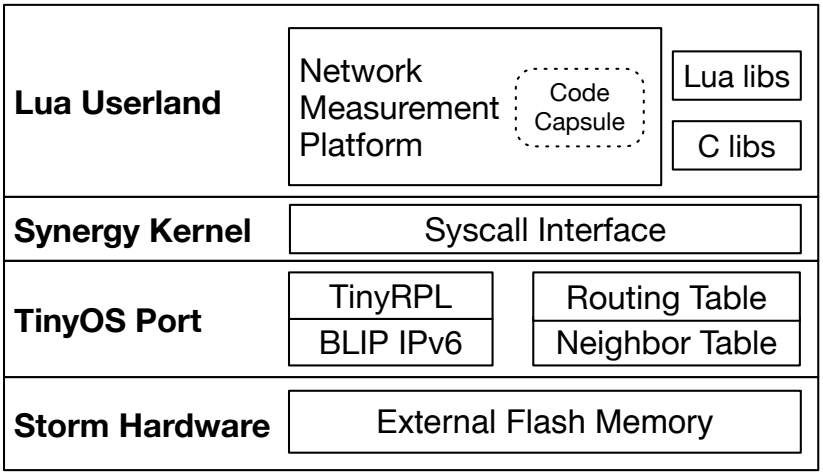


Figure 8: The full hardware/software stack for network experimentation. Kernel syscalls take advantage of the synergy between hardware and firmware to expose low-level metrics on the radio stack that would otherwise be difficult to incorporate.

stable routing mesh, then the reporting mechanism can fail. Thus, the network measurement platform allows for code to be transmitted over link-local or global IP, or over Bluetooth; in the event of a link or routing failure, an administrator can interactively debug the issue using a Lua shell, and either repair the issue or retrieve the data through some external means.

4.4. A C++ Userland

While many IoT applications can benefit from the agility of dynamic languages such as Lua, discussed above, C-type languages remain the staple of production embedded applications because of their predictability. We argue

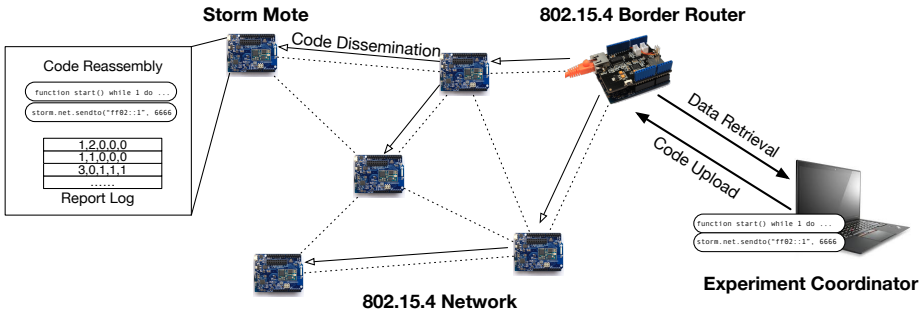


Figure 9: Code dissemination over WSN nodes. Each node contains the stack in Figure 8. Code is disseminated from a coordinator node via unicast to each node. After the experiment is complete, reports are sent back to the coordinator.

```
void write_i2c_reg(..., function<void()> cb){
    storm::i2c::write(..., [=]{
        storm::i2c::write(..., cb);
    });
}
write_i2c_reg(..., [=]{
    storm::sleep(200, [=]{
        write_i2c_reg(..., [=]{
            storm::sleep(50, [=]{
             ...
            });
        });
    });
});
```

Figure 10: Asynchronous C++ code for interacting with an I2C device.

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. For the I2C writing example above, the C++ code would look like Figure 10.

One use of the C++ userland on Firestorm was for firmware on the smart chair discussed in Section 6. 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.

4.4.1. Futures vs Callbacks

With a single stack, it is not possible for C++ to easily hide the callback hell in the same way as CORD and `await` did above. As a result we investigated the use of *promise*s or *future*s, as this pattern has been used in the Javascript world to resolve many of the same problems, with some success. Briefly, a promise

allows you to attach some logic to it that will be executed when the promise is `resolve`d. If the logic happens to be attached to the promise after the event happens, that is not a problem, it will be executed anyway. This property is essential for preventing race conditions and unpredictability.

Unfortunately, promises are not as suited for embedded development. There is a critical difference in the nature of events between embedded systems and web browsers. In the latter, most events only happen once, whereas on embedded systems events are often things like a periodic timer or a button interrupt that occur multiple times.

In this case, to get the same guarantees around race conditions, the promise must store a potentially unbounded list of events that have occurred, so that any handling logic which is attached later can be invoked on them. On a machine with a lot of RAM this may not be a problem, but on an embedded system, this is undesirable. One is forced therefore to trade off the reliability of promises (sometimes a handler may miss events, depending on race conditions) in order for them to work in the embedded space.

In contrast, despite their lack of syntactic aesthetics, callbacks do not suffer from any of these problems. At the time an asynchronous function is invoked with a callback, the callback and all nested handling logic inside it, is completely specified. Even if the asynchronous function were to emit several events synchronously as it is invoked, no events will be missed by handling logic.

5. 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 applications 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 (hence, "person"-where), and defer our discussion of discovery and service composition in a local-area network to the next section.

5.1. 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

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 11: 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

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 of services within a broadcast domain, a platform must

1. attend to self-describing services
2. adapt to usage patterns
3. provide event subscriptions
4. not require additional, external infrastructure

Self-describing services: In general, self-describing devices and services 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.

Adapt to usage-patterns: 802.15.4 and BLE encapsulate different modes of interaction — typically machine-to-machine and human-to-machine respectively. Providing functionality over both of these interfaces means the service can operate with different use-cases without the need to replicate logic. The challenge with this approach is that the communication to and from such a

service must be performed over both characteristic-oriented (BLE) and byte-oriented (802.15.4) media. With a characteristic-oriented service advertisement framework (which can be emulated over 802.15.4 using structured datagrams), 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.

Event Subscriptions: 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 abstraction for creating and handling subscriptions across networks.

No external infrastructure: For true self-assembly of smart device ensembles in arbitrary environments, a platform should not assume any existing infrastructure for invocation and discovery of services. In the case of discovery services that require external infrastructure (such as Zigbee), at sufficient scale it becomes difficult to maintain a consistent view of available resources (an issue which we address in Section 6). Instead, in the personal area, we explore how devices can directly discover relevant resources.

The SVCD framework integrates these design points into a simple, asynchronous API (Figure 11). 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 [33], DNS-SD [34])
- discovery cannot be performed in a peer-to-peer manner by devices, requiring an external coordinator (ZigBee [35])
- 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)

A brief investigation of ZigBee illustrates many of the prohibitive issues with state-of-the-art discovery protocols. ZigBee's strength is in its impressive cluster library, which is an object-oriented list of device service classifications. These include power configuration, thermostats, fan control, temperature measurement, and many others. ZigBee Application Profiles define sets of required and optional devices and clusters needed to implement a given high-level application such as Home Automation. One issue is that ZigBee Application profiles prescribe the sets of devices that can participate in an application, so it is difficult to incorporate custom devices (such as those by makers and hobbyists) into a ZigBee network. Also, ZigBee devices may only communicate through parent nodes acting as routers, so devices cannot independently discover service

providers in their broadcast domain. Lastly, ZigBee does not support subscriptions to changes in device characteristics.

5.2. 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, obviating the app-per-device approach. Note that the GitHub manifest is not required for correct operation — it is simply a descriptive aid for human operators. Applications operate entirely upon the service and characteristic identifiers in the messages.

Figure 12 is an example of a service description contained in the service manifest file. The use of succinct identifiers and service/attribute grouping

```
"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 12: 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.

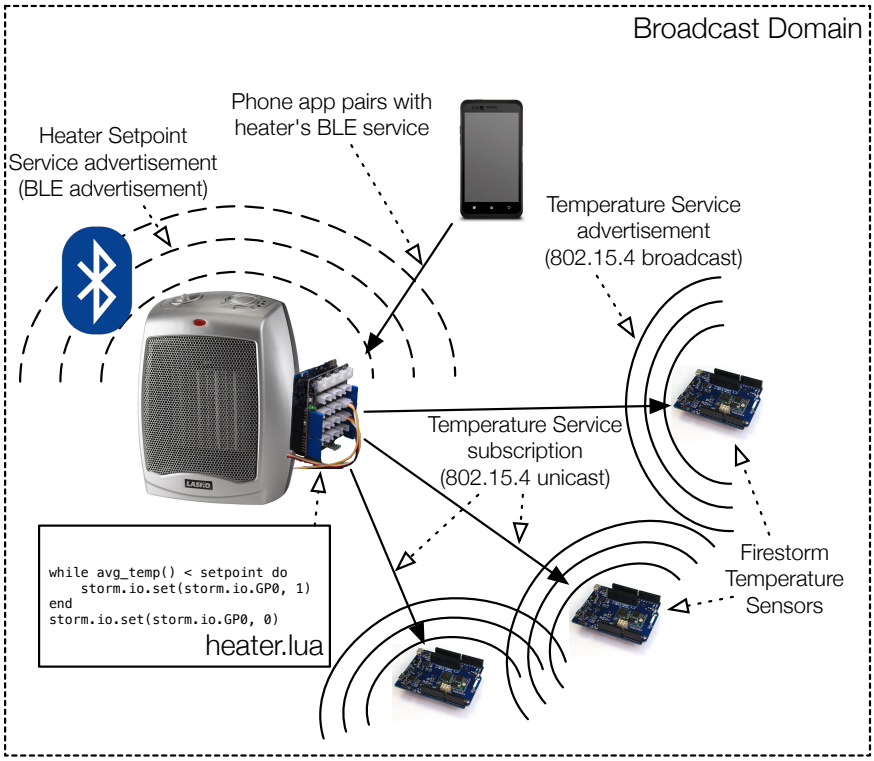


Figure 13: Example of composing locally-discovered services into an ensemble application using a mobile phone, 802.15.4 and BLE.

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 [36], 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 systems like UPnP, in which advertisements do not contain an actual description of services, but rather a URL pointer to where those descriptions reside.

The API, as seen in Figure 11, 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 is compatible with 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 6.

5.3. 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 13).

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 12) 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 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. The application continues to listen for service and device advertisements so that it can quickly react to the addition of any new devices.

6. 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.

6.1. 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.

Each device advertises itself to the CQBS archiver as a set of streams: a *stream* (Figure 14) 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:

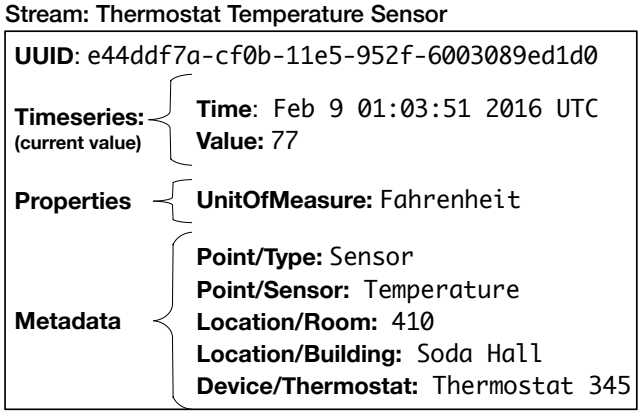


Figure 14: A stream example of a thermostat's temperature sensor

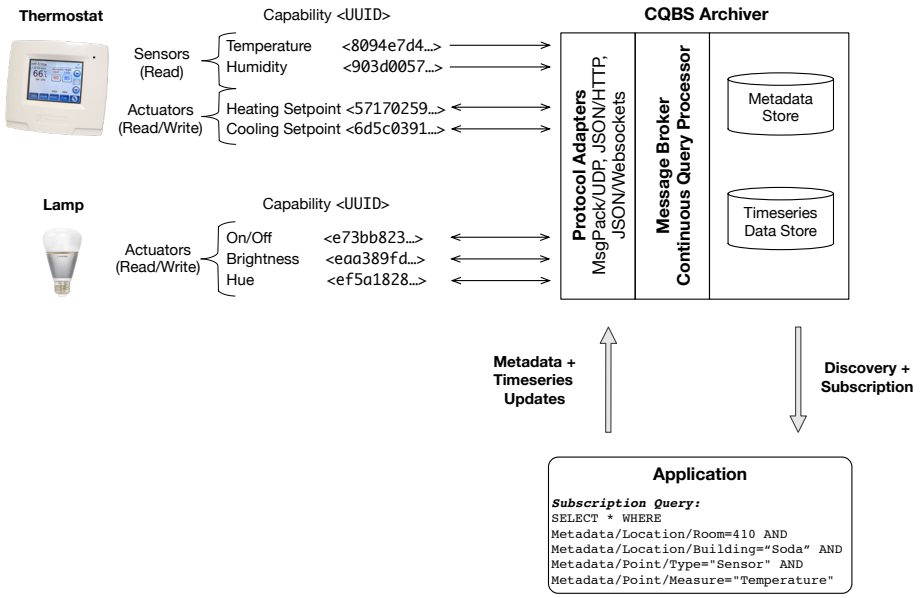


Figure 15: The archiver is the central component in a local-area IoT deployment. Devices register themselves with the archiver using rich metadata to describe their context. Applications can then discover these devices by expressing queries over the metadata. Applications can also access historical and 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.

- 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 (e.g. lighting, heating, printing, visualization, etc) 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.

6.2. 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 15. 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 [37] or provide query-based subscriptions that do not update the publisher-consumer binding as metadata changes [38].

These concerns remain largely unaddressed by modern systems concerned with the composition of services over networked things. These systems – including CORBA [39], Jini [40], AllJoyn [41] and Iotivity [42] – 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 in the case of real-time data-streaming applications.

6.3. 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 [43] 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 [44], 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.

6.4. 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.

7. 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?
- Many IoT devices and services are not directly reachable from the Internet (NAT for example). How would these devices and services communicate?
- How would secure transport mechanisms such as SSL work at such a large scale? Can we lower the administrative burden?

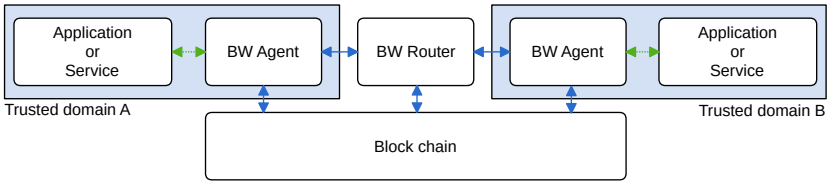


Figure 16: The architecture of BOSSWAVE communication

- How would permissions for operations on IoT devices and services work? Can we lower the management overhead?
- How can IoT devices and associated micro-services engage in an open market?

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.

7.1. Architecture

The core functionality offered by BW is a global publish/subscribe network with a security model that allows scaling both at a technological level, and a management one.

Each application or service interacting with the BW network communicates via a BW *agent* which the service or application completely trusts, over a plaintext protocol. Ideally the agent would be running on the same machine. That agent then interacts with the rest of the network over encrypted channels assuming that any other agent on the network could be hostile. For situations where many agents must cooperate without trusting each other, a modified Ethereum[45] blockchain is used, and the logic is encoded as a smart contract.

7.2. 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. To make these public keys easy to handle, a smart contract on the blockchain is used to create a human-readable alias of the key. The contract assures that the alias is unique, immutable and that every agent on the network knows about its existence.

7.3. 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.

An example of such a URI would be:

```
Y3-WHr5VQJRxN8r6cCYej3bo4q-rxF_NnfF_74V-4L0=/eecs/soda/410/lighting
```

Using a blockchain smart contract alias, the following URI is completely equivalent:

```
scratch.ns/eecs/soda/410/lighting
```

Mapping from the namespace public key to the key of the router responsible for handling the namespace's traffic is done via a block chain contract, as is the mapping from that router's key to it's IP address.

As a result of these contracts, a human readable URI maps reliably and securely to an IP address of a server to connect to, and the public key that server will use to prove its identity, without relying on any centralised services such as DNS.

7.4. 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 agent 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 agent. The agent 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 agent knows what the designated router's public key is in advance (it is obtained from the contract on the block chain).

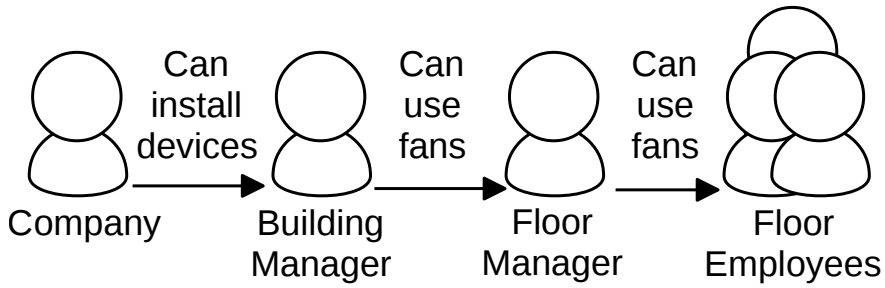
7.5. Permissions

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`/`, where K_C is a public key. 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 the block chain, in a smart contract that only admits DOTs whose signatures are valid.

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.

7.6. 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.

7.7. Control plane

In BW, an Ethereum-like block chain is used to handle control actions, such as storing DOTs, creating human readable aliases, mapping keys to IP addresses and other tasks. Originally, this role was served by a distributed hash table. While all the objects the block chain stores are intrinsically unforgeable as they are signed by the entity with the authority to create them, the use of a DHT introduced an attack vector that weakened the security model: if an object is added to the DHT it is not guaranteed to stay there forever, nor be visible to all agents. A block chain, in contrast, provides a much stronger guarantee. If an agent on the network knows it is up to date with the heaviest chain, then it knows it has the same complete view of control objects that every other agent has.

This allows us to improve the efficacy of operations such as revocations. If an entity or DOT is revoked, and the revoking agent waits for the operation to appear on the chain with enough weight following it (also known as confirmations), then that agent knows with high probability that every other agent on the network will treat the object as revoked for all future communications.

7.8. Applications

BOSSWAVE has been used for a number of services and applications. Existing devices are attached to the network by having a *proxy driver* that translates from the device's existing control protocol (such as a REST API or BACNET) to a control interface exposed as BW URIs. Typically, the connection between the driver and the device is on a trusted network, as most existing smart device APIs are notoriously insecure. Once the driver is running, however, all other control logic can communicate with it, over BW, from anywhere in the world without compromising security.

Aside from device drivers, persistent services are run to implement more complex logic that the devices themselves do not support. For example, a scheduler that uses sensor and occupancy data to adjust the HVAC settings of a building to minimize power consumption. While services can be run anywhere and on any platform, we have also constructed a framework around Docker containers, *spawnpoint* that allows a user to discover computational resources they have permission to use and automatically *spawn* an instance of a service or driver. The advantage of this framework is that the servers underneath spawnpoint can be located wherever is convenient (based on cost or perhaps data locality constraints) or even migrated, and the users are unaffected.

By merging spawnpoint with cryptocurrency micropayments on the block chain, BW-aware devices can automatically purchase computing power as and when they need it. This decouples a purchased IoT device from the services provided by vendors. A purchased device can continue to operate on the time

scale of building lifetimes (decades) even if the vendor stops selling the product or stops supporting them.

8. 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.

9. References

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

[2] 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, Springer, 2005, pp. 115–148.

[3] J. Polastre, R. Szewczyk, D. Culler, Telos: enabling ultra-low power wireless research, in: Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on, IEEE, 2005, pp. 364–369.

[4] J. L. Hill, D. E. Culler, Mica: A wireless platform for deeply embedded networks, IEEE micro 22 (6) (2002) 12–24.

[5] S. Greenberg, C. Fitchett, Phidgets: easy development of physical interfaces through physical widgets, in: Proceedings of the 14th annual ACM symposium on User interface software and technology, ACM, 2001, pp. 209–218.

[6] J. King, R. Bose, H.-I. Yang, S. Pickles, A. Helal, Atlas: A service-oriented sensor platform: Hardware and middleware to enable programmable pervasive spaces, in: Proceedings. 2006 31st IEEE Conference on Local Computer Networks, IEEE, 2006, pp. 630–638.

[7] T. Gu, H. K. Pung, D. Q. Zhang, Toward an osgi-based infrastructure for context-aware applications, IEEE Pervasive Computing 3 (4) (2004) 66–74.

[8] A. Wang, Y.-T. Huang, C.-T. Lee, H.-P. Hsu, P. H. Chou, Ecobt: Miniature, versatile mote platform based on bluetooth low energy technology, in: Internet of Things (iThings), 2014 IEEE International Conference on, and Green Computing and Communications (GreenCom), IEEE and Cyber, Physical and Social Computing (CPSCom), IEEE, IEEE, 2014, pp. 148–154.

[9] J. Beutel, O. Kasten, F. Mattern, K. Römer, F. Siegemund, L. Thiele, Prototyping wireless sensor network applications with btnodes, in: European Workshop on Wireless Sensor Networks, Springer, 2004, pp. 323–338.

[10] G. Aloi, G. Caliciuri, G. Fortino, P. Pace, A smartphone-centric approach for integrating heterogeneous sensor networks, in: Proceedings of the 9th International Conference on Body Area Networks, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014, pp. 247–252.

[11] G. Aloi, G. Caliciuri, G. Fortino, R. Gravina, P. Pace, W. Russo, C. Savaglio, A mobile multi-technology gateway to enable iot interoperability, in: 2016 IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI), IEEE, 2016, pp. 259–264.

[12] T. Zachariah, N. Klugman, B. Campbell, J. Adkins, N. Jackson, P. Dutta, The internet of things has a gateway problem, in: Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications, ACM, 2015, pp. 27–32.

[13] D. Locke, Mq telemetry transport (mqtt) v3. 1 protocol specification, IBM developerWorks Technical Library], available at http://www. ibm. com/developerworks/webservices/library/ws-mqtt/index. html.

[14] U. Hunkeler, H. L. Truong, A. Stanford-Clark, MQTT-S—A publish/subscribe protocol for Wireless Sensor Networks, in: Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on, IEEE, 2008, pp. 791–798.

[15] A. Carzaniga, D. S. Rosenblum, A. L. Wolf, Achieving scalability and expressiveness in an internet-scale event notification service, in: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing, ACM, 2000, pp. 219–227.

[16] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, M. Ward, Gryphon: An information flow based approach to message brokering, arXiv preprint cs/9810019.

[17] M. Hapner, R. Burridge, R. Sharma, J. Fialli, K. Stout, Java message service, Sun Microsystems Inc., Santa Clara, CA.

[18] B. Segall, D. Arnold, Elvin has left the building: A publish/subscribe notification service with quenching, Proceedings of the I997 Australian UNLX Users Group (A UUG'I997) (1997) 243–255.

[19] G. Cugola, E. Di Nitto, A. Fuggetta, The JEDI event-based infrastructure and its application to the development of the OPSS WFMS, Software Engineering, IEEE Transactions on 27 (9) (2001) 827–850.

[20] F. Tian, B. Reinwald, H. Pirahesh, T. Mayr, J. Myllymaki, Implementing a scalable XML publish/subscribe system using relational database systems, in: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, ACM, 2004, pp. 479–490.

[21] Y. Diao, P. Fischer, M. J. Franklin, R. To, Yfilter: Efficient and scalable filtering of XML documents, in: Data Engineering, 2002. Proceedings. 18th International Conference on, IEEE, 2002, pp. 341–342.

[22] M. Altınel, M. J. Franklin, Efficient filtering of XML documents for selective dissemination of information, in: Proc. of the 26th Int'l Conference on Very Large Data Bases (VLDB), Cairo, Egypt, 2000.

[23] P. Wyckoff, S. W. McLaughry, T. J. Lehman, D. A. Ford, T spaces, IBM Systems Journal 37 (3) (1998) 454–474.

[24] M. P. Andersen, G. Fierro, D. E. Culler, Enabling synergy in iot: Platform to service and beyond, in: 2016 IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI), IEEE, 2016, pp. 1–12.

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

[26] Z. Shelby, J. Nieminen, T. Savolainen, M. Isomaki, B. Patil, C. Gomez, IPv6 over BLUETOOTH(R) Low Energy, IETF RFC 7668 (Oct. 2015). doi:10.17487/rfc7668.
URL `https://rfc-editor.org/rfc/rfc7668.txt`

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

[28] D. L. Tennenhouse, D. J. Wetherall, Towards an active network architecture, in: DARPA Active Networks Conference and Exposition, 2002. Proceedings, IEEE, 2002, pp. 2–15.

[29] J. W. Hui, 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, ACM, 2004, pp. 81–94.

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

[31] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, T. Voigt, Cross-level sensor network simulation with cooja, in: Local Computer Networks, Proceedings 2006 31st IEEE Conference on, IEEE, 2006, pp. 641–648.

[32] T. Issariyakul, E. Hossain, Introduction to network simulator NS2, Springer Science & Business Media, 2011.

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

[34] S. Cheshire, M. Krochmal, DNS-based service discovery, Tech. rep. (2013).

[35] Z. Alliance, Zigbee specification (2006).

[36] MsgPack, http://msgpack.org/index.html (2015).

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

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

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

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

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

[42] L. Foundation, IoTivity, https://www.iotivity.org (2015).

[43] 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).

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

[45] G. Wood, Ethereum: A secure decentralised generalised transaction ledger, Ethereum Project Yellow Paper.