Title
Resource-Constrained Sensing as a Shared Utility

Author
Adkins, Joshua David

Publication Date
2023

Peer reviewed|Thesis/dissertation

Resource-Constrained Sensing as a Shared Utility

by

Joshua David Adkins

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering & Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Associate Professor Prabal Dutta, Chair
Professor John Wawrzynek, Co-chair
Professor Mani Srivastava
Professor Jan M. Rabaey

Fall 2022

Resource-Constrained Sensing as a Shared Utility

Abstract

Resource-Constrained Sensing as a Shared Utility

by

Joshua David Adkins

Doctor of Philosophy in Engineering - Electrical Engineering & Computer Sciences

University of California, Berkeley

Associate Professor Prabal Dutta, Chair

Professor John Wawrzynek, Co-chair

Cloud computing revolutionized the ease with which we can build, deploy, and scale distributed computing services. These advances, however, have not extended to the physically distributed and resource-constrained computers deployed throughout the world to collect data, and their resource constraints have thus far confined them to function as inefficient, fixed-purpose data forwarders. Breaking these distributed sensors free of their resource-constraints by including them in a dynamic, programmable, distributed system will not only enable easier deployment and scaling of applications relying on their data, but it will also give us the ability to collect and process never-before-seen data and discover new ways sensing the world around us.

We enable this vision in two parts. First we present a signpost-based platform which eases the building and deployment of sensors by providing the core services and hardware necessary for them to function. Next we explore the benefits of, and build a resource manager to form these resource-constrained sensors into a compute cluster akin to those found in the cloud. This enables multiple users to simultaneous program a cluster of sensors and quickly iterate on their programs through an application framework which abstracts away the details of scheduling and task distribution. By forming these sensors into a multiprogrammable cluster, we enable them to be accessed as a shared sensing utility rather than as a collection of individual nodes.

...

Acknowledgments

This work would not be possible without the many collaborators and supporters I've had over the last six years. I would like to start by thanking my dissertation committee, Jan M. Rabaey, Mani Srivastava who has been consistently supportive of the vision behind this work, John Wawrzynek who provided helpful late advising, and especially Prabal Dutta who has advised me for many years starting at the University of Michigan when I was an undergrad and continuing through my time at Berkeley. He has advised, debated, and entertained far flung thoughts while believing in this project. He has also supported my work on the other projects that are important to me along the way, and I am grateful for this flexibility. I am a much better researcher because of him.

Many people helped with the work that is highlighted in this dissertation. The Signpost platform, resource manager, and everything in-between were group efforts that do not exist in a vacuum. Branden Ghena, Neal Jackson, Pat Pannuto, Bradford Campbell, and Amit Levy put in many, many hours of implementation to make the Signpost work possible. They also listened to and debated the trends of low-power processors and wireless radios, eventually prompting the broad collection and comparison of these technologies over time. Joseph Noor was critical to designing and implementing the sensor resource manager, and Botong Ou helped significantly in making WASM work well on sensor nodes.

I would also like to thank everyone from Lab11, including Branden Ghena, Noah Klugman, Lane Powell (by proxy), Pat Pannuto, Neal Jackson, Meghan Clark, Bradford Campbell, William Huang, Ben Kempke, Sam DeBruin, Rohit Ramesh, Ye-Sheng Kuo, Thomas Zachariah, Matt Podolsky, and Jean-Luc Watson. Everyone who started at Michigan took me under their wing as an undergrad. Those who continued on to or joined in Berkeley were friends, collaborators, mentors, and co-conspirators for both the academic and less-academic parts of grad school. I certainly would not be here today without this community. I'd like to specifically call out Branden, who originally pulled me into the lab and was a consistent source of debate, idea generation, and kindness; Neal, who was my roommate for many years and is still my friend and a good support system even in the final days of writing; and Noah, who presented me with an opportunity that was both deeply distracting, but also critically inspiring in a time where my inspiration for the good that could be done with technology was severely lacking. He has since become a close friend in our path to purse that shared goal.

I am grateful to get to work with everyone at *n*Line. Knowing that a group of such thoughtful, intelligent, and conscientious people believe so deeply in our work gives me confidence in myself, and a large part of the motivation to finish this dissertation was the knowledge that I would be able to focus more deeply on the work with them on the other side.

Outside of academic circles, my friends have been there to consistently provide emotional support, refuge, and positive distractions throughout grad school. Jean, Paulina, and Genevieve have been my anchors through the many changes in my life over these years; the non-judgemental space they hold keeps me happy and balanced. I appreciate Michelle's listening, thinking with me, and ultimately showing me new ways of being as I thought

through what I wanted in my life and work. Anna and Stella always provide a place of relaxation that has been invaluable over the past few years.

Finally I'd like to recognize my family, especially my parents for their support in this process. Throughout my childhood they went above and beyond to support my passions with their belief, encouragement, and time. This encouragement lead me to becoming an engineer and continuing on to graduate school, and during this journey they have never pushed or questioned the path that I've chosen. I am grateful for everything they've done to help me get here.

* * *

This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and in part by Terraswarm, an SRC program sponsored by MARCO and DARPA. Additionally, this material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under grant numbers DGE-1256260 and DGE-1106400, NSF/Intel CPS Security under grant 1505684, and generous gifts from Intel.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing, and more broadly computing as an abstract utility, revolutionized the ease with which we can build, deploy and scale web-based computing services. Programmers can build services by using frameworks that facilitate the programming of complex distributed systems, take advantage of virtualization to amortize the cost of hardware management, and liberally deploy multiple versions of their program to simultaneously operate code in production and iterate quickly during development. All require compute easily accessible behind a consistent, reliable abstraction.

In contrast, it is currently not nearly as easy to place sensors in the world, collect data from them, and use the data they collect. Being physically close to the source of the data imposes constraints on the sensor's size, power, and communication options, and overcoming these constraints often requires an expert who understands how to build and optimize the sensor's hardware and software. After being built, these bespoke systems often serve as static and single-purpose data collectors. There is rarely a notion of sharing the data coming from a deployed sensor or dynamically reconfiguring a set of sensors based on a change in their context, and if updates are made it's often with high risk and requiring physical access to the sensors upon failure.

We want to enable a world in which people without deep technical expertise, like city planners, environmental scientist, building managers, and economists, could build or buy a sensor, deploy those sensors, and collect and share the sensor data. We also wish to enable a world in which embedded systems experts use sensors to build advanced applications rather than focusing on the painful process of manually reimplementing the resource optimizations required for basic functionality. To do this, we focus on two distinct aspects of the problem. First, we address the difficulty of reimplementing the core services necessary for a resource constrained sensor and its deployment, and then we build systems to make sets of these sensors function as reliable infrastructure, amenable to being re-tasked and shared as an abstract utility like the modern cloud.

We address the difficulty of building and deploying sensors by constructing a multi-tenant sensor platform that mounts on signposts around a city. The platform can be easily deployed by bolting to a standard street sign and uses wireless communication and solar energy harvesting

rather than their expensive and limited wired counterparts. After deployment, up to five sensors modules can then plug into the platform, sharing the platform's services—power, communication, location high-performance compute, time, and storage—without requiring their own implementations of these functions. Providing these platform services makes the sensors significantly easier to build, often possible with small modifications to off-the-shelf components. Sharing the platform also amortizes the cost of deployment and maintenance across multiple sensors.

Once the resources themselves are provided it becomes obvious that the availability of these key services is only one aspect of the challenge to collecting and using the data from a set of sensors. Due to the unaddressed resource constraints and concerns about reliability and privacy, data cannot be directly streamed from the sensors to the cloud for processing. At least some, and often quite a bit more, compute needs to be placed on the sensor to filter, transform, and fuse the raw data. This compute is often application-specific, and sometimes even specific to individual sensors within a broad deployment. Writing, deploying, testing, and iterating on such sensor-level code can be extremely challenging both for experts and non-experts alike.

The code is difficult to write because it often is comprised of many different applications coordinating within a sensor network. The code is difficult to test because it can often only be tested in situ, because sensor have unique access to the data required for testing. Iterative software development becomes difficult because code updates to sensors are quite risky given that they rarely run true operating systems which can isolate failures and receive a new program. A sensor hardware platform with only key services is akin to a data center with power and networking, but no common operating system, and no virtualization or tooling to help with building, deploying and testing code. Solutions have been proposed to these individual problems, but not in an architecture which addresses all of them simultaneously.

To truly ease the use and management of sensors distributed throughout our environment, we claim that we need to build tooling and programming frameworks analogous to those which run in the cloud, but which can operate within the sensor's resource constraints. This brings use to our thesis statement.

1.1 Thesis Statement

Enabling multiple, non-cooperative, full-stack applications to be deployed on distributed, resource-constrained sensors and their supporting infrastructure facilitates easier and more confident application deployment and iteration, and ultimately leads to higher-quality data collection from distributed sensors.

1.2 Contributions of the Dissertation

This dissertation articulates both the key problems of deploying and programming resource-constrained sensors and proposes solutions to those problems with the goal of making sensor networks easier to build, and dynamic, taskable, and more accessible, like today's cloud infrastructure, once deployed.

We begin with a signpost-based sensor platform which makes it easier to build and deploy sensors within cities. The platform changes the deployment strategy of city-scale sensors by shifting to use solar energy harvesting and wireless networking over their wired counterparts and it makes sensors easier to build by providing key services like networking, energy, and storage to pluggable sensor modules. We show that this deployment strategy can provide enough energy for common applications and sensing modalities, from tens to hundreds of milliwatts per sensor module, even in worst case scenarios, and that a sensor module's energy usage can be monitored and isolated from that of other modules. We then discuss the limitations of this platform and draw parallels between the needs of software multiprogramming and isolation, and the resource isolation and multi-tenancy that the platform provides in hardware.

In Chapter 3, we make both a quantitative and philosophical case for the deployment of more complex applications to resource-constrained sensors. We show that given the state of processors and wireless communication it is currently one to five orders of magnitude more efficient to perform common processing tasks locally on a sensor node rather than after transmitting data to the cloud; by analyzing processor and radio power trends we show this trade-off will increasingly favor local processing over time. We also discuss the opportunity local processing provides for the training of machine learning models by providing access to data that cannot be transmitted to the cloud and for the future reliability and privacy of sensor networks.

In Chapter 4, we list and categorize prior work on making sensors and distributed cloud environments easier to program, multiprogram, task, and manage. We show that while past systems have enabled most of the properties necessary for utility sensing, none have simultaneously offered all of the key components. Sensor-specific approaches successfully enable either the distributed programming of sensors or the multi-programming of sensors, but never both; architectures built targeting cloud servers, or access-points and other powered, gateway-class devices can enable all of the key components but cannot operate under the more severe resource constraints common to battery-powered, energy-harvesting, or more physically-distributed devices. This points to the need for a solution which can bridge the gap between these two domains.

We then propose a resource-manager in Chapter 5, which enables the simultaneous multiprogramming and distributed macroprogramming of sensors and cloud servers by multiple application frameworks. This resource manager takes advantage of advances in virtual machine and processor technology to enable multiple, isolated applications to run on a resource-constrained sensor, and all elements of the resource manager are reconsidered, from low-power networking protocols to sensor-specific resource abstractions, to enable the resource

manager to run in a distributed, resource-constrained environment. In addition to the resource management architecture, we propose runtime abstractions which enable applications to collect and efficiently share data from the same sensor on a sensor node. In testing we show that the resource manager enables multiple programming frameworks with multiple users to simultaneously task and reconfigure the sensor network and that the resource manager has an acceptably low energy and memory overhead to operate on energy-harvesting and battery-powered devices.

We end in Chapter 6 by giving an example of how application frameworks can be built for the resource manager. We discuss the core components of an application framework that allow it to uniquely adapt to distributed sensing. A proposal for *active scheduling* enables application frameworks to make scheduling decision in the face of dynamic environments and the heterogeneous performance of sensor and cloud nodes by deploying tasks specifically designed to collect information about the context and performance of a sensor. We then present the application frameworks that we implemented for the resource manager, including a streaming version of MapReduce, and reflect on the successes and shortcomings of these application frameworks. While we are not yet able to program sensor networks in high-level data processing languages used by domain scientists, the resource managers and application frameworks presented in this dissertation enable utility sensing by turning a network of sensors into reliable, retaskable, and shareable sensing infrastructure.

Chapter 2

Signpost: Sensing as a Shared Utility

Today, more than 50% of the world's population live in urban areas, and the U.N. projects that to increase to 66% by 2050 [1]. With increasing population density, there is growing interest in making cities safer, cleaner, healthier, more sustainable, more responsive, and more efficient—in a word, smarter. Supporting this interest are numerous funding opportunities [2–4], interested cities [5–7], and active research projects [8–11], all targeting new technology to enable smarter cities. And for good reason: applications such as pedestrian route planning based on air quality, noise pollution monitoring, and automatic emergency response alerts can all improve the quality of life for a city's inhabitants.

However, we believe that the difficulty of deploying existing smart city technology and applications is impeding progress. Deployments are rooted in single-purpose hardware, necessitating redesigns to support upgraded sensors or revised goals. Moreover, each system requires a re-implementation of standard resources such as power, communications, and storage, taking developer time away from the core application. Deploying sensors is difficult too, with the reliance on energy from wired mains constraining installation locations. These problems limit not only production-ready technology, but also make it particularly challenging to perform short-term, exploratory research, speaking to the need for a platform that will lower the barrier to entry.

To address these challenges, we present Signpost, a modular, energy-harvesting platform enabling deployable city-scale sensing applications. It mounts to pervasive sign posts (Figure 2.1) and harvests energy from a vertically mounted solar panel. To reduce the burden of developing new applications, Signpost provides commonly required services including power, communications, processing, storage, time, and location. The platform is modular, with eight pluggable slots for sensors, processors, and radios, facilitating modifications and upgrades to the system. To enable shared deployments, Signpost is multi-tenant, supporting multiple applications simultaneously and enforcing isolation between them.

Key to Signpost's deployability is its energy-harvesting, modular architecture. Harvesting energy enables the system to sever ties to wired infrastructure. This in turn opens up an increased selection of deployment locations, allowing for more granular deployments. Harvesting also enables short-term, pop-up deployments to drive application development and

Figure 2.1: The Signpost platform easily mounts to existing street sign posts, harvests from an integrated 0.1 mm solar panel, and provides tenant sensor modules with power, communications, processing, storage, time, and location. Signpost is open source, with all hardware and software available online.

experimentation. Support for modularity allows the sensors on the platform to be changed to suit application needs. More fundamentally, however, modularity permits Signpost to take advantage of future technology improvements, improving its capabilities over time.

An energy-harvesting, multi-tenant platform faces challenges that do not exist for mains-powered, single-purpose systems. For one, eliminating the connection to mains power limits the energy available for sensing. We assess the expected solar energy throughout the US, finding that a module can expect an average power of at least 120–210 mW for 50% of weeks. We also provide APIs allowing software to adapt to existing energy, reducing functionality in times of famine and opportunistically increasing it when possible. Another challenge is managing and sharing platform resources to support multiple stakeholders with unaligned interests. We explore the hardware and software requirements for measuring usage and enforcing isolation, describing guarantees necessary for sharing Signpost's limited energy budget between applications.

We envision a testbed of Signposts supporting short-term experimentation by many users. Signposts have been deployed on the University of California, Berkeley campus for five years. The ongoing deployment monitors weather, senses TV whitespace spectrum usage, and observes vehicular traffic. We have found Signpost modules are generally easy to create and the software API is simple to implement on commonly used software and hardware platforms such as Arduino and ARM Mbed. To facilitate the creation of new hardware and software

to run on Signpost, we have also created desktop development kits capable of emulating deployed behavior. We hope that by providing a platform for city-scale sensing that reduces the barriers to deployable applications, supporting that platform with development tools and accessible interfaces, and working with the community to realize their sensing needs, we can gain deeper insight into the workings of urban areas and enable higher-level applications that impact policy and quality of life throughout a city.

2.1 Prior Shared Sensing Platforms

Existing work in urban sensing generally falls into three categories: static deployments of sensing applications, mobile or human-based participatory sensing, and—most similarly to Signpost—deployments of generic sensing infrastructure. The first two categories are particularly insightful as a guide to which services are frequently needed by existing applications, which we summarize in Table 2.1.

Examples of static deployments include acoustic sensors to monitor, characterize, and localize different sounds [8, 11, 12], particulate sensors to monitor air quality [10], and electromagnetic, radiological, and meteorological sensing to track people [13] and cars [14], measure road conditions [15, 16], monitor wireless traffic [17], locate point sources of radiation [18], and identify severe weather in urban environments [19]. Most deployments are not long-term and are only deployed for the purposes of evaluation. Additionally, almost all of these deployments depend on either mains power or a battery for an energy source, motivated by the desire for rapid prototyping. Many of these deployments use a proof-of-concept node design built mostly with off-the-shelf components, without much consideration for optimized energy consumption [8, 11–13, 15, 16]. By providing a platform that already handles energy-harvesting, Signpost could provide sustainability to these deployments. Further, based on reported power numbers, with the exception of the high power Micronet nodes [19, 20], many of these applications and experiments could run on the Signpost platform without significant redesign.

The majority of work that targets urban sensing uses participatory methods, in which users participate with mobile phones and other handheld devices [22, 23], or vehicles are outfitted with various sensors [24, 25]. These methods use existing mobile resources to collect similar data to static deployments. Many have paired mobile phones with handheld air quality monitors [10, 26–28], or used phones to directly meter noise pollution [29–31] or traffic conditions [32–34]. Similar to participatory sensing methods, vehicular sensor networks monitor air quality, traffic, and road conditions [25, 28, 35–37], and even detect rogue cellular base stations [21]. These types of deployments often scale very well as the mobility of the devices allows a few sensors to reach a much larger area. However, incentivizing participation can be difficult and coverage can be unpredictable and potentially insufficient.

Finally, several platforms provide generic sensing infrastructure, suitable for many types of smart city applications. CitySense proposes an open, city-scale wireless networking and sensor testbed [38]. It utilizes mains-powered, street pole mounted embedded Linux nodes with 802.11 mesh networking and enables in-situ node programming by end users. Argos, a passive

Deployment	Energy	Network	Processing	Storage	Time	Sync	Location
Caraoke [14]	✓	✓			✓		
Bouillet et al. [15]	✓	✓					
AirCloud [10]	✓	✓					
Girod et al. [12]	✓	✓	✓			✓	✓
Lédeczi et al. [11]	✓	✓	✓			✓	✓
SenseFlow [13]	✓	✓					
Argos [17]	✓	✓			✓		
SONYC [8]	✓	✓	✓	✓			
Kyun Queue [16]	✓	✓		✓	✓		
Micronet [20]	✓	✓		✓			
Seaglass [21]	✓	✓		✓	✓		✓

Table 2.1: Services required by existing applications. Time is millisecond-accurate as provided by services like NTP, while Sync is microsecond-accurate as provided by GPS. Location is GPS-level accurate coordinates. These represent the minimum services a platform should provide to support existing applications and simplify the creation of new ones. Many of these applications could run on Signpost without significant modifications.

wireless mapping application, builds on a 26 node CitySense deployment [17]. Unfortunately, the CitySense architecture met many logistical challenges that ultimately limited a scaled deployment [39]. The Array of Things project utilizes a network of sensor nodes distributed throughout Chicago to gather environmental data including light, temperature, humidity, and air quality [9]. Like CitySense, Array of Things sensor nodes assume wired power and networking, and thus must be installed in locations where these resources are present. Signpost also provides an open testbed for smart city research. However, through its focus on deployability and modularity, Signpost reaches a different design point than these projects, resulting in a resource-constrained, energy-harvesting, and multi-tenant platform that is more easily deployed, but potentially more challenging to program.

2.2 Platform Overview

In the following sections, we present the design, implementation, and evaluation of Signpost, a modular, solar energy-harvesting, sensing platform. In the Signpost platform, sensor hardware connects to a shared backplane via a standard electrical and mechanical interface, enabling modularity. The backplane serves as the module interconnect and has the ability to electrically isolate each module, allowing energy use of any particular module to be limited. To support these sensor modules, the platform harvests solar energy, monitors a shared battery, and distributes metered power. It provides multiple radio interfaces for different communication patterns and shares them among the modules. Other services are implemented as well, including time and location, data storage, and compute offload using a Linux-class co-processor, and these services can be accessed by modules through a standard software

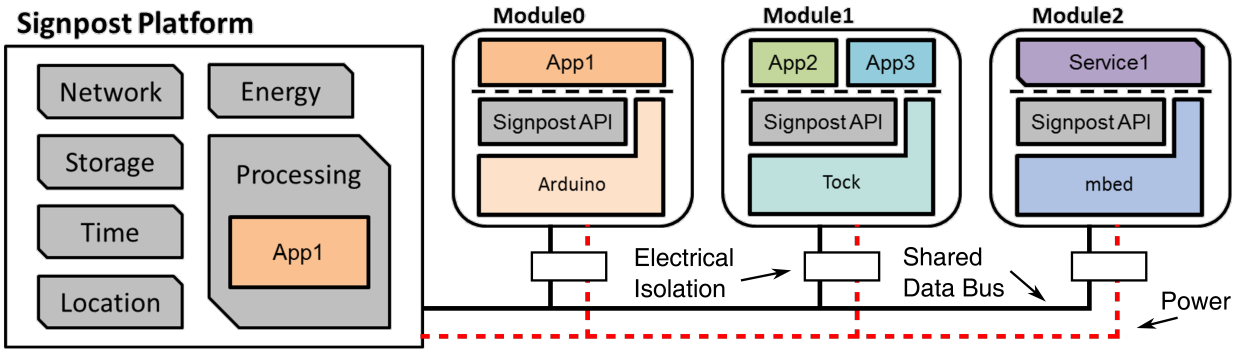


Figure 2.2: Signpost platform overview. Signpost monitors and distributes energy to connected modules and provides shared networking, Linux processing, storage, time, and location services. Modules implement one or more sensing modalities and utilize many possible software stacks, running one or more applications or even providing additional services to the platform. Applications can potentially be distributed across the platform and modules. This platform design supports development and deployment of urban sensing applications.

API. Resources and modules are orchestrated by a microcontroller-based system controller that oversees the operation of the Signpost platform. All of these components are housed in a waterproof aluminum case that to bolts to a standard street sign post for easy deployment. Figure 2.2 shows an overview of the platform.

2.3 Platform Design Requirements

The Signpost platform's design is guided by four high-level goals:

- **Deployability** is the primary concern of the platform and is key to enabling larger and more frequent deployments, and ultimately wider adoption by the community.
- **Accessibility** reduces burden for developers, thus the platform needs to provide services that meet common application needs.
- **Modularity** allows developers to modify and extend sensing capabilities to support new applications and upgrade modules as technology improves.
- **Multi-tenancy** enables the platform to simultaneously host mutually-untrusting applications created by multiple stakeholders, reducing deployment burden and the cost of experimentation.

Deployability

Deployability is the primary concern for the Signpost platform. Many urban sensing applications require fine-grained sensing, which is not possible for platforms that can only be

deployed with easy access to mains power or wired networking. Additionally, to support ad-hoc experimentation, the platform needs to be easily installed, removed, and moved. A deployment made today may not meet the sensing needs of an application tomorrow.

In order to enable deployability, Signpost does not depend on mains power or wired networks. Relying on wired infrastructure would limit Signpost deployments to locations with grid access, such as the top of streetlight poles, and would require costly and time-consuming installation by city utility workers. To support easy physical installation, the platform attaches to existing infrastructure found ubiquitously in urban areas—sign posts.

Making these deployability decisions allows Signpost to better support some applications while restricting others, particularly applications with high power sensors, significant bandwidth needs, or heavy computation. To address these concerns, the platform needs to provide software primitives that enable applications to adapt to available energy and bandwidth. Even if these primitives prove insufficient, we believe that in time most applications will still become possible on Signpost due to the rapid power scaling of embedded hardware. In the last decade alone, best-in-class microcontroller active current has decreased from 220 µA/MHz to 10 µA/MHz [40, 41], radio transmission power has reduced by 3-5x [42, 43], and many sensors have followed similar trajectories. By embracing modularity, hardware can be updated to capitalize on these improvements, with the tradeoff between deployability and resource constraints increasingly favoring the Signpost architecture.

Accessibility

Informed by a review of prior sensing projects in Section 2.1, Signpost provides several services to support accessibility and reduce the burden for application developers.

Energy

Since wired mains power is not an option for Signpost, we turn to batteries and energy harvesting to power the system. Batteries alone may be sufficient for short-term research deployments, but replacement is not scalable for geographically distributed deployments. Instead, a battery would need to store enough energy for the entire deployment duration. Assuming a 1 cm thick Li-ion battery the size of the Signpost solar panel (0.096 m^2) yields a storage capacity of 576 Wh [44]. For one year of lifetime, this would result in an average platform power budget of 66 mW.

The expected budget can be improved significantly with the addition of solar energy harvesting. An optimally oriented, 17% efficient solar panel with the same area as Signpost's would generate 2.4 W on average indefinitely in Seattle, a city with notably poor solar conditions [45]. Even with vertical panel placement and sub-optimal panel orientation, the addition of energy harvesting yields an increase in energy provided to the platform as we demonstrate in Section 2.5, resulting in increased application capabilities.

Communications

Signpost needs to support periodic data transmissions, firmware updates, and occasional bulk data uploads. Coverage is needed over a wide area and neither wired network nor WiFi access points can be expected to be accessible for all deployed Signposts. One solution to these problems is cellular radios, especially the machine-to-machine focused LTE Cat-1, LTE-M, or NB-IoT networks. Cellular networks provide high throughput and good coverage, but also come with costs, both in terms of high power draw and network usage fees.

Alternative solutions include low-power, wide-area networks such as LoRaWAN [46], which provides data transfer at rates of 1-20 kbps with a range of several kilometers and power draw significantly lower than cellular radios. LoRaWAN networks can be deployed by end users, allowing a network to be set up to support a Signpost deployment. However, LoRaWAN predominately supports uplink communications, making firmware updates and other downlink-focused applications more difficult.

Finally, local communication facilitates interactions between a Signpost and any nearby residents or users of the platform. Communication protocols such as Bluetooth Low Energy would enable the platform to interact directly with nearby smartphones.

Processing

In nearly any sensing system, data must be processed, batched, transformed, and analyzed, and in the face of energy constraints, local computation is preferable over transferring all data to the cloud. Providing a processing service is not necessarily just about computational capability. A familiar processing environment in which developers can use familiar languages and libraries lowers the barrier to entry for domain scientists.

Many existing urban sensing platforms provide processing by using some variation of a Linux computer as their primary processor [8, 9, 17, 18, 38]. For an energy-constrained system, however, supporting an always-on Linux computer is problematic. Even the lowest power Linux compute modules we survey draw 200-500 mW when active [47]. One compromise is to use a Linux environment not as a core controller, but as a co-processor, employed occasionally to process batched data. This allows developers to use languages and libraries to which they are accustomed, but requires them to split applications between two execution environments.

Storage

With low power and low cost flash memory widely available, data storage could be a module-supplied resource. However, we argue it should be centralized on Signpost for two reasons. First, a central data store aids manual data collection (likely over a short-range wireless link). This is useful for collecting high-fidelity data from multiple modules, particularly in the early experimentation phases of a deployment. Second, co-locating the central storage with shared processing resources allows for fast and easy access to batched data.

Time and Location

Synchronizing clocks throughout a sensor network deployment is critical to many applications [48]. Providing the capability to synchronize within 100 ns allows a group of Signposts to achieve localization within 30 m for RF signals and less than one meter for audio signals. In addition to just synchronization, the ability to timestamp data and understand the local time of day and year is useful for adapting operation (for example, slowing sampling before night) or predicting available solar harvesting energy. Location also provides automatic installation metadata and enables localization-based applications, such as gunshot detection. Fortunately, all are easily provided by GPS modules, although some care needs to be taken when expecting GPS use in dense city environments where fewer satellites may be in line-of-sight of the receiver. The addition of a stable and low power real-time clock can act as an optimization for a time and location system on a stationary platform by allowing the GPS to be predominantly disabled. This reduces system power draw while maintaining sufficient accuracy for many applications.

Modularity

Modularity enables not only specialization, but it also allows the platform to be upgraded over time, adapting to technology improvements for sensor modules and platform resources alike. Supporting modularity requires standardized electrical and mechanical interfaces to allow sensor modules to be installed and replaced as needed. The electrical interface should be simple but sufficient, including connections to power and an internal communication bus over which modules access platform services. Other signals can be added to support performance, for example a time synchronization signal, but such additions should be kept to a minimum to keep module creation simple.

Regarding mechanical considerations, the interface must allow for a robust connection to the physical platform without significantly limiting sensing capability. Weatherproofing plays an important part in the design of this interface since Signpost will be deployed outdoors, as does physical security since platforms will be unattended for long periods. Additionally, sensor module developers should be able to easily tailor the module enclosure to support the physical and environmental requirements of their sensors.

Multi-tenancy

Finally, Signpost is designed to support multiple stakeholders simultaneously, allowing a single hardware deployment to act as a testbed for multiple applications. Support for multi-tenancy requires fair sharing of resources between applications. For most system services, this reduces to platform software recording usage and implementing some fairness policy.

Sharing energy is a more complex problem and the top priority of a multi-tenant, energy-harvesting system [49]. The power requirements of one application should not limit the capabilities of another. To support this, a platform must first be able to accurately measure

and control access to energy. This involves metering not just modules, but also system resources, so that their energy draw may be charged against the application which accessed them.

Second, the platform must use these measurements to implement an energy policy. In the presence of variability, applications need guarantees of energy availability to reason about future processing capabilities. There is one important guarantee: the energy allocated to an application must only decrease in a predictable fashion. It can be spent directly by the application, indirectly by a service the application uses, or taken regularly as a platform tax, but it must not decrease in a manner unpredictable to the application. Particularly, energy should never be taken to support other applications (although it could be given). If energy is harvested by the platform, the allocation of a particular application may increase, but having a minimum known energy to rely on allows applications to plan for future actions. Support for energy isolation has been explored in prior work [49].

Features to support multi-tenancy have an added benefit in supporting overall system reliability. Modules can be isolated from the platform entirely if a hardware or software failure occurs.

2.4 Platform Implementation

The Signpost architecture is shown in Figure 2.3. The Signpost platform is defined by the Power Module, Control Module, Backplane, and Radio Module. Additional modules connect via a standard electrical and mechanical interface. A full Signpost has six general-purpose module slots, one of which is taken by the Radio Module, leaving five for sensing capabilities. The size of the entire system, including a case, is 42.9 cm high, 30.0 cm wide, and 8.4 cm thick. For comparison, the minimum size of a speed limit sign in the United States is 91 cm by 61 cm [50].

Backplane

The Backplane is the backbone of the Signpost. It has physical and electrical connections for modules, signal routing between modules, and isolation hardware. The Backplane has eight slots in which modules can be connected. Two are special-purpose, corresponding to dedicated signals for the Power Module and Control Module. The remaining six are standard interfaces for modules. The interface provides power at 5 V, access to a shared I^2C bus, two dedicated I/O lines to the Control Module, a Pulse Per Second (PPS) signal for synchronization, and a USB slave connection.

Modules are not required to implement all signals in this interface. However, we expect that most modules will use the I^2C bus and dedicated I/O signals, and that some complex modules will implement USB or PPS support.

All module connections can be individually isolated, along with buffering for I^2C connections. These isolators can be activated by the Control Module and prevent individual

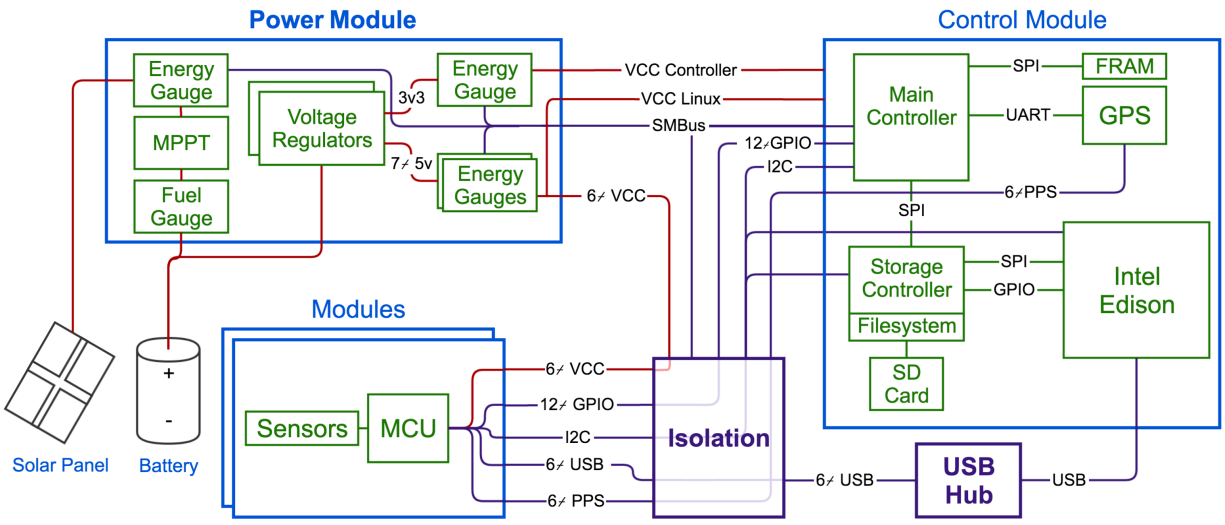


Figure 2.3: Signpost architecture. The Power Module is capable of harvesting energy from a solar panel, storing energy in a battery, supplying power at the correct voltage to modules, and monitoring the energy use of modules. The Control Module provides storage, time and location, and Linux processing services, and also monitors modules with the capability of isolating them from the system if necessary. Finally, there are the modules themselves, with many possible capabilities. This architecture allows for modular and extensible sensing while minimizing deployment complexity.

modules from negatively impacting the rest of the Signpost. The Backplane also accepts a voltage reference signal from each module and handles translation of voltage levels for all signals except USB, allowing modules to perform I/O at any voltage between 1.65 V and 5 V.

Power Module

The Power Module is responsible for energy harvesting, management, monitoring, and distribution on the Signpost platform. Energy is harvested from a Voltaic Systems 17 W solar panel, a 37 cm by 26 cm panel with an expected 17% efficiency. The solar panel output is monitored by a coulomb counter, and regulated by a maximum power point tracking battery charger. Excess energy is stored in a custom 100 Wh Li-ion battery pack.

System energy is further regulated for consumption before being distributed to the Backplane and modules. Each regulator can provide a constant 1.5 A, and is protected from shorts by a load switch. Each module's power rail is monitored by a coulomb counter that also provides instantaneous current readings, supporting energy accounting.

The Power Module also includes a hardware watchdog that monitors the platform. This further increases Signpost reliability by providing a redundant watchdog in the event of software failures.

Control Module

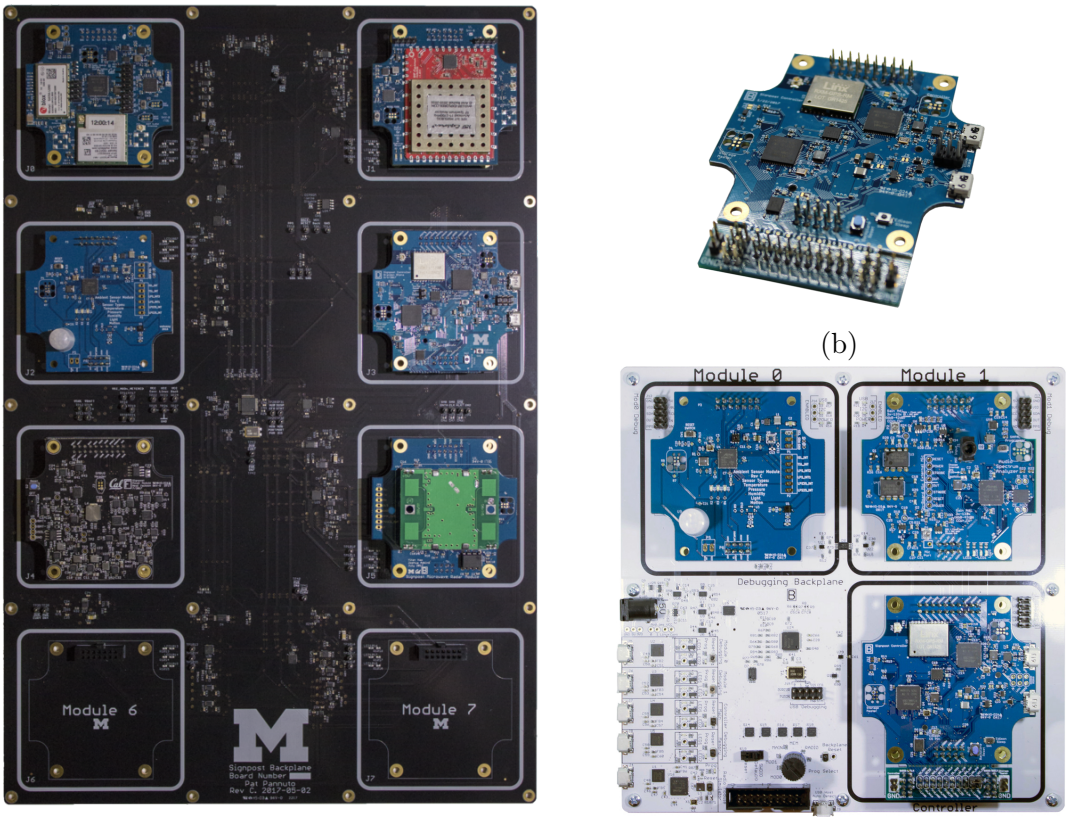
The Control Module handles system tasks, such as managing the module energy usage, assigning module addresses, and monitoring system faults. It also provides time, location, storage, and processing services to the sensor modules. Computation is handled by two ARM Cortex-M4 microcontrollers.

One microcontroller is responsible for isolation, managing the GPS, and accounting for module energy. It can also communicate with sensor modules on the shared I^2C bus and through dedicated per-module I/O signals, sending information such as location and time to the sensor modules in response to Signpost API calls. A globally synchronized Pulse Per Second signal is routed from the GPS to all sensor modules. The second microcontroller is responsible for managing an SD card and providing the storage API to the sensor modules. Each of these subsystems is power gated and can be entirely disabled to save energy.

Finally, the Control Module has an Intel Edison Linux compute module for higher performance processing capabilities. Contrary to common system design, while the Edison is the most capable computer on the Signpost, it is not in control of the system. Instead, the Edison is a coprocessor, capable of batch processing and using languages and libraries that are difficult or impractical to port to embedded microcontrollers. The Intel Edison connects directly to modules over USB, with each module playing the role of a USB slave device. It can also communicate with modules over an internal SPI bus by using one of the Cortex-M4s on the Control Module to forward messages to the shared I^2C bus. The power usage of the Edison is individually monitored, allowing its energy to be attributed to the module utilizing its services.

Radio Module

The Radio Module provides communications services to the Signpost. To handle diverse communication needs, it hosts cellular, LoRa, and BLE radios. An ARM Cortex-M4 microcontroller handles receiving messages through the shared I^2C bus or via USB from the Intel Edison and sending them to the appropriate radio interface. A U-blox SARA-U260 cellular radio is capable of both 2G and 3G operation at up to 7.2 Mb/s. However, it draws up to 2.5 W in its highest throughput modes [51]. A Multitech xDot radio module provides LoRaWAN communications. Sending data through LoRaWAN is more sustainable from an energy budget standpoint, with the module drawing less than 0.5 W in its highest power state [52]. Finally, the Radio Module includes an nRF51822 BLE SoC. This enables Signpost to send real-time data about the environment to nearby smartphones. Providing three communications interfaces allows Signpost to make decisions about which radio to use based on quality of service, latency, throughput, and energy requirements.



(a) (b) (c)

Figure 2.4: A populated Backplane (a), Control Module (b) and Development Backplane (c). The Backplane serves as the Signpost interconnect, while the smaller Development Backplane is the desktop equivalent, enabling easy module and application creation and testing. The Control Module manages Signpost energy and provides services to sensor modules. Existing sensor modules are also shown, with the RF spectrum and radar modules at the top and bottom right of the populated Backplane respectively, and the environmental and audio sensing modules on the top left and top right of the Development Backplane.

Sensor Modules

Four sensor modules have been created for Signpost and are in use. The existing modules perform ambient environmental sensing (temperature, humidity, pressure, and light), monitor energy in seven audio frequency bins ranging from 63 Hz to 16 kHz, measure RF spectrum usage within 15 MHz to 2.7 GHz, and detect motion within 20 m with a microwave radar. Each was made by a different student, including two undergraduates. All of the sensor modules and the Signpost Backplane are shown in Figure 2.4.

Service	System Call	Description
Init	`i2c_address = module_init(api_handles)`	Initialize module
Network	`response = network_post(url, request)`	HTTP POST data to URL
	`network_advertise(buf, len)`	Advertise data over BLE
	`network_send_bytes(destination, buf, len)`	Send via best available medium
Storage	`record = storage_write(buf, len)`	Store data
Energy	`energy_info = energy_query( )`	Request module energy use
	`energy_set_warning(threshold, callback)`	Receive energy usage warning
	`energy_set_duty_cycle(duty_cycle)`	Request duty cycling of module
Processing	`processing_call_rpc(path, buf, len, callback)`	Run code on Linux compute
Messaging	`messaging_subscribe(callback)`	Receive message from a module
	`messaging_send(module_id, buf, len)`	Send message to another module
Time	`time_info = get_time( )`	Request current time and date
	`time_info = get_time_of_next_pps( )`	Request time at next PPS edge
Location	`location_info = get_location( )`	Request location

Table 2.2: Signpost API examples. Abstract versions of several Signpost API calls for each system service are shown. Providing a high-level API enables easier application development.

Module Software

To enable access to the resources on Signpost, we provide APIs for applications that abstract away the specific details of messages sent over the shared I^2C bus and allow module creators to write software at a higher level. Abstract versions of several API calls are listed in Table 2.2, including calls to allow module applications to POST data, write to an append-only log, be automatically duty-cycled, start processes on the Intel Edison, and send messages to other modules.

All API calls are layered on a minimal intra-Signpost network protocol. The library code is written in C on top of a hardware abstraction layer requiring I^2C master, I^2C slave, and GPIO implementations. We implement the library using the Tock operating system [53] for our own development purposes and have ported the library to the Arduino [54] and ARM Mbed [55] stacks to support a wider array of module designs.

Signpost supports multiple views on what it means to be an application. A module may run one or more applications, and an application may be constrained to a single module, include processing code run on the Intel Edison, exist logically across several modules connected by the messaging API, or even across Signposts distributed around a city, connected by wireless communications. An example of the Signpost software model is shown in Figure 2.2 where one or more applications are running on heterogeneous sensor modules and accessing Signpost services through a common API.

Development

In addition to the full, weatherproofed Signpost platform, a development version of the system aids in creating and testing module hardware and software, as shown in Figure 2.4.

The development Signpost supports two modules and a Control Module. While meant to be wall-powered, it has the same isolation and monitoring hardware as a full Signpost, allowing it to emulate various energy states, track module energy use, and disable modules when they exceed their allocation. Rather than including radios, the development Signpost implements the radio API, but sends data over a USB serial connection instead of an RF link. Identical Control Module and Backplane hardware is used on both systems, allowing desktop experimentation with applications that is faithful to deployed system.

2.5 Evaluation of Resource Sufficiency and Sharing

We evaluate the key claims of the Signpost platform, including deployability, the implications of a deployable design on energy availability, and the ability to support multiple applications. We also benchmark several Signpost services. Finally, given these capabilities and constraints, we explore the types of applications capable of running on Signpost and how they interact with system resources.

Deployment Metrics

A primary goal of the Signpost platform is deployability, and over the course of nine months we deploy the platform on over 50 occasions, for varying lengths of time, at several locations. In all of these deployments, we found Signpost to meet our deployability goals in both speed and effort.

Specifically, we find that two students can deploy a single Signpost in less than five minutes. In a specific case, it took less than 90 minutes to walk and deploy twelve Signposts across a portion of the UC Berkeley campus. Although we take no precautions, the deployments have experienced no vandalism or theft, even with Signposts placed near a popular concert venue in an area with relatively high property crime. We believe that this indicates the platform is unobtrusive and blends in with other city infrastructure. Approval for these deployments, while sometimes slow for bureaucratic reasons, has been simple due to the non-destructive, attachment method. While this level of deployability comes at the cost of energy availability, a system with these properties greatly facilitates ad-hoc experiments and highly-granular long term sensing applications.

Signpost Energy

This focus on deployability makes energy availability a fundamental challenge for Signpost. We investigate the overhead of multi-tenancy and expectations for how much energy Signpost can harvest.

Platform Overhead

While supporting city-scale sensing is the purpose of Signpost, not all energy goes directly to applications. In particular, multi-tenancy and platform services each incur overhead. These costs can be primarily attributed to the static power of the regulation and monitoring hardware, which have a total quiescent power draw of 13.2 mW. The components for module isolation draw an additional 1 mW, as do the microcontrollers on the Control Module, on average.

Additionally, the over-sized charging and regulation circuitry has a lower efficiency than similar circuitry designed to match the requirements of a single-purpose sensor. We measure the battery charging efficiency to be 85% at a wide range of power inputs, and the regulator efficiency to be 89% at all but the lowest power draws. Across the platform, this totals to 76% efficiency and a base power draw of 16 mW, less than 2% of the 50th percentile average power budget and 6-18% of the 95th percentile budget. We believe this is an acceptable overhead for the advantages of multi-tenancy.

Services provided by the Control Module, such as storage and location, are power gated when not in use and do not contribute to the static power of the platform. If applications request these services, their energy is attributed to the sensor modules using them. We find the Intel Edison Linux module draws 15–24 mW in sleep mode, the GPS chip draws 40 mW when tracking satellites, and the Radio Module sleeps at less than 1 mW. The SD card is enabled on demand, and therefore has no idle power draw.

Harvesting

A key enabler of deployability is the shift to a solar energy-harvesting power source. To further increase deployability, it is preferable to make no assumptions about solar panel positioning, and therefore expect the panel to be deployed vertically facing an arbitrary direction. We evaluate the expected energy availability under these constraints in different locations, solar panel directions, and times of year.

We start this evaluation by deploying four solar panels on sign posts in Ann Arbor, Michigan, with one panel pointing in each cardinal direction. A building is located to the south of the posts and a small hill directly west. For each panel, we record the open-circuit voltage and short-circuit current at ten second intervals and estimate the power output of the panels by assuming an 80% fill factor. Figure 2.5 shows the output of this experiment for one week in July 2016 and one week in March 2017. We present both the instantaneous output of each solar panel and the daily averages.

This experiment shows that the power availability of a Signpost is highly variable, ranging from over 3.08 W for the south facing panel on March, 22nd to only 219 mW for the north facing panel on March, 25th. We find that the direction, season, and degree of cloud cover all contribute to this variability. While some of the variability can be buffered by the battery, variability will inevitably be experienced by applications running on Signpost.

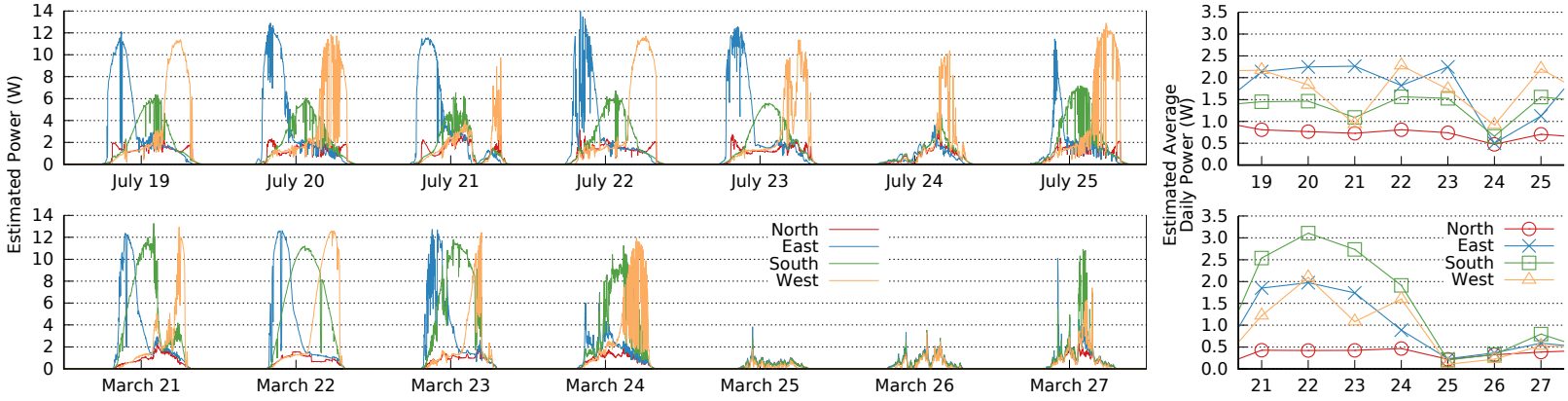


Figure 2.5: Solar harvesting in four different cardinal directions and two seasons. The experiments are run in July 2016 and March 2017 in Ann Arbor, Michigan, with each including periods of both sunny and cloudy days. At left is estimated power generated from solar panels mounted vertically in four cardinal directions captured in 10 second intervals over a week. At right is the average daily power provided by each solar panel. There are large variations in average power both due to direction and daily weather patterns. While some daily variations can be buffered by the battery, Signpost will still experience variability in available energy to which it must adapt.

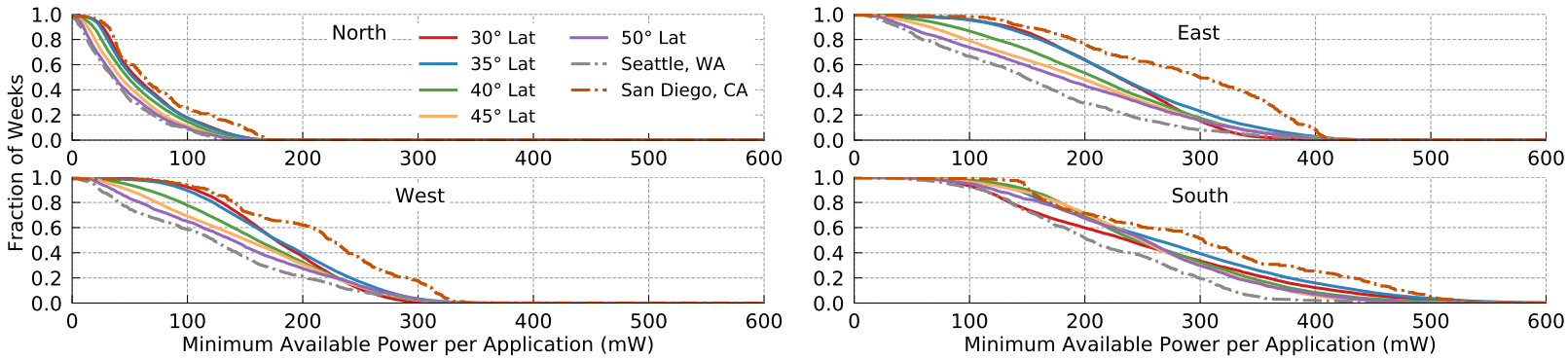


Figure 2.6: Fraction of weeks when an application can expect a minimum power income at different latitudes and cardinal directions. To evaluate how much power a Signpost application can expect under varying deployment conditions, we model the solar harvesting potential of a vertical Signpost facing the four cardinal directions across the United States. We use a standard solar model that accounts for both direct and diffuse light [56] along with hourly irradiance data from the NREL MTS2 2005 dataset [57]. We group these locations by latitude, and also plot distributions for Seattle, Washington and San Diego, California, where local weather patterns create poor and near-ideal solar harvesting conditions, respectively. The per application expected minimum power is calculated by subtracting the static power draw (16 mW) from the weekly average harvested power, dividing among an expected five applications, and multiplying by the regulator efficiency (76%). We find that orientation generally has a stronger influence on harvested energy than latitude or climate.

To more broadly determine the expected power budget for a sensing application running on Signpost, we create an energy availability model that predicts the average weekly power available to Signposts at different geographic locations in the United States throughout the year. The model is based on hourly direct and diffuse light measurements at 1,500 locations around the United States from the NREL MTS2 dataset [57], and these measurements are converted into expected power output using a standard harvesting model for tilted solar panels which takes into account solar panel direction, angle, and the harvestable portion of diffuse light [56]. We compare our model with the experimental data shown in Figure 2.5 and find the model strictly underestimates our experimental results by an average 3.3% on sunny days and 22% on cloudy days. We believe this error primarily can be attributed to diffuse light collection for north-facing solar panels, a scenario that is not well studied in solar modeling literature.

The results of this model are displayed in Figure 2.6 as the fraction of weeks at which an application will have a minimum available power. To generate this plot, we group the weekly average power data by latitude, subtract the platform overhead and regulator efficiency losses discussed in Section 2.5, then divide by an expected five applications (assuming one for each available module slot on Signpost). In addition to showing data for each latitude, we also plot energy available in Seattle, Washington and San Diego, California, which are particularly poor and ideal solar energy harvesting locations, respectively, in the United States. We see that the 95th percentile of available weekly average power ranges from 3.84 mW per application for a north facing Signpost in Seattle, WA to 147 mW per application for a south facing Signpost in San Diego, CA.

We conclude that, in general, the direction at which Signpost is placed impacts available energy more than the latitude of the platform. This creates a tradeoff between deployability and energy availability. While it is possible to entirely ignore orientation when deploying Signposts, this comes at the cost of expected energy for some of the deployed systems. Putting in care to avoid facing north when possible may be a sufficient compromise.

One aspect which is not included in the prior evaluations is potential shading from nearby obstructions. This is a particularly real concern in urban areas where buildings are expected to obstruct direct sunlight for portions of each day. The amount of shade a Signpost can expect is, however, particularly deployment-specific and difficult to predict in a general fashion. For example, due to its vertical orientation, even with a building directly to its east a west facing Signpost can expect to harvest most of its predicted clear-sky energy. In our deployments, we have found that Signposts deployed under moderate, continuous shade (under a tree in this case) see harvested energy similar to a north facing, clear-sky Signpost.

Managing Multi-tenancy

Signpost expects to host not just a single application, but several. Here, we evaluate how the system responds to multiple demands to its resources simultaneously.

Energy Isolation

The primary resource that must be shared between all applications is energy. On Signpost, we virtualize stored energy, making it appear to each application that they have independent batteries. Stored energy in the battery is split into a "virtual allocation" for each application. A virtual allocation is guaranteed to never deplete except when predictably spent. For example, it will never be taken to support another application's needs. This allows programs to plan and make decisions based on available energy that are independent of the actions and needs of others.

On an energy-harvesting platform, an additional question arises in how to distribute incoming energy. A fair model distributes energy equally between applications, but there must be a maximum allocation for each. If an application stores the energy it is given but does not use it, its allocation would eventually expand to the entire capacity of the battery. Instead, we define a maximum capacity for each virtual allocation. Harvested energy is then divided between applications that are below maximum capacity. This adds variability to the amount of energy an application receives based on the actions of other applications running on the platform. However, this variability is no worse than the variability inherent to energy harvesting systems in the first place. Policy choices and support for energy isolation are discussed further in another work [49].

Figure 2.7 demonstrates energy sharing in practice. Three modules are installed on one Signpost, each running a single application and given virtual allocations with a maximum capacity of 10,000 mWh. Data is shown for a 20 hour period, from night to night. The deployed Signpost has a building directly to the east, only allowing it to harvest later in the day. Displayed are the five-minute average power draws for each application and the net power into the battery. Energy allocations are also reported every five minutes for each module and the battery.

Each application has a different strategy for energy use. The first heavily duty-cycles itself and is active for only a brief period every ten minutes. This results in an average power draw of less than 4 mW, and consequently its virtual allocation stays near or at maximum capacity the entire time. The second application continuously draws 250 mW, an amount that cannot be sustained while the Signpost is receiving no direct sunlight. It eventually exhausts its allocation and is disabled by the platform. Later in the day, when energy is being harvested, it is allocated a portion of incoming energy and resumes operation. The third application adapts to the amount of energy available to it, remaining in continuous operation. Its power draw increases when the solar panel receives direct light, corresponding to an increase in sampling rate in the application. As this experiment demonstrates, Signpost is able to isolate the energy needs of applications from each other.

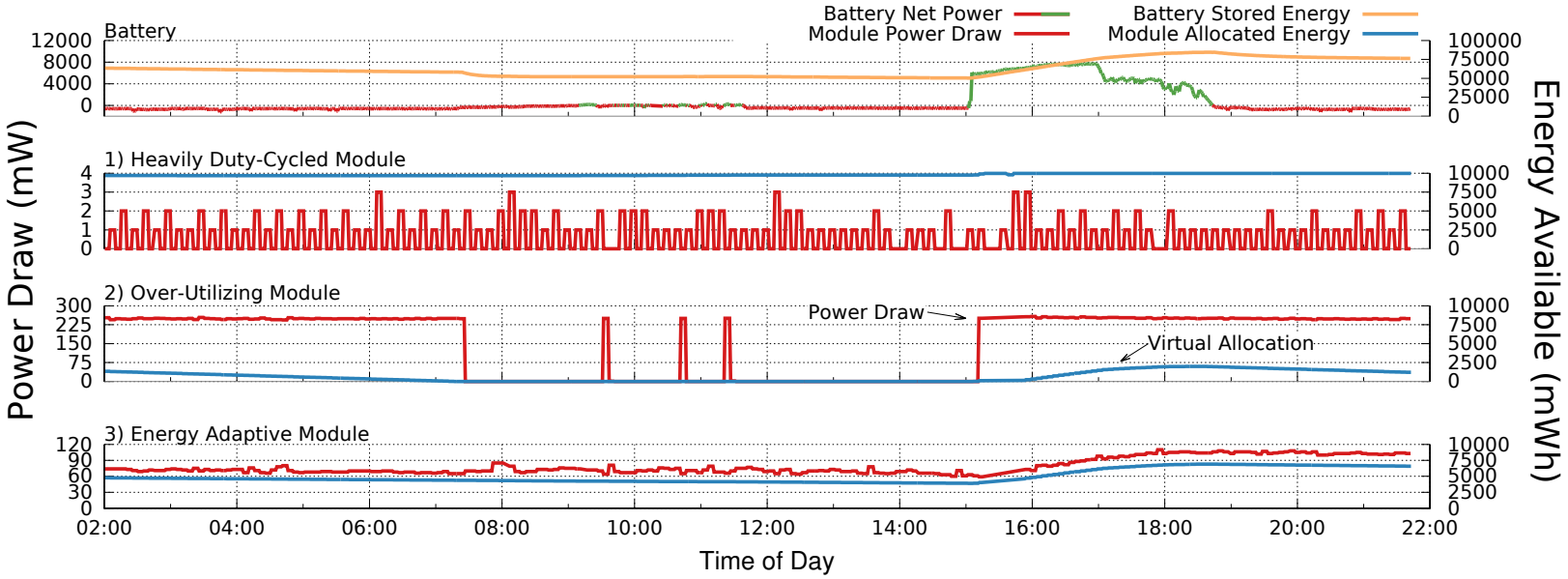


Figure 2.7: Energy isolation on Signpost. Energy allocation and five-minute average power draw are displayed for three simultaneously running applications and the platform as a whole. Each application employs a different strategy for energy use. The first is only active for a brief period every ten minutes, achieving a low average power, and storing up an allocation of energy. The second continuously runs, exhausting its budget, and is disabled by the platform, to be enabled later when energy is available again. The third adapts its actions based on the available energy, running continuously without depleting its allocation. Signpost is capable of balancing the needs of these three applications simultaneously, assigning each a "virtual allocation" of energy it draws from without affecting the operation of the others.

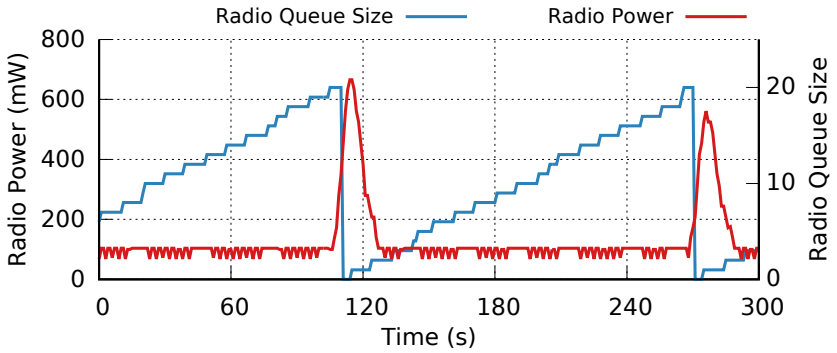


Figure 2.8: Communication policy in practice. The power draw of the Radio Module is shown along with the number of messages queued to be sent. The communication policy is set to automatically transfer data over a cellular connection if the queue reaches twenty messages, as can be seen by the increased power draw. This policy allows the platform to adapt to both increased application requests and poor network conditions by utilizing high-power resources.

Internal Communication

The Signpost design includes a single, shared, multi-master I^2C network for internal communication, such as requests to platform services. When multiple applications are running simultaneously, this bus can be a source of contention. While the Signpost design expects only a modest utilization of the shared I^2C bus, in practice sensing events can often be correlated and traffic can be bursty. Theoretically the listen-before-talk requirement of I^2C should make the bus achieve nearly 100% reception rates even in these scenarios, however we observe that this feature is not implemented in all TWI/I^2C peripherals. Assuming no carrier sense capability, the I^2C bus resembles the original unslotted ALOHAnet [58], and the target utilization rate should be kept to the 20% proposed by ALOHA. This corresponds to a total traffic of 80 kbps on a 400 kHz I^2C bus, which we believe is sufficient for most sensing applications. Applications that require higher throughput can make use of the optional USB bus.

Microbenchmarks

Several services are important to benchmark due to their impact on the range and performance of Signpost applications.

Communication Policy

Signpost provides multiple wireless interfaces. These have an advantage in supporting various communication policies that determine how data should be be transmitted based on quality of service needs and the current energy state of the platform. One simple policy is to primarily use the lower power LoRaWAN radio for data transmission unless the message queue gets too full, which could occur when applications have large amounts of data to transfer or in

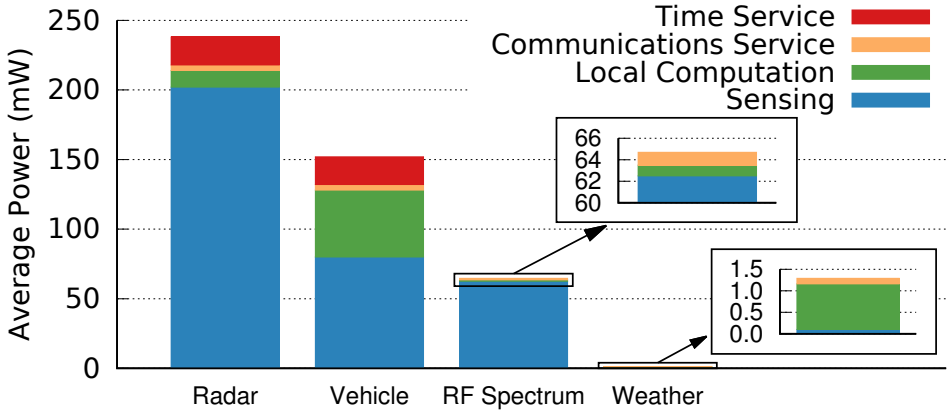


Figure 2.9: Resource usage of example applications. We break apart the major components of usage for example applications into sensing cost, local computation, and network and time service requests. Heavily duty-cycled applications such as the weather monitoring app have nearly inconsequential average power. Applications performing constant sensing with tight timing requirements both draw a higher total power and remit a greater share platform power draw. Applications like spectrum sensing can achieve moderate average power draw even with high instantaneous sensing power using duty cycling. Dynamically adjusting duty cycling allows spectrum sensing to adapt to energy availability.

poor radio conditions when LoRaWAN bandwidth is limited. When the queue gets too full, the cellular radio is activated and all queued messages are transferred quickly. In Figure 2.8, we demonstrate an example of this policy. Poor communication conditions are emulated by removing the LoRaWAN radio antenna, causing messages to be queued until the cellular radio is activated to dispatch them, resulting in briefly increased power draw.

Synchronization

Some applications require coordination between multiple modules on a single Signpost or between multiple Signposts, requiring tight synchronization [48]. On Signpost, a PPS signal is routed to each of the sensor modules from the GPS to provide this synchronization. We find the timing difference across Signposts to be 75 ns in the average case with a 95th percentile metric of 97 ns. We observe little skew in the signal from Control Module to sensor modules (less than 6 ns) and almost no variation from module to module. We expect this synchronization precision to suffice for many applications, providing sufficient resolution for RF localization on the order of tens of meters and sub-meter audio localization.

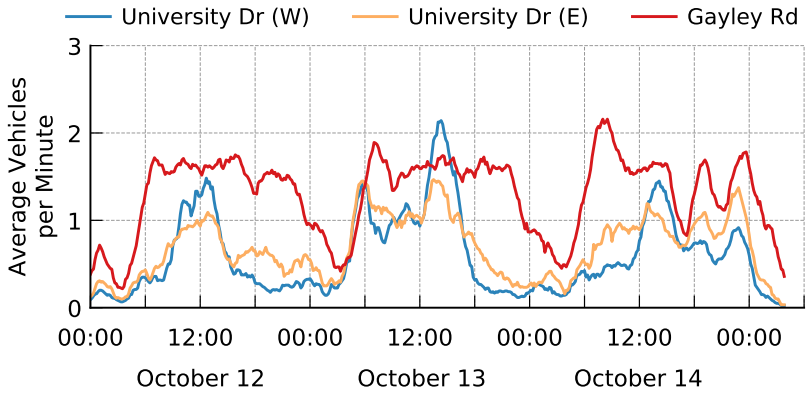


Figure 2.10: Vehicle counting application. Several days of processed audio data are collected in October 2017 for the vehicle counting application. Prominent peaks across several audio frequency bands are used to detect vehicles. We plot estimated vehicles per minute averaged over a one hour time window. The Signposts on University Drive are close, but do not have completely redundant traffic paths. We note that Gayley Road sees traffic much later into the night because it is a through street that routes around campus. Interestingly, all the Signposts experience traffic until around midnight on October 14th, and after further examination, this was due to a concert at a nearby venue. Clear peaks in traffic can be seen before and after the concert, which started at 20:00.

2.6 Example Applications

Applications run on sensor modules and have access to system resources through physical connections and software APIs. We design several applications (and sensor modules) and deploy them on the Berkeley campus for several months. While applications written by users will be different, these examples can inform the types of applications that are possible on Signpost. We describe our applications, the platform resources they use, and some example results. Figure 2.9 shows the power drawn by different components of the applications, broken down into draw by sensors, local processors, and the communications and time services.

Weather Monitoring

The weather monitoring application uses the environmental sensing module to sample temperature, pressure, and humidity every ten minutes, sending it to the cloud via the Signpost network API. After the data reaches the cloud, it is posted to Weather Underground to help support their goal of distributed weather sensing. The application achieves very low power operation even without implementing sleep mode by using the energy API to power off the sensor module between samples.

```
uint8_t send_buf[DATA_SIZE];

void send_samples (void) {
  // Add a timestamp to the data
  time_t time = get_time();
  memcpy(send_buf, time, sizeof(time_t));

  // Send data over network, allowing Signpost to decide how
  network_send_bytes(send_buf, DATA_SIZE);
}

int main (void) {
  // Initialize the module with Signpost
  api_t* handles[] = NULL; // provides no services
  module_init(handles);

  // Collect audio data with an ADC, placing it into send_buf
  adc_continuous_sample(SAMPLE_RATE, &data_ready_callback);

  // Send samples every ten seconds
  timer_every(10000, &send_samples);
}
```

Figure 2.11: Example module software. This software snippet from the vehicular sensing application collects averaged volume data for ten seconds and transmits it using the network API. Timestamps for the collected data are requested from the time API and appended to the data before transmitting it. Access to the Signpost APIs makes applications easier to create.

Vehicle Counting

The vehicle counting application runs on the audio sensing module, which provides the volume of audio in seven frequency bins collected up to 100 times per second. This module should in principle allow high-level event recognition (e.g. vehicle detection), without capturing recognizable human speech. The application records these volumes, averages them over a second, and every transmits the results every ten seconds to the cloud using the network API. To properly identify vehicle movement, the application must know the precise time at which a volume sample is taken, so the time API is used to timestamp each batch. The code for this application is shown in Figure 2.11, and the average power draw and resource usage are shown in Figure 2.9. The requirement for precise timing information results in the application being charged for a portion of the GPS power. Additionally, the local processor must stay active to continually sample and process incoming audio volume data. Once the data are in the cloud, it is processed to look for peaks that are indicative of a moving car. An example of the output is shown in Figure 2.10.

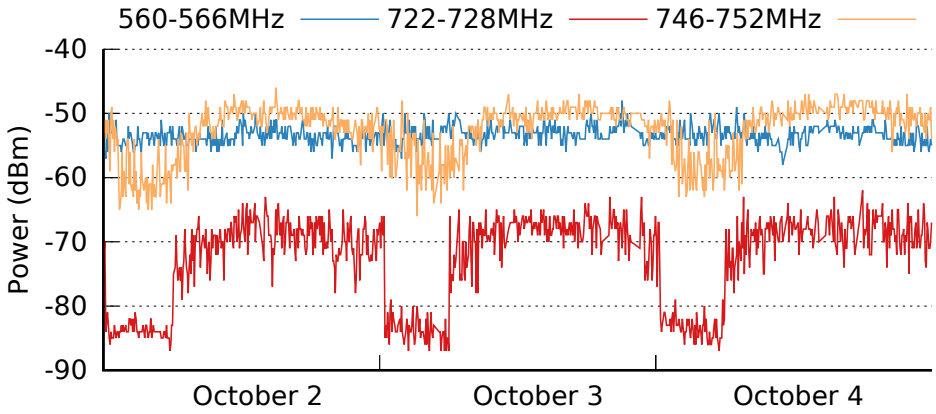


Figure 2.12: RF spectrum sensing application. A sample of RF spectrum data from October 2017 in three frequency bands corresponding to a local TV station (560 MHz), AT&T owned spectrum (722 MHz), and Verizon owned spectrum (746 MHz). Distributed and fined-grained spectrum sensing could help to build better models of RF propagation and inform policy around the reuse of underutilized spectrum. The two higher frequency bands are particularly interesting due to their cyclic nature.

RF Spectrum Sensing

The white space sensing application runs on the RF spectrum module and periodically samples the energy on each of the TV white space channels (every 6 MHz from 470-830 MHz). For thirty seconds, the spectrum analyzer reads the energy on these channels and computes the min, max, mean, and standard deviation for each channel. The application then sends this data with the Signpost network API and uses the energy API to power off. While the duration for power off is currently set to three minutes, it could be adapted to available energy without significantly degrading the utility of the application.

Three days of this data are shown for several interesting channels in Figure 2.12. While our RF spectrum module does not yet meet the FCC requirements for a white space utilization sensor, collecting distributed RF spectrum data can be used to inform RF propagation models and inform policy about the reuse of underutilized spectrum.

2.7 Lessons Learned: A Case for Dynamic Multiprogramming and Resource Management

The goal of the Signpost platform was to lower the bar to deploying distributed sensing applications within cities, and in some respects it achieved that goal. The platform made it easier to build, deploy, and test new modalities of sensors. It alleviated the designer from needing to think about the often-difficult to implement core services, and the energy and communication resources provided by a signpost are sufficient to support many kinds of sensors and many sensing applications. Furthermore, resource metering and hardware isolation work well and enable the amortization of deployment and maintenance between multiple sensors. From the outside, these properties met the goals of sensing as a shared utility in the sense that users can effectively share Signpost, which lowers the bar to deploying resource-constrained sensors.

One difficulty that the platform faced, however, was that the applications we could build with sensors using Signpost were not particularly compelling. They were mostly simple, sense-and-send applications. The level of sensing was severely constrained by the available resources. Using the platform and its hardware resources for more complex functions was extremely difficult. Even for simple applications, the extra burden of reacting to the resources available to the sensor required a type of programming to which most developers are not accustomed.

Building more complex, stand-alone applications, such as event detectors which performed local signal processing or sophisticated filtering was difficult for several reasons: (1) writing complex code in a low-level language like C is slow, (2) deploying new code iterations on a Signpost sensor was risky because a bug in an update could prevent future updates, (3) code could often only be tested *in situ* because the events being detected only occur in the field, and (4) deploying and testing new code disrupted the collection of possibly valuable data from the existing application.

Writing applications which coordinated between sensors running on multiple signposts or even two sensors on the same Signpost was even more challenging, repeating all the challenges of stand-alone applications stated above and additionally requiring the programmer to handle coordination in the face of variable and constrained resources. This essentially required each programmer to implement a bespoke distributed computing framework. We tried and ultimately failed to implement several kinds of event detection and localization applications on Signpost.

The team that designed signpost anticipated that programming the sensors, and specifically running distributed applications across multiple sensors would be a key challenge in a deployment of Signposts. Many of the early design concepts focused on this issue rather than the hardware platform itself. In retrospect, we did not realize at the time that we were trying to solve a challenging problem that spanned several domains. Conversations jumped from around from the language and programming models we would use to describe a distributed application, to the way each individual sensor would participate in the distributed

application, to how multiple sensors producing different data streams could make their data available to other programmers and applications, all while contending with the fact that an operating system which provided multiprogramming, resource isolation, or even a snapshot of what resources were available did not yet exist for this class of sensor or the dynamism we anticipated.

Perhaps that is why we built signpost the way we did—to provide sensor builders with the appearance of an operating system through a standard hardware interface at a time when making a software operating system was too difficult due to heterogeneity and lack of hardware isolation techniques in the embedded processor space. It's clear now that unless one decided to array a Signpost with five identical sensors, this platform-level resource metering and isolation is foundational to, but did not help us achieve Signpost's goal of making *applications* easier to develop, and, more fundamentally, did not go far enough in enabling the idea of sensing as a shared utility. Achieving this loftier goal requires two major additional steps.

The first is the ability to perform resource isolation and dynamically multiprogram an individual sensor module. While at the time we knew this would be possible, and it is even a architecture we call out in Figure 2.2, in practice, the Tock embedded operating system, was not quite ready to support multiple applications at the time of Signpost's implementation. With multiprogramming at the sensor module-level, code iteration is de-risked because the underlying multiprogramming abstraction can necessarily isolate misbehaving code, production and development code and run simultaneously, and, if the goal is to enable a shared sensing fabric, it can allow multiple users to use the sensors on a Signpost in an application without necessarily needing to know how to build and deploy a sensor module.

The second requirement, alluded to above, is the need for a high-level programming abstraction, especially for distributed applications. It should also be noted that nearly every embedded sensing application is necessarily distributed at least between the sensor collecting data and a cloud application that receives and processes that data, even if multiple sensors are not involved. Today we rarely program other distributed systems by individually programming each node in the system. Instead, a unified programming model is provided, and a programming framework either statically or dynamically distributes code fragments throughout the system and connects their inputs and outputs. This is true in distributed data processing systems, and web frameworks, and these frameworks were created because the alternative—thinking about and manually distributing the code to each element of the system—is often untenable. In the case of a Signpost, or any resource-constrained sensing system, the distribution requires knowledge about available resources to be centralized so that a programming framework can make code placement decisions without exceeding those resources. This is the topic we turn to next.

Chapter 3

Advantages of Dynamic Local Processing

We would not need to write complex or dynamic applications for a platform like Signpost if all sensor data could be streamed directly from the sensor to a more capable computer in the cloud, and many existing sensing architectures and researchers argue for the simplicity of deploying sensors which act as single-purpose data forwarders. In this chapter we push back against the idea of sensors as statically-configured data forwarders, and motivate the need for dynamic applications to be deployed to sensors for local data processing. We show that it is not only currently more efficient to perform local processing on the sensor for most applications, but that it is becoming increasingly more efficient over time. In addition to efficiency, properties such as reliability and privacy also motivate the need for dynamic, on-sensor compute as opposed to entirely cloud-based processing.

More philosophically, the ability to easily perform local computation at the source of the data gives us access significant data that can be sensed but not communicated back to the cloud due to resource constraints. Sensors which are confined to acting as single-purpose data forwarders could not extract value from this data, and such an architecture assumes that our knowledge of a sensor's data streams is complete at the time of deployment. This assumption of knowledge is often contradictory to the need to deploy a sensor in the first place, and certainly contradictory to using resource-constrained sensors for novel science. Dynamic applications are therefore necessary to realize the full value of a sensor deployment.

3.1 Efficiency

Placing sensors near the source of the data often puts the sensors themselves under resource constraints. Because they are placed in the physical world, wires for communication and power are costly and difficult to provision, and their size is constrained for practical and aesthetic reasons, limiting the size of a sensor's battery or ability to harvest energy from its environment. The presence of these resource constraints often requires the sensor designer to carefully optimize the use of energy on a sensor, including the decision of when to send data to a computer with more resources and when to process and filter data locally.

Traditionally, the amount of energy used to wirelessly communicate dominated a sensor's energy budget [59, 60]. In recent years, however, wireless radios and protocols have improved, lowering the amount of energy required to send data. At the same time, the amount of energy used by the processors to perform computation has also fallen significantly, and processors have added features like floating point units (FPUs) [61] and special-purpose machine learning accelerators [62, 63] to lower the amount of energy and increase the processing speed of specific applications.

To determine the whether it is more efficient to perform computation locally or send data to the cloud, we examine the energy efficiency of a range of wireless protocols then compare the amount of energy to send data through those protocols to the amount of energy it would take to process that data for a variety of applications on low-power processors. Additionally we examine this trade-off over the last two decades to better understand whether future energy efficiency trends will favor more or less local processing. We also discuss new technologies such as backscatter communications that have the potential to dramatically change this trade-off between transmission and computation.

Energy of wireless communication technologies

To perform an energy trade-off analysis, we need a catalog of the amount of energy required to communicate using the wireless technologies commonly found on sensor nodes. An overview is shown in Table 3.1. These technologies span from lower power and lower-range protocols such as Bluetooth Low Energy (BLE) and WiFi, to long range and managed cellular networks like LoRa and LTE, to newer IoT focused cellular protocols such as LTE-M and NB-IoT.

We see that the amount of energy required to transmit data spans four orders of magnitude, largely due to the range of communication achieved which itself spans from tens of meters to many kilometers. Even within a protocol, the energy required to transmit usable data can span several orders of magnitude due to differences in transmit power between implementations, the amount of energy required to startup the radio, and the overhead of protocol tasks such as scheduling. Some protocols and radios have the ability to scale back the amount of energy used based on needed range and network congestion, but often these options are discrete or themselves have energy overhead. In Section 3.1 we use the best available energy efficiency to compare the amount of energy required to send and process data using several select algorithms.

Due to the wide range in energy per bit, to evaluate how the energy efficiency of wireless communication is changing in time, and to subsequently compare that to the energy efficiency of low-power processors, we must create a metric that standardizes communication energy efficiency across protocols and implementations. To do this we use energy per bit per meter. While this combined metric is an approximation and favors shorter range communication, all options fall well short of the theoretical limits of wireless communication efficiency and most technologies being compared have similar transmit ranges [64]. Wireless radios from the recent past are compared using this metric in Figure 3.1.

Technology	Energy/bit (nJ)	Throughput
Bluetooth Low Energy [65–67]	10 - 30	1 Mbps
802.15.4 [66, 68]	63 -85	250 kbps
WiFi [69, 70]	3 - 1,000	72+ Mbps[a]
LoRa [71, 72]	3,500 - 361,000	980 - 21,900 bps
2G [73]	8,000 - 30,000[b]	239 kbps[c]
3G [73, 74]	431 - 10,000[b]	2 Mbps[c]
LTE [74, 75]	500 - 10,000[b]	100 Mbps[c]
LTE-M [76, 77]	922 - 2,322[d]	300 - 375 kbps
NB-IoT [76, 77]	2,700 - 6,900	30 - 169 kbps

[a] The WiFi protocol and available radios are able to transmit much faster than 72 Mbps, but this analysis only includes radios focused on low-power and embedded WiFi.
[b] Broad ranges in energy per bit are due to significant transmission start-up cost. The lowest energy numbers are achieved during large transmission at the peak data rate. For a single transmission energy per bit could be as high as 500,000 nJ. [c] Max throughput is listed, however throughput may be lowered when transmission is longer range or during periods of higher congestion.

Table 3.1: An overview of the energy and throughput of various wireless communication technologies. This provides a sense of the amount of energy a sensor using these technologies may use to transmit data and the associated latency of transmitting that data. Note that these technologies are not easily comparable. They operate at vastly different communication ranges and networking topologies. Many of the technologies have the ability to scale data rate up or down depending on the available link budget. For cellular technologies the energy associated with starting a transmission and scheduling is significant and not easily included in a single energy per bit metric. The presented numbers attempt to present the lowest reasonable energy metric for each technology so that conclusions drawn about the amount of computation that is optimal to perform locally are valid even for conditions that are most favorable towards offloading data.

We see that the energy efficiency of wireless radios is improving exponentially in time and that commercial technologies lag those presented in research by 4-6 years. We also see that the trend of continuing efficiency improvements in active radios has not continued in recent years, and that even across protocols, radios achieve similar peak efficiencies. This makes sense as active radios are all bound by the same physical limitations and the same circuits which improve efficiency for one protocol can be used in other protocols.

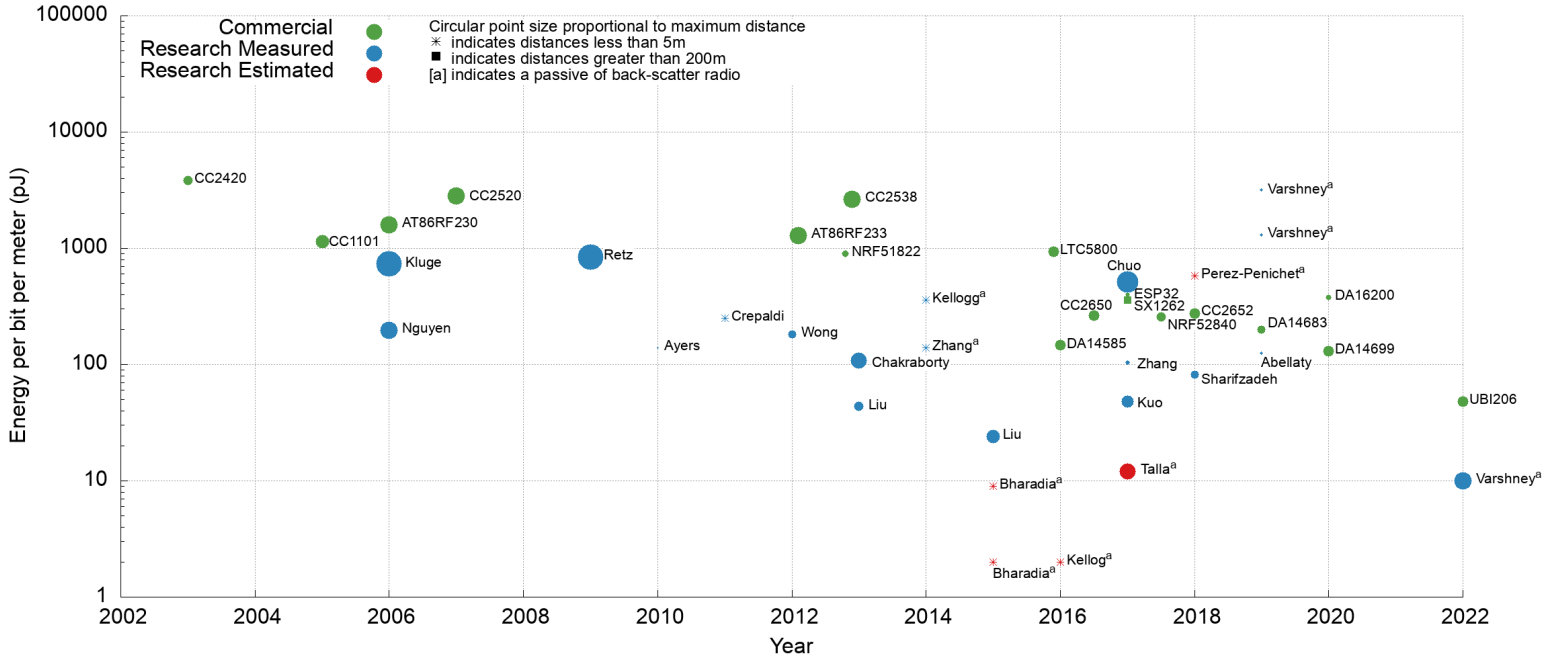


Figure 3.1: The energy per bit per meter of low-power wireless radio technologies, including commercial technologies, realized research technologies, and simulated or estimated research technologies. Distance is presented at the most optimal efficiency and is calculated using published link budgets and the Hata model [78] with a 20 dbm fading margin or from reported measurements. The use of the Hata model is an attempt to compare technologies across different deployment scenarios and frequency ranges, but could lead to error in the calculated metric for indoor deployments. We see an exponential drop in the amount of energy required to perform wireless communication over time and clear 4-6 year lag between efficiencies in the research domain and those available commercially. While energy efficiency continues to improve, especially commercially, the improvements of efficiency in the research domain have slowed in recent years. With the exception of Judo [79], no realized passive technologies are more efficient than realized active radio technologies [80, 81]. From analyzing these trends there is no indication of an upcoming dramatic shift in wireless communication efficiency that would change the energy optimal trade-off between the processing and sending of data. The full dataset can be found in Appendix A

One possibility for a discontinuity this trend is passive or backscatter radios which do not use their energy to transmit a wireless signal, but instead rely on the energy of the powered base station or access point and encode data in the signal reflected to the base station or reader. Back-scatter radios in Figure 3.1 include Bharadia et al [82], Kellog et al [83, 84], Talla et al [85], Zhang et al [80, 86], Varshney et al [79, 87], and Pérez-Penichet et al [81]. While this does not lower the energy of receiving data, which still must be extracted in the same way active radios, it has the potential to significantly improve the efficiency of offloading data from a low-power sensor to the cloud.

In practice, however, because a signal is reflected and not transmitted, it experiences double the path loss, and the effective range of a passive radio is much lower than an active radio using the same bandwidth, coding, and protocol. Additionally, with the exception of Varshney et al. [79] no constructed and measured passive radio improves on the combined efficiency metric of active research or even commercially available radios. It is unclear power consumption results presented in simulation from Bharadia et al [82], Kellog et al [83], and Talla et al. [85] would be as efficient if produced and measured. Given this, we do not expect passive radios to dramatically improve the widely available energy efficiency of wireless communication in the near future, but we may see ideas from passive radios like those in Varshney et al. [79] help continue to improve efficiency at the current rate.

Trade-off between sending and processing

While the energy efficiency of wireless radios is improving, the energy efficiency of processors is improving as well. More fined-grained power gating, lower power volatile and non-volatile memory, and subthreshold technology has significantly lowered the active power of commercially available low-power processors in recent years [88, 89]. To compare the rate of wireless radio efficiency improvements to that of low-power processors we calculate the number of processor cycles that could be performed per bit of information transmitted a 10 m distance as a function of time, taking the most efficient commercial processor and radio available in a given year, as shown in Figure 3.2.

We see that over time, and especially in the last 5 years, processor efficiency improvements have outpaced radio efficiency improvements. This seems particularly tied to the release of subthreshold technologies commercially [41, 88, 89], but, as can be seen in Appendix A, other manufacturers who do not rely on subthreshold technology are also significantly improving low-power processor efficiency. Additionally, this does not consider the hardware accelerators or improved ALUs which can perform more complex instructions in fewer cycles that are now included in low power processors [61, 62].

The relative rates of improvement, however, do not provide a sense of whether it is advantageous to send data or process that data locally for specific applications. This depends not only on the relative energy efficiencies of the wireless radio and processor, but also the algorithmic efficiency for the specific processing that needs to be applied to the data. An algorithm that requires more cycles for each unit input data will be relatively less efficient to execute locally. The trade-off will also depend on the data compression rate of an algorithm.

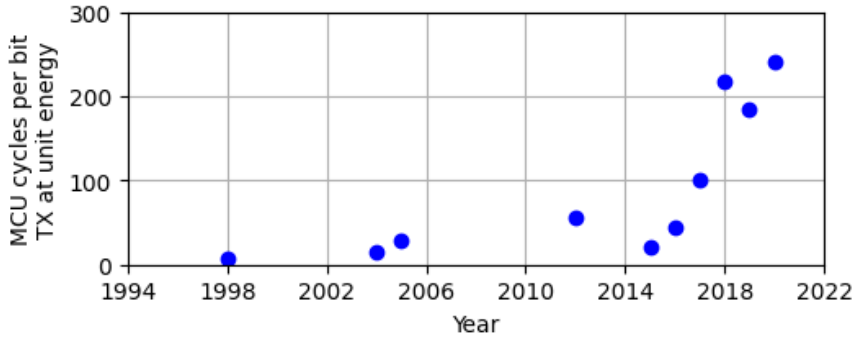


Figure 3.2: The number of MCU cycles that can be performed per bit of data transmitted a 10 m distance at a unit energy over time. A point is plotted in every year that a more efficient processor or radio technology was released commercially. Energy per bit required for a 10 m transmission is calculated using the energy per bit per meter metric presented in Figure 3.1, and processor energy per cycle metrics are from published datasheets. We see that energy efficiency improvements in low-power processors have recently significantly outpaced energy improvements in wireless communication. This makes more local computation more energy efficiency if it filters or ultimately reduces the amount of data that must be transmitted. Complete data used to generate this figure can be found in Appendix A.

For processing that is used to perform event detection, this is directly proportional to the fraction of events that are detected, while compressing data like an image only lowers the amount of data sent by the compression ratio.

To understand this trade-off for real algorithms, we find performance metrics for commonly used signal processing techniques, JPEG compression algorithms, and newer machine learning inference models, and use those to ground the comparison between executing those algorithms locally or sending the input data to the cloud. The results are shown in Table 3.2. Although we cannot exhaustively catalog the algorithms used for every application, nor can we anticipate the application trade-offs of sending data to the cloud compared to filtering it locally, the analysis is still revealing. For instance, micro machine learning models have lower accuracy on low-power processors than larger models that can be run on more resourceful machines [90, 91], and the importance of this accuracy difference will depend on the specific use-case. Still, we hope these examples can inform the magnitude of this trade-off given current radio and processor energy efficiencies.

			Wireless Technology									
Algorithm	**Cycles**	**Data Size**	**BLE**	**802.15.4**	**WiFi**	**LoRa**	**2G**	**3G**	**LTE**	**LTE-M**	**NB-IoT**	**Bharadia [82][d]**
Filter	4	4 B	1.2K	7.6K	363	500K	1M	52K	60K	111K	327K	1.7
RFFT 32x1024 [92]	88 K	4 kB	56	354	17	19.7K	45K	2.4K	2.8K	5.2K	15K	0.08
CFFT 32x1024 [92]	139 K	4 kB	35	223	11	12.4K	28K	1.5K	1.8K	3.3K	9.6K	0.05
FIR Filter 80 Tap [92]	257	80 B	1.5K	9.5K	453	528K	1.2M	65K	75K	139K	407K	2.11
RMS Voltage[a]	12 K	8 kB	800	5K	242	282K	646K	35K	40K	74.4K	218K	1.13
JPEG Compression [93][c]	18.3 M	147 kB	9.9	63	3.0	3.5K	7.9K	428	496	916	2.7K	0.01
VWW+MobileNetV2 [90][c]	18.4 M	48 kB	3.2	20	0.97	1.1K	2.6K	139	161	299	874	0.005
VWW+MobileNetV2 [90][c]	53 M	147 kB	3.4	22	1.03	1.2K	2.7K	148	172	317	929	0.005
VWW+TFLite [90, 91][c]	4.8 M	48 kB	12.3	77	3.68	4.3K	9.8K	529	613	1.1K	3.3K	0.02
DNN HotWord [94][b]	3.3 M	48 kB	17.5	110	5.26	6.1K	14K	755	876	1.6K	4.7K	0.02
CNN HotWord [94][b]	78 M	48 kB	0.7	4.6	0.22	258	591	32	37	68	199	0.001

[a] Assumes a 4 K, 16 bit multiply-accumulates that take 3 cycles each, two loads, and single cycle multiply accumulate instruction.
[b] Assumes the running 50, 16x1024 FFTs on one second of 24 kHz, 16 bit audio to create feature vectors as input to DNN and CNN in line with [94].
[c] Actual processor power may be higher by 2-3x for high memory applications. Benchmarks are usually performed with a subset of memory banks active.
[d] Uses 0.015 nJ/bit presented in Appendix A. Range limited to 7 m.

Table 3.2: The energy ratio of transmitting the input data over the specified wireless technology and performing the task locally. Numbers greater than 1 are more efficient to perform locally. The processor is assumed to an ARM Cortex M4 with an FPU running at 20 µA/MHz and 3.3 V, which is efficient, but not state-of-the-art among modern MCUs such as those listed in Appendix A. Energy to transmit the data uses the lowest energy per bit presented in Table 3.1. This is favorable to transmitting the data as it is often only achieved in during large batch transmissions and doesn't fully account for scheduling overhead [74]. We see that even for the most computationally intensive published algorithms, such as neutral networks designed to classify images and detect audio wake words on resource-constrained processors, it is still more energy efficient to perform the task locally on all active radios except for the most intensive algorithms using the most energy efficient radio technologies. The results for Bharadia et al. [82] are in simulation only and have a range limited to 7 m, but represent the potential for future passive radio technologies to shift this trade-off if simulations results are upheld in practice.

We see that for nearly all combinations of algorithms and wireless technologies it is many orders of magnitude more efficient to perform the processing locally compared to offloading the data and performing the same processing in the cloud. The only exceptions are the algorithms which take the most cycles for a given amount of input data paired with the lowest power wireless communication technologies, and for the simulated passive WiFi radio presented by Bharadia et al. [82]. The clear and lasting efficiency advantages of local execution supports the need for a system that can enable easier and more dynamic applications for resource-constrained embedded systems.

Latency and Cost Advantages

While sensor designers may be primarily energy constrained, the decision of whether to transmit or process data also impacts latency and sometimes the cost of a system. The optimal decision for both of these considerations is application-specific, but considering common scenarios makes it clear that local processing is often advantageous.

Latency

Latency can be considered similarly to energy with the efficiency improvement directly resulting from the ratio between communication throughput and the speed of the processor on the sensor. Common embedded processors run from 64-192 MHz (see Appendix A. From table Table 3.1 we can see that common low-power wireless protocols such as BLE and 802.15.4 range from 250 kbps to 2 Mbps, higher throughput protocols such as WiFi and cellular range from 1 Mbps to 72 Mbps, and long-range, low-power protocols range from 1 kbps to 350 kbps. Considering a 128 MHz processor, the processor may be able to only perform 1-2 cycles for every bit of information transmitted by the fastest protocols, but can perform 120 cycles per bit for common protocols like BLE and up to 120,000 for the slowest protocols. The similarity between cycles per bit transmitted at a unit energy and cycles per bit transmitted in a unit of time means the efficiency of applications presented in Table 3.2 will be similar to the efficiency trade-off in the time domain, with many applications performing at lower latency if the processing is performed locally and only the result is transmitted.

Cost

Cost will only be a factor if the wireless communication technology being used charges per unit data transmitted. This is most often the case with cellular technologies, but other schemes have been proposed such as paying for mobile phone-based backhaul, and user-deployed networks such as Helium [95–97]. Cellular pricing varies significantly with user-centric plans providing data for less than 1 cent USD per MB [98] and plans designed for global sensor networks charging as much as $0.70 USD per MB [99].

In table Table 3.3, we consider several applications including person counting, audio keyword detection, and power grid monitoring, and show the cost of transmitting the full data

			Monthly	
Application	**Type**	**Data**	**Cost ($0.01/MB)**	**Cost ($0.25/MB)**
Person counting[a]	continuous	650 MB	$6.50	$162.50
Audio keyword[b]	continuous	82 GB	$820	$20.5 K
Grid monitoring[c]	continuous	21 GB	$210	$5.3 K
Person counting[a]	triggered[d]	108 MB	$1.08	$27.09
Audio keyword[b]	triggered[e]	1.7 GB	$17	$425
Grid monitoring[c]	triggered[f]	4.8 MB	$0.05	$1.2
Person counting[a]	local[g]	276 kB	$ 0.01	$0.25
Audio keyword[b]	local[h]	640 B	$0.01	$0.25
Grid monitoring[c]	local[i]	640 B	$0.01	$0.25

[a] Assumes 224x224x3 B images sent every 10 minutes. [b] Assumes a 256 kbps audio stream. [c] Assumes a voltage wave sampled at 4 kHz. [d] Triggered for 4 hours per day. [e] Triggered for 30 minutes per day. [f] 2 second slice triggered 10 times per day. [g] One 64 B packet with counts sent per 10 minutes. [h] Ten 64 B packet with keyword sent per day. [i] Ten 64 B identified anomalies sent per day.

Table 3.3: The monthly cost of sending sensor data over the cellular network with varying degrees of local processing. Often streaming continuous data is untenable, but anomaly detection or local summarization can reduce data usage and the subsequent cellular cost to a reasonable amount. Cellular costs are based on common costs for both consumer and IoT cellular plans [98, 99]

stream compared to sending summarized or filtered data at two data costs. Local filtering and summarization is often critical to keeping cost in check when using cellular backhaul, especially when many sensors are deployed. As shown in Table 3.3, deploying applications without at least some local filtering on cellular networks is often untenable, and the need to tailor and modify the local filtering post-deployment calls for a system to facilitate the deployment of these applications.

3.2 Opportunity

The increase in efficiency and decrease in cost afforded by local data processing would give applications access to much more data than can currently be collected and analyzed in the cloud. Much of this data would be traditional sensors sampled at higher rates than we use them for in traditional applications. We see that many applications are enabled by sampling these sensors at higher rates and fingerprinting and finding patterns in their transient responses to events or in side-effects to other nearby physical phenomena.

Laput et al. shows that even a single sensor node with access to much higher sample rate data from a host of sensors can enable a wide range of classification tasks throughout a room [100]. They note that the three highest sample rate sensors—accelerometer, microphone,

and EMI—are also the most relied upon for classification. While Laput et al. uses a single, wall-powered sensor for this task, increased sensor density enabled by local processing would very likely improve classification accuracy.

Several papers including Gambiroža et al. and Zhang et al. show that more intensive processing on the transient responses of gas concentration sensors can improve accuracy, reduce measurement time, and make the sensors more resilient to confounding factors such as humidity compared to taking the steady-state response without processing [101, 102]. Burgués et al. show that the transient response of gas sensors can be used to estimate gas source distance even without wind measurements [103]. Local processing could enable even the more intensive processing chains such as the small LSTM model proposed by Gambiroža et all to be run on resource-constrained sensors.

Power grid sensors such as μPMUs have the ability to detect many types of grid faults, and, despite being powered, require significant cloud infrastructure and network costs to collect and store their data [104, 105]. Local processing has the potentially to significantly reduce the cost and increase the density of such sensor deployments leading to a better understanding of the power grid.

We know about the value of the data powering these applications because researchers have explicitly collected high frequency data about them, but this is not always possible. Applications in which phenomena occur in hard to reach places, or are distributed among many sensing points fundamentally can only be sensed by resource-constrained sensors. We often do not and cannot know the value of the data coming from these places a priori. As exciting as more dynamic and easier to deploy local data processing is for enabling the broader and more dense deployment of the applications listed above, it is arguably even more exciting for the applications yet to be discovered.

Unsupervised learning techniques running locally on sensors could tag abnormal events without transmitting the full data streams that are sensed. Federated learning could be optimized to minimize the energy of transmitting weights across the network link rather than to reduce training time or latency, allowing for the learning of not just a local but global view of abnormality or event classification [106, 107]. All of these enabled applications rely upon easing the burden to write, deploy, and iterate the software running locally on distributed sensor networks.

3.3 Reliability

Local computation on resource constrained devices, and application frameworks which support the scheduling and placement of distributed applications within a local network, lessens the network span and the subsequent likelihood of failure due to the network. This is especially important in deployment scenarios with sparse or intermittent network connectivity, but networking reliability even in common home deployment scenarios is worse than one might expect.

Bischof et al. show that according FCC datasets many providers only achieve 99% uptime, or 864 s of downtime per day, of network reliability with a 10% packet loss threshold [108]. This can be especially frustrating for local sensor-control loops where IoT device actuation is hampered by poor connectivity. Additionally, most power outages correspond to internet outages in areas with cable broadband. A study of smart home users found that 34% reported frustration with a smart home device due to a power or network outage [109]. These infrastructure outages do not include the potential outages of the cloud services on which resource-constrained devices that service as data forward may rely. Indeed many users noted that they experienced smart home device down time during a lengthy AWS outage [110]. Even in non-control scenarios, network down-time causes issues in data collection if local processing is not possible. Embedded devices often have limited storage, so if events are not processed and summarized locally, data may not be transmitted after the network is restored. A system which can deploy parts of an application locally in a reliability-conscious manner could eliminate these drawbakcs.

Beyond just network reliability, the ability to dynamically schedule devices to perform different tasks through a overarching programming framework could be used to more broadly support application reliability. Just as cloud processing frameworks like Apache spark reschedule lagging or failed tasks [111], sensing frameworks could reschedule sensing tasks from sensors that have run out of energy or processing tasks that were deployed to local gateways. This dynamic and re-taskable view of the sensor network could not be achieved by static data forwards.

3.4 Privacy

Even if distributed sensors could collect and stream raw data back from the cloud its unclear whether they should. Privacy is one of the top concerns of users in smart home systems [109, 112] with particular worry being focused on sensors such as microphones and cameras [113]. The ability to locally process or otherwise decimate data from these rich but invasive sensors may enable applications which would otherwise be untenable due to privacy concerns. Sensors could even be built with hardware limitations to constrain the amount of data that could be exported wirelessly.

If sensors do not process data locally, applications could still be deployed to process data within a user's home network or administrative domain without needing to offload data to cloud. This would be enabled by a distributed programming framework which includes resource-constrained sensors and other local computers. More generally, having the ability to dynamically decide where code is executed can enable us to choose where data is processed based on privacy concerns. This offers more granular control than a binary decision about whether data is too sensitive to be sent to or stored by a third-party server.

Chapter 4

The Landscape of Utility Sensing

Closely related influential work for enabling utility sensing for resource-constrained sensor networks draws on several threads of research. The distributed sensing community has long realized the difficulty of deploying and programming sensor networks. Some solutions attempt to ease programming with frameworks for programming sensors as individual nodes or as a collective (referred to as macroprogramming). Other solutions aim to amortize the cost of deployment and maintenance by enabling multiple parties to share a resource-constrained sensor or by enabling a single user to deploy multiple, isolated tasks. Both of these approaches recognized the performance, reliability, and privacy advantages of pushing more compute onto the sensor, but they have not been combined together, with frameworks for sensor network macroprogramming targetting a specific problem or application domain and sensor multiprogramming focusing on easing the challenge of writing programs for individual sensors rather than for an entire network.

At the other extreme, and largely simultaneously, similar problems as those being experienced by the distributed sensing community were being tackled by cloud computing researcher. In this setting, large numbers of commodity machines needed to coordinate to perform a range of computing tasks, and these machines needed to be shared among many developers using them for distinct purposes to amortize the capital and operational costs of the machines. Computing frameworks were developed that were easier to use than the most comparable high performance computing frameworks that came before them [111, 114]. New isolation techniques allowed for finer-grained sharing of resources such as memory, compute, and networking [115]. Of course this community had the advantage of a standardized and capable operating system on which to build, and they didn't need deal with platform heterogeneity, unique scheduling difficulties, or resource constraints common in distributed sensor networks.

Name	Key Idea	Resource-constrained	Multi-programming	Macro-programming	General-purpose
TinyOS [123]	An embedded OS with limited dynamic value dissemination capability.	✓			✓
Maté [118]	A sensor network-specific virtual machine language and runtime.	✓	✓		
Impala [124]	A sensor network-specific application framework and runtime for running multiple, dynamically loadable programs on a sensor node.	✓	✓		
SOS [125][a]	An embedded operating with dynamically loaded, cooperatively scheduled modules.	✓	-		✓
SensorWare [126]	A sensor network-specific language and extendable, multi-programmable runtime.	✓	✓		
Darjeeling [119]	A Java virtual machine for microcontrollers	✓	✓		✓
Tock [53]	Embedded OS that enables multiprogramming and a true separation between user space and kernel space with hardware memory protection.	✓	✓		✓
WASM [127]	Web Assembly. A language and VM bytecode format with compile-time bounded memory access.		✓		✓
WAMR [117]	A WASM runtime targetting microcontrollers with limited memory.	✓	✓		✓
eWASM [128]	A WASM runtime and paired ahead-of-time compiler enabling better fault isolation for resource-constrained devices.	✓	✓		✓
Micropython [129]	A Python runtime for microcontrollers.	✓		✓	

Name	Key Idea	Resource-constrained	Multi-programming	Macro-programming	General-purpose
DFuse [130]	A programming framework and API for dataflow programming and an algorithm for scheduling these tasks based on energy availability.	✓		✓	
Kairos [131]	A macroprogramming framework and complementary sensor node runtime for sensor network-specific tasks.	✓		✓	
TinyDB [121]	Query Sensors as a distributed database with extended SQL.	✓	✓	✓	
Regiment [120]	A macroprogramming framework and language for processing spatio-temporal sensor data streams.	✓		✓	
Ravel [122]	A programming framework and language for compiling and distributing a high-level application between cloud, gateway, and sensor components.	✓		✓	
YARN [132][b]	A resource manager and scheduler for cloud computing frameworks like Hadoop's MapReduce		✓	✓	
Kubernetes [133][b]	A container orchestration framework, scheduler, and associated resource manager for deploying micro-services on cloud computing clusters.		✓	✓	
Mesos [134]	A resource manager that enables multiple computing frameworks to share a computing cluster		✓	✓	✓
Hypriot [135]	Enables containerization on less-resourceful unix devices.		✓		✓

Name	Key Idea	Resource-constrained	Multi-programming	Macro-programming	General-purpose
DDFlow [136]	A Visual program language and runtime which dynamically distributes dataflow tasks among edge devices.		✓	✓	
ENORM [137]	A resource manager for scheduling computing tasks on nearby unix-based compute nodes to reduce networking latency and overall traffic.		✓		✓
ParaDrop [138]	A resource manager and container orchestration framework for wireless access points.		✓		✓
AWS Greengrass [139][c]	An IoT device manager and distributed application runtime.			✓	✓
Azure IoT Edge [140]	An IoT device manager and edge container orchestration framework.			✓	✓

[a] While technically multi-programmable, the lack of isolation between modules limits its utility.
[b] These can execute general purpose code, but tie the user to a single computing framework or orchestration interface. One could build a general framework on top of these. [c] Both Azure and Greengrass core runtimes extends into the local network but not onto resource-constrained devices, which are statically programmed data forwarders.

Table 4.1: An overview of related work on utility sensing covering select projects from research on resource-constrained networked sensors, cloud computing, and edge/fog computing. More focus is given to projects which enable the programming and management of resource-constrained devices. The number of programming frameworks, schedulers, and container orchestration frameworks targetting unix-based edge devices is too great to enumerate here; while the ideas in these papers may be helpful, they do not enable their applications to extend to resource-constrained devices [116]. We see that research targetting resource-constrained devices generally does not offer all three of multiprogramming, macroprogramming, and a general-purpose compute framework. Many are either underlying technologies which can enable general-purpose multiprogramming on a single device [53, 117–119], or macroprogramming frameworks which are limited to executed an application-specific task [120–122]. Cloud computing frameworks and resource manager enable general-purpose multiprogramming and macro-programming but rely on underlying containerization and virtualization technologies which are not available on resource-constrained embedded devices.

More recently the edge and fog computing efforts have split the difference between these two approaches. Some efforts have adapted solutions built for cloud computing to run in more distributed networks and proposed scheduling and placement solutions that are unique to edge computing. They have also proposed some solutions to the issue of heterogeneity, as their target machines cannot be assumed to have relatively standardized resources or performance. These solutions, however, largely ignore resource-constrained sensors, still requiring enough compute and memory to run a Unix-based operating system so that they can take advantage of the languages, interpreters, and isolation mechanisms that are common in the cloud. We use this chapter to summarize and describe each of these efforts, highlighting their contributions to achieving the vision resource-constrained utility sensing, and noting where they fall short.

4.1 Resource-Constrained Multiprogramming

The early days of sensor networks witnessed a push to make them more capable and more dynamic. Operating systems like TinyOS were built to abstract the intricacies of specific hardware, and new languages and node programming models were developed to make real-time and interrupt-driven code easier to write and more fault tolerant [123]. Those early operating systems, and indeed most deeply embedded operating systems today [141–143], existed as library operating systems which compile with the application into a single binary. Despite functioning much like an operating system from the point of view of the programmer, they did not provide the isolation and fault tolerance required to execute multiple, mutually-distrustful applications or even execute a core process, such as a terminal, which could restart a crashing program or receive updates. Low-power and resource-constrained processors then, and still today, lack virtual memory which is traditionally used for process isolation, making it very difficult to build a fully-fledged, Unix-like operating system [61]. Meanwhile, software-based techniques like software fault isolation (SFI) required binary rewriting, which could not be guaranteed on heavily resource-constrained hardware [144].

In response to this issue, and to solve the co-occurring problems of highly bandwidth constrained network links and energy constrained sensors, researchers developed light-weight, sensor-specific virtual machine (VM) languages and runtimes. These runtimes could receive much smaller byte code and then interpret, execute, and isolate their failures in software. Maté pushed this forward as a way to isolate failures and reduce the size of code updates [118]. Later sensor VMs such as Impala and SensorWare explicitly supported the execution of multiple programs in their runtimes [124, 126], while SOS supported the dynamic loading and execution of modules by its VM, although these modules were assumed to be cooperative [125]. All of these virtual machine languages were largely sensor-specific and were proposed both as virtual machine instruction sets and as a programming model rather than as intermediary targets for higher-level or more general languages. Their simplicity and specificity reduced the size of the VM byte code and made the runtime less resource-intensive and easier to implement, but these properties also limited the generality, portability of these systems.

To increase the generality of these solutions while keeping the benefits of a virtual machine, runtimes and interpreters for common byte code formats and high-level languages were implemented. There have been several Java Virtual Machine (JVM) implementations targetting resource-constrained processors [119, 145, 146], projects such as MicroPython to interpret Python on embedded systems [129], as well as interpreters for Lua and other languages [147]. Unfortunately, very few of these runtimes are still maintained, and the ones that are maintained use a significant portion of the processor's resources for the runtime itself. These VMs do not save energy or provide a better sensor programming model like the sensor-specific languages and runtimes, and while they may technically enable multiprogramming and fault isolation, this was never their explicit purpose, and it may not have been possible to run many programs given the runtime's own outsized resource footprint.

Since these early embedded VMs and operating systems, we've seen two major developments. The first development is the emergence of hardware memory protection units (MPUs) in microcontrollers [61]. While not as robust as virtual memory for memory isolation, MPUs can be configured to isolate multiple applications without significant software overhead. Tock uses the MPU to create an embedded OS that functions more like a traditional operating system, with a separately loadable kernel, a syscall boundary, and multiple, isolated applications, but without relying on virtual memory [53].

The second development is that the speed, memory, and efficiency of low-power microcontrollers has increased significantly over time as shown in Appendix A. In fact, low-power microcontrollers are now on par with the PDP-11 mainframes used to build the original Unix system, which had 144 kB of RAM and 1 MB of disk [148]. This increase in processing power could now enable VMs, such as resource-constrained JVMs, to run without consuming significant portions of a processor's resources. Recently, there have been efforts to make WebAssembly (WASM), an assembly language and bytecode format originally designed to execute efficient, portable, isolated program in the browser [127], run on resource-constrained microcontrollers. Several runtimes including WAMR, WASM3, and eWASM exist to execute WASM bytecode through interpretation, just-in-time compilation, or ahead-of-time compilation on resource-constrained systems [117, 128, 149]. WASM runtimes are particularly appealing because with a trusted WASM bytecode compiler, memory safety can be guaranteed with much lower software overhead than a traditional VM, or the bytecode can be compiled to the target architecture ahead-of-time for near-native performance [149].

The ability to multi-program individual sensor nodes is crucial to the development of sensing as a shared utility. It is the foundation for the execution of programs by mutually-distrustful parties, the deployment of multiple programs by the same user for testing and development, and even the reliable iteration of a single program to ensure there is a computing base to receive an iteration on failure. Multiprogramming alone, however, is insufficient as it does not abstract the programming of multiple nodes away from the user, nor does it provide a system for dividing the available resources, only isolating the resources once divisions have been made. Without a system to enable dynamic program deployment and resource sharing, multiprogramming capabilities may not be utilized, as its often unnecessary for a single user to build and run multiple programs at the time of deployment.

4.2 Macroprogramming

It is difficult to program a distributed system by individually programming each piece of the system. Instead, we create computing frameworks which provide a unified programming model that facilitates the decomposition and distribution of the individual pieces throughout the system. This is true for the web, where we have built frameworks to distribute code between the client and server, and true for big data processing where we build frameworks to distribute parallel computing tasks. The same is true of distributed sensing systems for which programs are either distributed throughout a network of sensors or partitioned across a sensor network and the cloud.

The difficulty of programming a full network of sensors by writing code for the individual nodes was quickly realized and frameworks were created to program the nodes collectively; these frameworks were called macroprogramming. Early macroprogramming frameworks like Regiment, Kairos, and DFuse enabled users to specify a query, objective, or data-flow in a high-level programming language, then parsed and distributed programs throughout the sensor network [120, 130, 131]. These systems primarily focused on the processing within the sensor network itself as opposed to processing distributed between the sensor network and servers or cloud computers with more resources. DFuse proposed energy-aware scheduling, while Regiment focused on the spatio-temporal aspects of in-network processing [120, 130]. Ravel proposed a domain-specific language which enabled a compiler to automatically decompose a program between the sensor, a local gateway, and the cloud [122]. All of these macroprogramming systems enabled a single instance of the distributed program to run within the sensor network rather than allowing for multiprogramming. While at the time of their writing these macroprogramming systems could have used one of the existing sensor-specific virtual machines, this did not occur. Understandably, the combination of macroprogramming and multiprogramming requires that at the time of program distribution, the programming framework be aware of the available resources on each node and then use that information for scheduling. Unfortunately such a capability never materialized in the multiprogramming solutions nor in the macroprogramming frameworks.

TinyDB enables multiprogramming and macroprogramming by allowing queries to be specified in an extended version of SQL and then sent, parsed and distributed in a sensor network [121]. Multiple queries could be run simultaneously. TinyDB also implements other database-like optimizations, optimizing sensor power by batching sampling events, automatically pushing local filters to the sensor node, and not sampling sensors that cannot impact the results of the query. TinyDB enables multiprogramming through a query prioritization service that combines data in a data stream together through averaging or drops data that does not impact the resulting value of the data stream. If it still cannot keep up with the number of requested queries, low priority results are dropped. TinyDB also proposed potential optimizations such as combining queries with redundant data [121].

We believe TinyDB gets closest to enabling sensing as a shared utility, but fails in one key regard—it limits the programmer to using its narrow, SQL-like programming language. All of the macroprogramming frameworks discussed have a similar flaw, prescribing their

programming model as the only way in which to program the sensor network. We note this limitation in the "general-purpose" column of Table 4.1. In practice, the requirements of a programming model are as diverse as the applications they serve, and it is unlikely one programming model or language can serve all of these needs. SQL may work very well for writing exploratory queries or returning periodic aggregations, but hinder complex signal processing or stateful filtering. The sensor-gateway-cloud architecture proposed by Ravel may work well for home automation, but does not enable on-sensor, in-network processing. We believe that this lack of generality, expressiveness and extensibility, combined with the lack of processing power and memory available at the time these frameworks were created, explain why these earlier efforts have not seen broad adoption or maintenance. Moving forward, we hope that the ideas from these earlier macroprogramming frameworks can be adapted and can coexist simultaneously for sensing as a shared utility, even on the same node or network.

4.3 Utility Computing in the Cloud

The modern cloud is the best representation we have of utility computing. At a datacenter-scale, machines are run and partitioned among many customers, and the cost of running the machines is amortized across these customers. Individual customers can program clusters of these machines using programming frameworks and container orchestration systems which handle the distribution and scheduling of tasks as well as the monitoring and restarting of these tasks on failure.

These cloud programming and container orchestration systems consist of several key components. At the core is a pool or cluster of machines with some resource isolation mechanism, whether full virtualization or something lighter-weight like containerization or Linux cgroups [150, 151]. These machines run a local monitoring program to communicate their state, configuration, and available resources to a central cluster manager, which may run on a single machine or group of machines running some consensus protocol. Users then submit tasks to this cluster manager, either directly by specifying individual binaries and their dependencies, or through a framework which parses high-level application code and submits one or more jobs to the cluster manager on behalf of the user. Finally a scheduler, either executed by the cluster or by the framework, places tasks onto individual nodes based on their resources and the cluster manager completes the execution of these tasks and monitors their state.

This high level architecture is consistent across the earliest cluster management systems such as Google's Borg [150], later adaptations of these cluster managers like Omega and Kubernetes [133, 152], application-specific frameworks and resource managers like Hadoop's MapReduce and YARN [132], and other resource managers like Apache Mesos [134]. Often complementary services are also run on the cluster to aid in programming like a distributed file system [153]. There are, however, several key differences in how a user or framework interacts with each of these cluster managers.

Borg and Kubernetes both allow jobs to be deployed with a set of configuration files which specify environment settings, networking requirements, failure semantics, and scheduling needs within the cluster. The cluster manager is responsible for applying this configuration state and using an internal scheduler to place tasks such that they satisfy the configuration [133, 150]. Because the configuration files specify a state, the their controllers will also try to maintain that state, restarting jobs on failure, automatically rescheduling tasks upon node failure, and even preempting and moving tasks to meet the needs of higher priority jobs. This gives a user little control over task scheduling intricacies and lends these cluster managers to long-running container orchestration workloads as opposed to faster-paced, interactive workloads. YARN is similar in that it is intended to only work with one computing framework, Hadoop's MapReduce, and includes a scheduler in the resource manager which knows how to schedule MapReduce jobs [132]. The tight coupling of these frameworks to their programming models is why we do not consider them "general-purpose" in Table 4.1, although dynamic configurations could certainly be applied by a framework rather than a user in Kubernetes or Borg to make them more general-purpose.

Mesos goes to the other extreme, providing basic resource management, task distribution, and isolation, but provides no scheduling or task monitoring facilities [134]. Instead it is expected for computing frameworks to query the current resources of the cluster and the states of their currently running tasks, and use this information to submit new tasks or task restarts along with their placement information to Mesos through the Mesos API. This was intended to allow for multiple non-cooperative frameworks, each with their own scheduling constraints, to share a cluster. Frameworks that use Mesos are therefore necessarily programmatic, as users cannot easily express full scheduling decisions or directly monitor the tasks that they submit.

While all of these systems directly enable utility computing, as shown in Table 4.1, none of them can operate on resource-constrained sensors. All expect isolation mechanisms that depend on virtual machines, containerization, or Linux cgroups. The components of their resource managers which run locally on each node are often too resource-intensive to execute under memory and CPU constraints, and they all expect the cluster to be deployed within the same subnet, which is not possible in most sensor network deployments.

More fundamentally, these cluster managers target almost exclusively compute, memory, disk, and networking resources, and make scheduling decisions based only on the availability of these resources and on data locality within the cluster [134]. The resources they manage are largely uniform, and a unit of CPU or memory is comparable to other nodes in the cluster. A cluster manager for sensor networks must also manage sensors and actuators as available resources and make scheduling decisions based on their capabilities and other metadata such as physical location. Furthermore, sensor nodes, gateways, and cloud machines that could be part of a sensor network cluster have vastly different capabilities by orders of magnitude, so a scheduling algorithm would need to be aware of this heterogeneity of CPU and network performance when making scheduling decisions.

Non-standard Device Management in the Cloud

With the recent rise of GPUs, neural-networking accelerators, and FPGAs in datacenters, cluster managers have started to enable scheduling based on the presence of devices beyond CPU, memory, networking, and disk. This is similar to the need for a sensor network cluster manager to view sensors and actuators as resources. Kubernetes supports a device plugin which allows for each node to register devices by vendor and type with the Kubernetes node manager (kubelet) [154], and Mesos's resource types in principle allow the registration of any resource, and only limits the types of resources by convention. These capabilities are limited, but can serve as a basis for incorporating sensors as resources in a managed cluster of sensors.

4.4 Resource Management and Programming Frameworks for the Edge and Fog

Many projects have pushed to form clusters of computers outside of cloud settings, including projects which attempt provide computing as a shared utility in "edge" and "fog" contexts. Volunteer Edge Computing systems enable the distribution of computation for a shared goal across wide-area networks of computers, addressing challenges in resource monitoring, management, and task scheduling [155, 156]. These efforts were eventually extended to mobile phones as they rose in ubiquity. FemtoClouds and Serendipity both focus on in-network compute and localized clouds for clusters of phones, enabling phones to offload compute to other nearby phones with more available resources [157, 158], where as Maui proposed a programming framework for the automatic distribution of code between phones and other machines for phone resource optimization [159]. These projects share the goal of creating clusters out of physically distributed, mobile, and somewhat more resource constrained machines, and their proposals for scheduling and networking contribute to the goal of sensing as a shared utility, but none of these systems directly target highly resource-constrained or non-unix-based systems.

The rise of smart homes, the IoT, and generally the presence of programmable gateways and computers in the local network has led to a dramatic increase in interest in fog and edge computing in recent years. We should note that, somewhat paradoxically, "edge" in this case does not refer to the most resource-constrained nodes at the end of a network graphs, but instead the gateways, access points, base stations, and other programmable in-network devices that are the last hop before the most resource-constrained devices. This field of work claims similar advantages as those discussed in Chapter 3—by placing compute in the network on these local gateways, latency can be reduced, reliability can be increased, and privacy can be better preserved. We see that underlying technologies such as containerization and other isolation mechanisms are being adapted to run on the architectures and processors commonly found on these devices [135, 160, 161].

Multiple resource managers and distributed programming systems have been proposed to facilitate the programming and scheduling of code on these edge and fog devices. Most of these

architectures are quite similar to those used to manage and program cloud-based resources. ENORM proposes a resource manager for gateway-class devices that can enable local devices such as phones to offload compute to the gateways [137]. ParaDrop similarly proposes a resource manager and Kubernetes-like configuration-based programming framework that allows services to be deployed on access points inside of containers [138]. Other projects, such as DDFlow, focus on the programming model, and schedule dataflow tasks throughout the network of devices [136]. The work in this space is quite extensive, and often algorithmic, focusing on the optimization of scheduling algorithms for specific constraint scenarios. Hong et al. provide a good overview and taxonomy of these efforts [116]. Finally, Amazon AWS GreenGrass and Microsoft Azure IoT provide similar solutions, extending container orchestration and function-as-a-service architectures to local Unix-class gateways. These gateways can receive data from local sensors and process the received data [139, 140].

Unfortunately, none of these systems target or can support resource-constrained sensors, and all of them regard such devices as static data forwarders. The misappropriation of the word "edge" to a class of devices that are not actually at the edge-most connected computing devices in the network suggest that perceived lack of capability and programmability of these resource-constrained devices. While the algorithms developed to schedule under constraints more similar to a resource-constrained sensor network could help, and the programming models to program these gateway-class devices are relevant to programming models for sensors, none of these systems offer a functional codebase upon which to construct sensing as a shared utility.

Chapter 5

Resource Management for Resource-Constrained Sensors

After reflection upon failing to sufficiently enable sensing as a shared utility in the Signpost platform, as discussed in Section 2.7, it was clear that achieving the goal of utility sensing requires three key abstractions: (1) Multi-tenancy not only in hardware but also in software because it de-risks code iteration by ensuring a reliable computing base, enables simultaneous testing and development, and amortizes the hardware, deployment, and maintenance cost among multiple users or applications; (2) Macroprogramming frameworks abstract the intricacies of writing and deploying distributed applications, which are fundamental to sensor networks, and (3) Generality, or the ability for multiple programming models or frameworks to coexist, which is critical to meeting the needs of multiple applications or a single application moving through its life-cycle from discovery, to testing, to production.

While alluded to in Chapter 4, we now briefly discuss the interdependence of multiprogramming, macroprogramming, and generality, as it relates to resource-constrained sensors. In Section 4.1, we see that the ability to multiprogram resource-constrained sensors has been present for nearly two decades, however these solutions did not enjoy broad adoption or experience long-term maintenance. We propose that it is because without macroprogramming and generality, there is no clear advantage to multiprogramming sensors. Consider early multiprogramming systems for mainframes or even desktops; in both of these scenarios there was at least one person, and often many people for every computer. To take advantage of the multiprogramming systems, multiple users would connect and submit programs to the computer. Even in the desktop space where the ratio of users to computers is more balanced, users can manually control multiple tasks for the computer. In a sensor network, it's simply not feasibly to engage with multiprogramming through direct interaction with each sensor, as there are too many. The only way to take advantage of multiprogramming is through the leverage offered by a macroprogramming tool.

From the other perspective, macroprogramming is much less useful, and nearly impossible, without multiprogramming. First, because multiprogramming is the foundation for reliable code updates, and the interactivity that is theoretically enabled by macroprogramming largely

comes from the ability rapidly iterate and test new ideas across an entire network of sensors. It is not something one would risk if it could render hard-to-reach sensors inoperable. Second, because the ability to take advantage of generality and multiple macroprogramming frameworks depends on deployed sensors continuing to be useful during testing and iteration, which requires multiprogramming. Beyond the inherent difficulties of maintaining an engineering project that starts as a research project, it may be that neither the multiprogramming nor the macroprogramming solutions in Chapter 4 continue to exist because they were not built together, and there was never a reason for their convergence.

To enable all three of these functions simultaneously, we look to solutions which enable utility computing in the modern cloud, cataloged in Section 4.3. At the core of each of the solutions is a resource manager. The resource manager coordinates multiprogramming by working with node-level isolation mechanisms to distribute, start, and restart tasks that are limited to specific resources. In the cloud these are often containers, but that is not fundamental to the resource manager itself. The resource manager enables programming frameworks to macroprogram clusters of computers by providing an interface for tasks to be started on multiple nodes with these resources constraints. Differing degrees of generality in programming frameworks are enabled depending on the specific resource-manager implementation and how it distributes resources and handles task execution requests.

Adapting existing Resource Managers

When we set out to implement a resource manager for resource-constrained sensors, the first consideration was whether an existing resource manager, such as Kubernetes, YARN, or Mesos could be adapted to collect resource information and distribute tasks to a cluster of sensors [132–134]. These resource managers consist of two parts: the resource manager central controller which centralizes and assigns resources and the resource manager agent, which runs on every node, collecting information about the available resources, launching tasks upon request, and updating the central on the status of running tasks. We started by evaluating whether the resource manager agent could run directly on a sensor node, and we quickly discovered, as shown in Table 5.1, that this is not feasible due to the memory constraints of our target processors.

Resource Manager	Average Memory Utilization
Mesos Agent	6 MB
Kubernetes Kubelet	2 MB
YARN NodeManager	194 MB

Table 5.1: Memory usage of existing resource manager agent processes. We see that existing resource managers can't be used directly on sensor nodes at least in part due to their memory footprint, which well exceeds the less than 128 kB of RAM available on resource-constrained sensors.

We then considered the possibility of rewriting the resource manager agent to use an existing resource management protocol. This was possible but had some significant shortcomings. All existing resource managers which we explored were designed to run within a subnet rather than over a wide-area network. Additionally, their networking protocols are not suitable for resource-constrained sensors. A proxy that translates between protocols and makes the resource manager available outside of the subnet could address these issues.

However, the core problem with existing resource managers is that the types of tasks they are able to distribute and manage are not easily extensible, nor are the types of resources. A resource manager for a cluster of sensors will need to be able to distribute tasks that are not containers, as containers depend on Linux Cgroups, and forking and modifying the central cluster manager was overkill and deemed untenable. We therefore sought to design and implement our own resource manager that could distribute tasks among more resourceful servers and gateways, and resource-constrained sensors. We call this resource manager EdgeRM, and we describe it's design in the remainder of this chapter.

5.1 Design

The design and overall architecture of EdgeRM is similar to the resource managers used in cloud settings, especially Apache Mesos [134]. The high-level architecture is presented in Figure 5.1. Agent processes run on each node, whether that node is a resource-constrained sensor, a server, or a gateway-class Linux device. These agent processes monitor the available resources of the nodes and advertise that information to the resource manager's central controller. The agent is responsible for interfacing with the central to expose and maintain an updated list of available resources, as well as accept and launch tasks. Application frameworks then request from the EdgeRM central manager the list of available nodes and resources, and submit tasks to nodes by responding with requests to use those resource for specific tasks. These tasks are forwarded to the nodes' agents, which launch and monitor the tasks. The frameworks can request additional resources or check on the status of a running task.

While this architecture is similar to resource managers that have been developed in the past, each component of EdgeRM was engineered to be suitable for a cluster of resource-constrained sensors. We discuss these considerations in the remainder of this section.

Scheduling

One key distinction between the resource managers discussed in Section 4.3 was the placement of the scheduler within the architecture of the system. Container orchestration systems like Kubernetes and Borg, and single-purpose systems like YARN, tend to integrate the scheduler into the resource manager, taking hints from applications about node placement or "affinity" but ultimately mapping tasks to nodes based on the available resources. Mesos, on the other hand, takes the opposite approach expecting the application framework to perform all

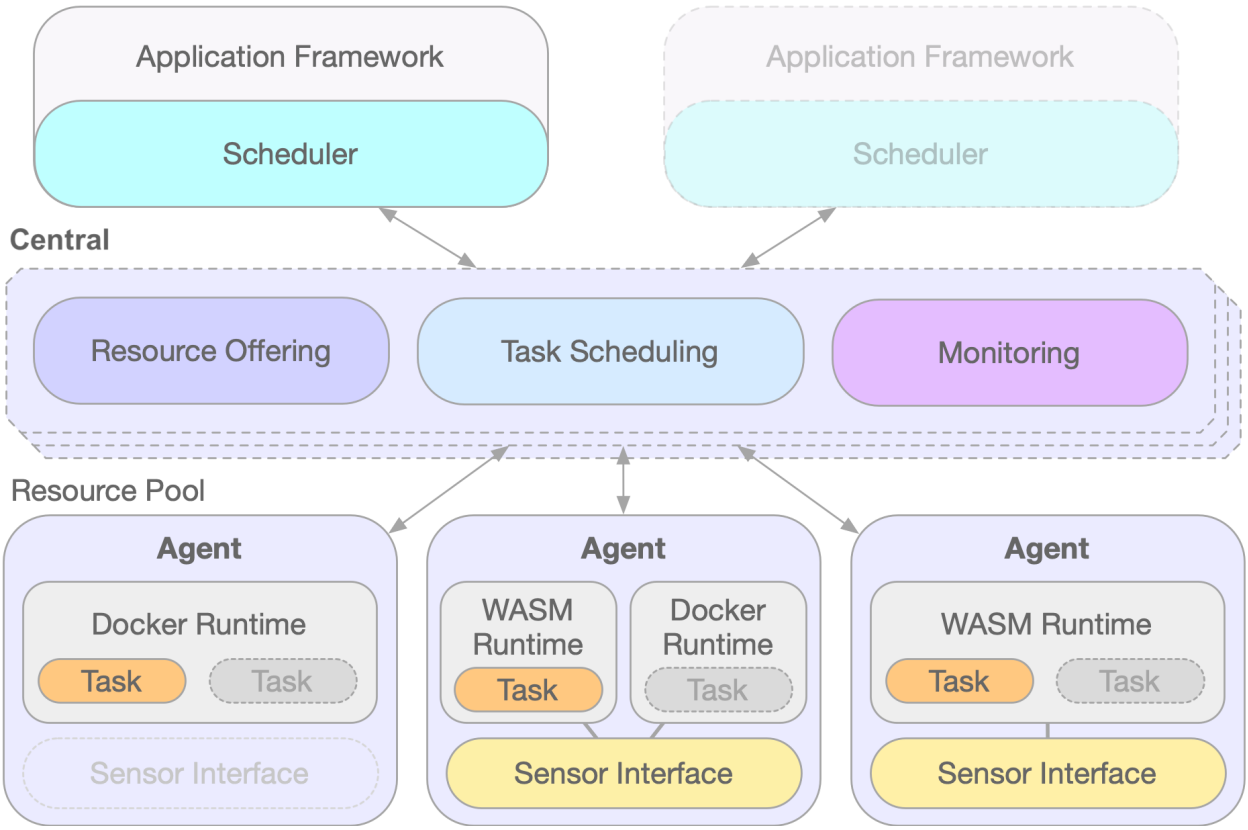


Figure 5.1: EdgeRM Architecture. EdgeRM agents send available resources to a central to be offered to multiple frameworks with their own independent schedulers. Compared to cloud-targeted resource managers, EdgeRM includes support for WASM runtimes, adds extended resource types, and uses communication protocols designed for resource-constrained agents with attached sensors.

scheduling based on the available resources. To meet our goal of generality, we take the latter approach with EdgeRM, as can be seen in Figure 5.1.

While EdgeRM could meet its goal of generality while allowing multiple application frameworks to submit tasks through an integrated scheduler, in practice, given the diversity of resource-constrained sensing applications, it is difficult to imagine creating a unified scheduler that can respond to the needs of all applications. This is because in a sensing scenario, the schedule itself may change due to the data reported by a sensor. Consider, for instance, a sensor changes rooms and needs to load a new application in response to this mobility, or an application framework that withholds an analysis task until a rare event occurs. It seems untenable to create a representation that allows the application framework to express its scheduling needs in these scenarios. Also, due to the diversity of data transport protocols for resource-constrained sensors, a resource manager should remain in the control plane, managing only those resources described in Section 5.1, and not data returned by the sensors.

To enable this flexibility, just like Mesos, EdgeRM sends out "offers" of a subset of the available resources to application frameworks, and frameworks respond with pairings of tasks and the resource which they can use. This approach stands in contrast to frameworks that submit a list of tasks and resources. A framework in EdgeRM processes the full set of these requests with its own scheduler.

Resource Definition and Types

A resource manager is primarily used to facilitate scheduling decisions by one or more schedulers. In the cloud, the information used to make scheduling decisions is the information tracked by the resource manager. For example, scheduling decisions are primarily made based on a machine's CPU, memory, and data locality, all of which is inherently tracked by the resource manager.

As we extend to sensors placed throughout a physical environment, this no longer holds. While schedulers still need to make decisions based on CPU and memory, they also need to make scheduling decisions based on location, network topology, and physical events. This raises the question—is this type of meta information a resource, and is it the resource manager's role to aggregate this information for a scheduler?

We argue that such metadata should not be handled by the resource manager because collecting this information often requires precious resources, and it is impossible to know a priori which types of metadata a given scheduler may need or the rate at which that metadata should be collected. Further, as discussed in Section 5.1, it is not practical to collect this information given the diversity of protocols through which data is reported.

Therefore, EdgeRM supports: (1) Resources, which are physical devices that are isolated by and accessed through the resource manager's execution environment, and (2) Attributes, which are static device properties that do not change through the deployment of a device. These are the same categories used by Mesos, but their definitions require clarification.

Resources

EdgeRM initially used Mesos' resource types: scalar, text, range, and set, and we found these to be sufficient for both traditional resource types like CPU, memory, and networking ports, as well as previously unconsidered resource types that are important for embedded systems, such as energy [134]. These resource types, however, assume that a task understands the methods by which these resource may be accessed, and it also assumes that these resources must be strictly isolated between tasks. Neither of these assumptions hold for sensor devices which must be accessed through EdgeRM.

Devices

To facilitate sensor devices, we introduce a new resource type "device." Similar to other resource types, device types have a name, but they also have a handle by which they can

RPi with Camera	Server	Embedded Sensor
Resources:	**Resources:**	**Resources:**
Scalar: CPU - 1.0	Scalar: CPU - 4.0	Scalar: CPU - 1.0
Scalar: Memory - 4 GB	Scalar: Memory - 8 GB	Scalar: Memory - 50 KB
Scalar: Disk - 8 GB	Scalar: Disk - 100 GB	Scalar: Disk - 100 KB
Range: Ports-3000-3005	Range: Ports-3000-4000	Scalar: Power - 1 mW
Device: Camera /dev/vchi		Device: Accelerometer - acc1
shareable- True		shareable - True
API - raspistill		API - WASIv1
		config - [1g, 2g, 4g]
		maxSampleRate - 4000
		rateDivisor - 2
		Device: Humidity - hum1
		shareable - True
		API - WASIv1
		config - [heat_on, heat_off]
		maxSampleRate - 5
Attributes:	**Attributes:**	**Attributes:**
Text: OS - debian-armv7l	Text: OS - debian-amd64	Text: OS - zephyr
Set: Executors - [Docker]	Set: Executors - [Docker]	Set: Executors - [WASM]
Text: ID - picam01	Text: Domain: pub.com	Text: ID - sensor01
	Text: ID - server1	

Table 5.2: Resources and attributes in a EdgeRM deployment. All devices list common resource types such as CPU and memory, however resources such as devices, domain names, and the available power are unique to a wide area sensor deployment. Device resource types have extended properties that correspond to their non-traditional resource usage. The shareability field indicates the ability for multiple applications to sue the device, the API field indicates the API through which devices are accessed, the config field indicates mutually exclusive configuration sets a device may be placed in and sample rate fields enable EdgeRM to make offers of specific sample rates for sensors while still enabling the sensor to be shared.

be uniquely referenced, a flag to indicate whether or not they can be shared, and a field to indicate an API by which the device can be accessed. These fields are necessary to handle device heterogeneity and enable schedulers to ship tasks that align with the supported sensor access API. Devices may also note a set of possible mutually exclusive configurations they can accept and two sensor specific fields about sample rates. For the configuration field, once a configuration has been offered and a task issued within a specific configuration, all subsequent offers are restricted to match the claimed configuration. This configuration setting may be overly restrictive, as a configuration could successfully change over time without conflict, but it is simple and ensures isolation between applications.

The two sample rate fields—a max sample rate and sample rate divisor—are basic provisions to enable the EdgeRM central to reason about the way data can be shared within the device hardware. This is to contend with the fact that often sensors may be sampled at a continuous rate, but once that rate is chosen, there are a finite but large set of other

sample rates compatible with that rate. For instance, a sensor sampled at 10 Hz could support sampling at 5 Hz by providing every other sample to the lower rate application or 20 Hz by doubling the sampling rate and providing every other sample to the original application, but not 3 Hz as it is not divisible by the original rate. We simplify this finite but large set down to two fields, a max rate and a divisor, where valid sample rates are any factor or multiple of the divisor below the maximum rate. Applications may request a sensor device with sample rates up to the maximum rate specified by the device, and the EdgeRM central will adjust the divisor field to indicate the new possible rates. Devices may preemptively set the divisor to prevent requests of poorly compatible rates such as large prime numbers. These fields are optional and primarily necessary for applications which need to sample at a higher rate than would be possible using software sampling within their given CPU allocation.

Attributes

Attributes are system constants which do not change throughout a device deployment. We commonly implement attributes specifying a unique device ID, the architecture and OS of a device, the executors supported by the device, and the public IP or domain name of a device if one is present. These attributes are critical to making scheduling decisions. For instance, many tasks can only be scheduled on machines that are publicly reachable because they need to collect sensor readings from a wide range of devices behind NATs. Example platforms and their EdgeRM resources are shown in Table 5.2.

Networking and Communication

Resources and attributes are coordinated through the EdgeRM messaging protocol. Table 5.3 provides an overview of the messaging API. Unlike other resource managers that establish and maintain bi-directional communication between entities, EdgeRM opts for a client-server communication model that is initiated by the agent or framework. The reasons for this are two-fold: first, the overhead involved in establishing and maintaining long-lived persistent connections on resource-constrained agents is impractical. Second, these clusters are often comprised of devices spanning wide-area networks and devices located behind NATs. As a result, only the central is assumed to expose a public network address; thus the central serves as the centralized endpoint connecting the cluster components. Agents that are publicly accessible can include their endpoint information as an attribute within the resource manager.

Message Type	From → To	Fields	Actions Taken
Ping	Agent → Central	AgentID*, PingRate*, Resources, Attributes, TaskStatus	Register Agent, Update Agent State
Pong	Central → Agent	Ack*, TaskInfo	Run or Kill Task if Requested
RequestOffers	Framework → Central	—	Collect Offers
ResourceOffers	Central → Framework	OfferID*, Array{AgentID*, Resources*}	—
RunTask	Framework → Central	OfferID*, AgentID*, TaskInfo*	Queue Task for Agent
TaskStatus	Central → Framework	Ack*, TaskStatus*	—
SubMessage Types	—	Fields	—
Resources	—	Array{ResourceName*,ResourceType*,ResourceValue*}	—
Attributes	—	Array{AttributeName*,AttributeType*,AttributeValue*}	—
TaskInfo	—	TaskID*, TaskContainer*, TaskEnvironment, TaskResources*	—
TaskStatus	—	Array{TaskID*, TaskState*, ErrorMessage}	—

Table 5.3: EdgeRM messaging protocol. An overview of the messages between different components in EdgeRM and their fields, with sub-messages separated for clarity. Required fields are marked with *. All messages are client-initiated, where the Agent and Framework act as clients, and the central is the server. The central then responds, piggybacking information onto the response. This allows agents to control their energy usage at the cost of higher latency for task execution, and it allows for agents and frameworks to communicate with the central from behind a NAT. Many fields are left optional so that agents can further limit communication to strictly what is necessary to keep their resources and task states up to date. Currently HTTP, CoAP, and WebSockets are supported the communication protocol, however any client-server protocol could be used.

EdgeRM also operates over additional networking protocols compared to other resource managers. Currently EdgeRM exposes HTTP, CoAP, and WebSocket endpoints. This serves devices with a variety of capabilities and operating systems. The EdgeRM central can be expanded to handle additional networking protocols as the protocols used by resource-constrained devices change over time. The downside of this client-server model and protocols is of course that tasks cannot be started asynchronously or at a rate higher than the rate at which the agent contacts the central. We propose and have implemented rate adaptation to enable rapidly-deployed tasks, but task requests will still need to wait until the next scheduled agent ping to start this process.

Messaging Protocol

The EdgeRM messaging protocol is divided into three parts:

Resource Aggregation. Each agent connects to the cluster via a *Ping* issued to the central. The ping contains agent details, resource information, device attributes, current availability, and task status. The central aggregates the information from agents within the cluster to maintain an updated view of the resource pool. If an agent does not ping the central within its specified window, the central does not consider its resources available, but to accommodate resource-constrained devices the central imposes no requirements on the rate at which an agent pings.

Resource Offering. Frameworks that seek to deploy tasks over the resource pool issue a *RequestOffers* message to the central. The central replies with *ResourceOffers* containing a subset of the current available resources based on the resource offering policy, which is configurable by the system administrator. Each offer is associated with an expiration, at which point unclaimed resources are made available for subsequent framework offer requests.

Task Scheduling. A framework can claim an offered resource by issuing one or more *RunTask* messages specifying a task to deploy on resources contained within the offer. Included in this message are configurations and environment variables necessary to launch the task on the specified agent. The central forwards the task request to the chosen agents on their next ping by attaching a RunTask message to the pong response. Frameworks can monitor tasks via a ping request that collects *TaskStatus* updates, and can issue requests to kill running tasks.

Interface	Functionality
configureDevice(deviceID, config)	Set a device's config to one of the possible config options.
getSample(deviceID, value*)	Synchronously sample sensor data.
getSampleAsync(deviceID, callback(value))	Asynchronously sample sensor data.
getSamplePeriodic(deviceID, periodMs, callback(value))	Periodically sample sensor data.
getSampleBuffer(deviceID, periodUs, buf*, len, callback(buf*))	Read sensor values into a buffer of set length at the specified sample rate.
getSampleBufferContinuous(deviceID, periodsUs, buf1*, buf2*, len, callback(buf*))	Repeatedly read sensor values into buffers of set length at the specified sample rate, switching between buffers.
getSampleWhen(deviceID, condition, threshold, callback(value))	Call callback when the device returns a value meeting the condition (greater than or less than) and threshold.
stopSampling(deviceID)	Stops any continuous sampling.

Table 5.4: WebAssembly Sensor Interface. While a standardized WASM interface for sensors is under development [162], we develop our own interface to balance the needs of both low-power sensing and high-sample rate sensing. The above interface allows applications to use few resources when waiting on a sensor to meet a certain value with the *getSampleWhen* API call. This enables the runtime to push thresholds into hardware and sleep if possible, conserving resources. The interface also enables sensors to perform high rate sampling of signals such as an accelerator or audio interface with the the double-buffered *getSampleBufferContinuous* call. These calls are inserted into the WAMR runtime, and the resources, sample rates, and configurations they consume are checked against the task's allocated resources before execution. Time handling calls which take runtime resources are allocated to the task's CPU utilization.

Isolation and Execution Environments

A key requirement of EdgeRM is a general execution environment and runtime suitable to the resource-constrained and low-power processors driving sensors. Linux-class machines in an edge environment may also benefit from low latency and low overhead task execution compared to relatively heavier-weight containers. To this end, EdgeRM adopts WebAssembly [163], a portable instruction format initially designed for secure and sandboxed computation in browser environments with near-native performance, that has also recently received attention as a suitable intermediary representation for embedded device applications [128]. Specifically, we include the WebAssembly Micro Runtime (WAMR) in the embedded agent [117]. Embedded agents that run WAMR can receive and launch arbitrary tasks compiled to WebAssembly. When augmented with a suitable sensor access interface, WebAssembly enables general and isolated computing.

To ensure the WASM runtime effectively isolates and switches between tasks, the runtime periodically preempts and switches tasks, ensuring no task runs longer than its allocation within an agent-implemented time slice. All access to hardware, networking, and sensors is mediated through an API in which memory bounds can be checked before performing any action. This API closely mirrors the WASI API with additional methods for sensor access and networking [162]. Time spent in the runtime on task-specific functionality is deducted from the task's CPU allocation. The API is designed such that memory used in a call resides within a task's memory region. Networking resources are tracked by monitoring the number of packets sent. EdgeRM could also adopt additional isolation mechanisms and execution environments. For instance, the distribution of Tock binaries would be a suitable, although less portable option [53]. We need to implement all of the additional usage monitoring and isolation mechanisms in each new execution environment however, which takes non-trivial development time and effort. The agent is capable of running WASM tasks in any environment with a suitable networking stack and threading library.

For non-resource-constrained machines, EdgeRM also allows the execution of containers. This is especially practical for long-running tasks such as databases or data receivers, for which containers often already exist. The agent uses existing isolation mechanism present in the Docker container runtime to isolate container resources.

Sensor Interface

Unlike Docker containers, where access to system resources is available through a well-defined POSIX interface, WebAssembly enforces a sandbox that constrains applications to structured control flow within a pre-allocated linear memory region. Any sort of external resource, such as filesystem access, is provided by explicitly granting functions to a WebAssembly module. While this design is well-suited to a resource management abstraction, the actively developed and in-progress WebAssembly System Interface is still developing an interface for sensors [162].

We also note that the the virtualization and isolation of shareable hardware like a sensor is significantly different than a resource such as CPU, memory, disk or networking. Unlike CPU and networking the amount of access is not a sufficient isolation mechanism because the timing of access to sensors is critically important for an application. Unlike memory and disk, sensors cannot be partitioned into regions in which tasks receive full control over their section.

To support portable sensor access for WebAssembly tasks across the resource cluster, we propose a system interface for sensor access in WASM and create a custom virtualization layer which attempts to serve all calls made to the sensor interface. When loading a module that requires sensor resources, the runtime explicitly grants the sensor API to the executing task, and each access to that API is validated by the runtime before a platform-specific implementation of the API function is called.

The sensor access API is summarized in Table 5.4. Sensors are defined by a set of capabilities, which refer to the raw underlying sensors (e.g. temperature), and set of possible configurations. Tasks may change the configuration of a sensor for which it has access if that configuration was granted at the time of task issues by EdgeRM. Tasks can call the API with specific sampling parameters so long as those parameters are within the sampling constraints that were granted to the task. A full description of how EdgeRM handles sampling constrains is discussed in Section 5.1.

To virtualize sensor access, the EdgeRM agent reads the set of requests for a sensor and attempts to merge them into a cohesive sampling strategy. In practice, the agent configures the sensor for highest sampling rate currently being requested and subsamples the sensor data for lower sample rates. For single-sample APIs such as *getSample* and *getSampleAsync*, the sampling layer either initiates a single sample or returns the most recent sample from continuous sampling. The addition of the *getSampleWhen* API was critical to allow sensors to push interrupt-driven sampling into the sensor peripherals and sleep the processor to save power, although it should be noted that often multiple thresholds cannot always be combined. Other extensions to this API are planned, such as the ability to continuously sample a sensor, but only return to process those samples when an external condition is met.

This proposed sensor access API is not the only API supported by EdgeRM, and each EdgeRM agent specifies the API by which a device is accessed, however standardization of sensor access will improve task portability. The features of this API such as the ability to interleave high-rate samples and push simple filters into hardware will be critical to a more standard sensor access API.

Resource Offering and Fairness

When multiple frameworks are sharing a cluster of resources, usually some policy is applied to ensure that those resources are fairly allocated among frameworks. Mesos uses dominant resource fairness (DRF), which extends min-max fairness from a single resource type to multiple resource types [134, 164]. DRF makes assumptions about resource type and cluster usage that do not hold for clusters of resource-constrained sensors. First, it assumes that

the quantity of a resource is representative of its value across the cluster of resource. For example, it assumes that 1 CPU on a sensor node is equivalent to 1 CPU on a cloud server. This is clearly not true as many tasks may be strictly associated with a specific sensor or gateway within the network. Second, the way that resources are offered using DRF in other resource managers assumes that tasks are relatively short lived, and that as frameworks leave or join, their resources can then be redistributed according to the new DRF allocation. This is not true in sensor networks, and a newly joining framework could be indefinitely starved of resources. Lastly, DRF assumes that resource allocation values can be summed and compared, which may not be true for sensor resources or configurations.

We implement several modifications to the most straightforward DRF algorithm to handle these cases. To better handle the distribution of tasks across nodes, we pool resources based on whether a node has device-type resources or not. That is, CPU on a node without devices is counted as a separate resource from CPU on a node with devices and the same goes for memory and other resource types. This is an imperfect heuristic, but it does make the resources on sensor nodes distinct and more comparable to one another during resource distribution. When a framework puts a device into an exclusive state, or uses a non-shareable device, we also assume that the framework has used either all the resources on that node, or its proportion of the resources on the node if it is still being shared. This disincentivizes frameworks from using devices in a way that cannot be shared by other frameworks because all resources for the exclusively configured device are counted towards the dominant resource share for the framework.

To address the issue of long running tasks that cause resource starvation for newly-joined frameworks, we propose two policies. The first, more aggressive, policy with higher utilization kills tasks and offers new resources upon a new framework joining. This allows higher utilization of the cluster, but creates unexpected failures on reallocation. It also allows joining frameworks to forceably terminate the tasks of other frameworks, which is a poor incentive. The second policy implements a greedy pseudo-framework to reserve resources on the cluster. When a new framework joins, these reservations are released and the reservations of the pseudo-framework take a smaller amount of the cluster. This policy balances early frameworks which receive more of the available resources with long-running tasks, while still allowing new frameworks some resources when they join.

EdgeRM enables other properties to be configured to control resource allocation and fairness. At deployment, some frameworks can be granted preemptive priority on a cluster. Frameworks may also be given weighting as described in the original DRF algorithm [164]. We also note that while EdgeRM currently does not have the notion of a user, and a user could sway the DRF algorithm simply by creating multiple frameworks, the same DRF algorithms could be applied to users between frameworks if users authenticate to the EdgeRM central and resources were associated with users rather than frameworks. This authentication layer is planned as future work.

Fault Tolerance

A fault tolerant central is critical to system reliability, as both frameworks and agents rely on the central coordinate resources offering and task execution. We designed the central to be able to reconstruct its complete state from the pings it receives from agents and framework schedulers. As such, recovery from a failed central simply requires that connected frameworks and agents redirect requests to a backup or standby central, and the DRF algorithm converges as more information is received. Fault tolerance is achieved by running backup centrals in standby mode. A standby central registered with the current central is added to a configuration disseminated to frameworks and agents with a total central ordering. Upon failure, the first standby central is promoted. Requests issued to any standby central are redirected to the current central. To lower latency for state reconstruction multiple centrals could also be run and coordinated through a consensus protocol.

5.2 Implementation

A POSIX-based agent implementation is written in Python which uses Docker as its container runtime and WAMR as its WebAssembly runtime. Embedded agents are implemented for the Zephyr OS [141] and the Particle [165] embedded platform. Both of these systems use WAMR to execute WebAssembly modules. The embedded agent is currently capable of executing seven simultaneous WebAssembly tasks. This number is limited by both memory and the footprint of a WAMR WebAssembly runtime, which uses excess memory due to memory alignment requirements that could be mitigated through engineering effort. The EdgeRM central is implemented in Python along with a small local database to track system state.

Porting the edge agent to a new platform requires the implementation of a timer, malloc, free, thread creation, UDP send and receive, and functions to access the sensors which are callable from WebAssembly modules. Implementations are encouraged but not required to implement part of a standard system interface for WebAssembly modules. This standardizes sensor access and other common functionality such as timing and networking in WebAssembly [162]. While this functionality is not always present on embedded platforms, it is supported by most embedded operating systems. More advanced parts of the edge agent, such as double-buffered sampling and high sample rates may take greater implementation effort, such as the implementation of DMA transfers from a sensor to the sample buffer provided by a task. We note that this implementation effort would naturally be required for any application that needs such a high sample rate and is not purely a requirement of EdgeRM. We implement several frameworks for EdgeRM. These frameworks and their associated libraries are discussed in Chapter 6.

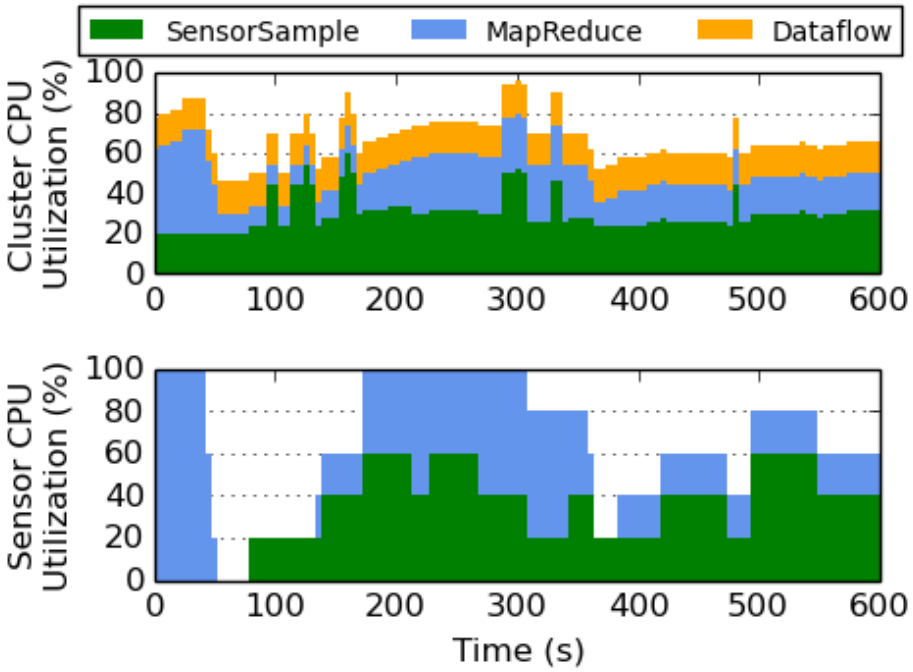


Figure 5.2: Utilization of the edge cluster (top) and a single sensor (bottom) by three programming frameworks over a ten minute period. Multiple users deploy jobs to the edge cluster through three programming frameworks using EdgeRM. These three frameworks are capable of multiplexing the cluster and can deploy tasks on both sensor and server nodes simultaneously. The mediation of resources through EdgeRM enables multi-tenancy on constrained, embedded devices that are traditionally singe-purpose.

5.3 Scalability and Overhead

Our evaluation begins by demonstrating the multi-tenancy enabled by EdgeRM and showing a snapshot the cluster's utilization (§5.3) when tasked by multiple frameworks. We then perform an overhead analysis of the EdgeRM agent implementations (§5.3) and the WebAssembly execution environment (§5.3). The evaluation depends on the use of several frameworks which use EdgeRM. More discussion of these frameworks can be found in Chapter 6.

EdgeRM Cluster Utilization

To demonstrate the multi-tenancy enabled by EdgeRM we collect the share of the cluster CPU used by three frameworks as multiple users use the cluster. This 10 minute snapshot is shown in Figure 5.2. We see that multiple frameworks and users are able to share the cluster and its resources, with short-running tasks such as those generated by the Sensor Sample and Filter framework creating spikes in utilization. We also see the CPU utilization of a single sensor, seeing that tasks from both the Sensor Sample and Filter framework and the MapReduce

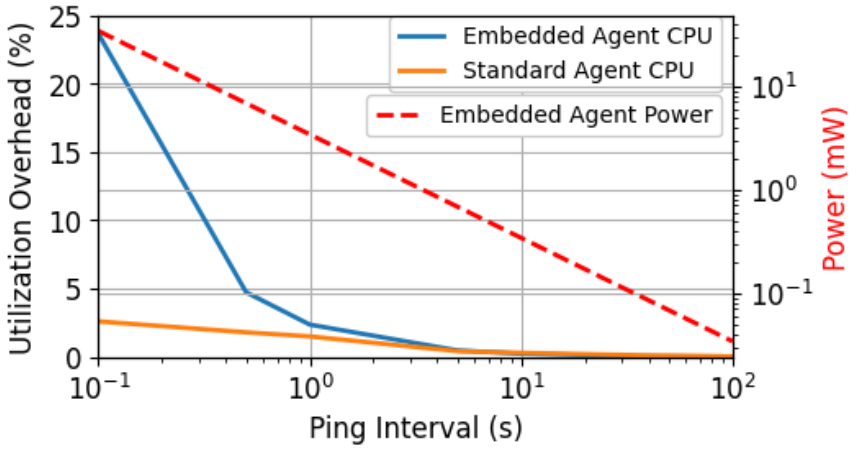


Figure 5.3: Compute and power overhead of the EdgeRM agent, plotted as a function of agent ping interval. As the ping interval is increased, overhead falls proportionately. On the embedded agent (evaluated on an NRF52840 MCU) ping intervals greater than 1 s have CPU utilization below 5 %, and ping intervals greater than 100 s have a power consumption of less than 34 µW. A bounded exponential back-off on ping interval maintain interactivity while decreasing power.

framework are executing simultaneously. When resources are not available for more tasks, the frameworks may direct tasks to other sensor nodes as appropriate. The execution of multiple simultaneous tasks, and specifically the dynamic deployment of simultaneous tasks from multiple programming frameworks onto a resource-constrained embedded node would not be possible without EdgeRM. We note the use of MapReduce primarily for illustrative purposes.

Agent Memory and Code Size

The memory footprint and code size of the EdgeRM agent implementations are presented in Table 5.5. The Python agent implementation was profiled on a Raspberry Pi 3B+, corresponding to a 2.4% memory footprint overhead [166]. The embedded agent was profiled on an NRF52840 MCU. The resoruce manager implementation with the WARM runtime use 65 kB of code space and 2.6 kB of SRAM. An additional 22.3 kB of SRAM is needed for every task executed on the embedded agent no matter the task size (and more is required of the task itself uses more memory). This relatively high per-task overhead is primarily due to the minimum memory region of 16 kB needed to execute a task in WAMR, however we expect this could be significantly decreased with a more optimized implementation. The remainder of the code and RAM are used by networking and OS libraries that the EdgeRM agent uses, but would also be required by most embedded applications.

While the overhead introduced by EdgeRM on an embedded device is not insignificant, it nevertheless falls within the practical range of modern microcontrollers, especially when

Code Segment	Text (B)	Data (B)	BSS (B)
Total	317,918	3,748	90,460
OpentThread (Net)	149,116	—	38,087
Agent Library	15,764	—	1,401
WAMR Runtime	51,564	—	1,250
WASM Task (min. ea.)	—	—	22,269

(a) embedded agent

	Memory Usage (MB)
Python Agent	22.8

(b) python agent

Table 5.5: Memory and code footprints of the EdgeRM agent implementations. The embedded agent flash and RAM utilization are decomposed into constituent components. A significant portion of Flash and RAM utilization is due to the networking stack and the underlying OS, which would also be required by a monolithic firmware. Remaining unused memory is available to store and execute WASM tasks. The minimum memory for each task is 22,269 Bytes, which includes all task state, thread stack and heap, and the minimum 16,384B required to execute a WASM module.

considering the computational benefits enabled by integrating an EdgeRM agent into a computing cluster. On the NRF52840, which has 256 kB of SRAM, we are able to execute seven simultaneous WASM tasks, and we expect this number to increase with more optimized WASM runtime implementations and ever-growing MCU memory sizes.

Agent CPU and Power Usage

Compute and power overhead of the EdgeRM agent is directly proportional to the frequency at which an agent pings the central. Figure 5.3 presents the average compute utilization and power consumption as the interval between successive agent pings is increased. The standard agent compute utilization was profiled on a Raspberry Pi 3B+, while the embedded agent compute utilization and power consumption were profiled on an NRF52840 MCU connected to an OpenThread 802.15.4 network. Most of the CPU utilization and power draw is attributable to the networking operations required to ping the central. For powered embedded devices, a ping rate of greater than 1 s keep CPU utilization below 5 %; for energy-constrained devices, a ping rate of 10 s draws 340 µW and a ping rate of 100 s draws 34 µW. To conserve energy while still enabling fast iteration during interactive periods embedded agents can exponentially back-off their ping rate.

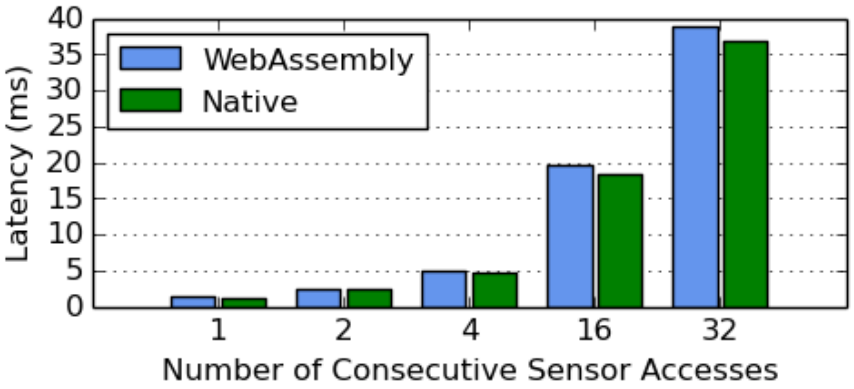


Figure 5.4: Latency overhead of accessing on-board sensors through WASM.Sensors are accessed a number of times using a WebAssembly task with the WASM sensor interface and access time is compared to directly accessing the sensor with through the underlying platform. WebAssembly introduces less than 5 % latency overhead.

WebAssembly Overhead

The overhead analysis of the WebAssembly execution environment is decomposed into a (1) sensor access latency overhead and a (2) compute overhead analysis. The former is encountered when tasks are issued to collect sensor data, while the latter indicates the overhead required to sandbox execution of pure compute tasks with the WebAssembly interpreter. Device memory consumed by the runtime is included in the memory footprint presented in Table 5.5.

Sensor access latency

Figure 5.4 presents a latency overhead analysis of the WebAssembly runtime and sensor interface with respect to native implementations. Latency was profiled by fetching temperature values from a BME280 sensor a varying number of times to indicate (1) the fixed startup cost of loading the WAMR runtime, and (2) the marginal overhead of individual sensor accesses, with respect to native.

The startup cost of booting the WebAssembly runtime is on the order of hundreds of microseconds, indicating a less than 10% overhead with respect to a sensor access. Once loaded, an individual WebAssembly sensor task access has a latency overhead of 5% with respect to the latency required by the underlying platform SDK. We also note that this latency is only present through the direct sensor sample API describe in Table 5.4. API calls which prompt the runtime periodically call the WASM module with sensor results would note incur this overhead.

WebAssembly compute overhead

The current implementation of the EdgeRM agent uses the WAMR runtime in interpreter mode, directly interpreting WASM bytecode. Currently WAMR reports that interpreted WebAssembly runs 11-16x slower than native code for common benchmarks such as Coremark and Fibonacci [149, 167]. We soon hope to integrate ahead-of-time compilation and execution, which runs at 85-95 % of native speed, as a service so that EdgeRM can better support computationally intensive tasks. EdgeRM could transparently compile WebAssembly to a target architecture if the architecture attribute is present.

5.4 Other Considerations

As an early system we envision EdgeRM and systems like it will continue to evolve. To be a fully functioning resource manager for resource-constrained devices, EdgeRM would need several additional developments which we discuss here.

Naming and standardization

For frameworks to effectively schedule tasks on nodes, the nodes and frameworks must agree on resource types and the names of resources. Interestingly, as long the resource is of a type that EdgeRM understands how to fairly distribute, EdgeRM is agnostic to and not prescriptive about the exact names of the resources themselves. This lassez-faire strategy starts to break down with the device resource as discussed in Section 5.1. In the device resource, specific fields in the resource definition such as *maxSampleRate* are used to ensure fairness during resource offering because fairness cannot be ensured by simply splitting a quantity of the available resource, and some standardization of this kind is likely necessary for the portability of frameworks between clusters of different node types. Ultimately naming and the standardization of names in a system must be solved organically or through a formal standardization process.

Energy Resources and Metering

One key resource of resource-constrained devices is their energy, and as a resource manager, allocating slices of energy, or more likely average power over some period time, would be necessary to ensure the fair splitting of resources among frameworks and applications. Most likely, this energy resource allocation would be consumed far before a CPU allocation was consumed if a device is energy-constrained. Unfortunately the effective metering of energy is difficult on many resource-constrained sensors. On some systems, such as Signpost, hardware is included to directly meter power and energy consumption. Metering may also become more common with approaches such as counting the energy quanta used by a common switching regulator [168, 169] Both of these methods would lend themselves well to EdgeRM, and the EdgeRM agent could be extended to advertise a maximum allowable average power draw,

then meter usage on a fix time interval, cutting off tasks that use above their allocation. This is identical to the energy allocation policy promoted by Signpost and shown in Figure 2.7, except isolation is being performed in software.

We also note, however, that energy usage is a consequence of other metered resources. A combination of CPU, networking, and sensor operations could in principle be modeled to stay within a fixed power budget. This limitation could even be placed a priori by EdgeRM if the power draw of each component could be expressed to EdgeRM and subsequently to the frameworks by EdgeRM. This approach has drawbacks however. It would require a sensor designer to deeply understand the power draw of each part of the design before the node could become part of an EdgeRM cluster. It would also required the resource manager to more deeply understand the specific resources for each node, exacerbating the naming problem described in Section 5.4. A mix of these two models could also be implemented where the sensor designer coarsely understands the power draw of each component and isolates energy usage based on other resource consumption, but does not convey these resource dependencies to EdgeRM. This mixed approach would allow energy to be effectively isolated with out metering hardware, but would require the framework to test or discover the amount of energy used by each of its tasks over time, similar to the performance profiling discussed in Chapter 6.

Chapter 6

Application Frameworks for Utility Sensing

.

While the resource manager serves as the core of the cluster, application frameworks enable users to perform tasks on and collect data from resource-constrained sensors. These frameworks define a programming model, receive programs from one or more users in this model, and parse and distribute tasks to sensor nodes, gateways, and servers that are part of the cluster to achieve the application's goal. Without effective and easy-to-use application frameworks, we it would be difficult to achieve the vision of utility sensing.

One consequence of the EdgeRM design, and particularly the decision to not perform or assist with scheduling, is that the application frameworks using EdgeRM will be quite complex. Like their cloud counterparts, they will be responsible for the scheduling of tasks to nodes with sufficient resources. Unlike their cloud counterparts, they are also burdened with the additional scheduling requirements that are specific to distributed and resource-constrained sensor networks. Optimal scheduling decisions could be made based on the location and metadata associated with a sensor, on data returned by a sensor or other sensors in the network, on the amount of energy available to nodes, on the network topology and likelihood of link failure, and on the performance of specific nodes for specific tasks, among other things.

Application frameworks also have the opportunity to perform a number of optimizations to make the applications they support more capable or efficient. For instance, a sensor query framework could combine tasks from multiple queries that request the same data. Data that is known not to impact the query need not be sampled at all. These optimization are crucial, especially under energy constraints, and many have been proposed by prior macroprogramming frameworks such as TinyDB and DFuse [121, 130]. In this chapter we start by discussing an end-to-end workflow of an example framework. We then propose solutions to enable the easier building of frameworks for EdgeRM in Section 6.2 and present example frameworks we have built using EdgeRM in Section 6.3.

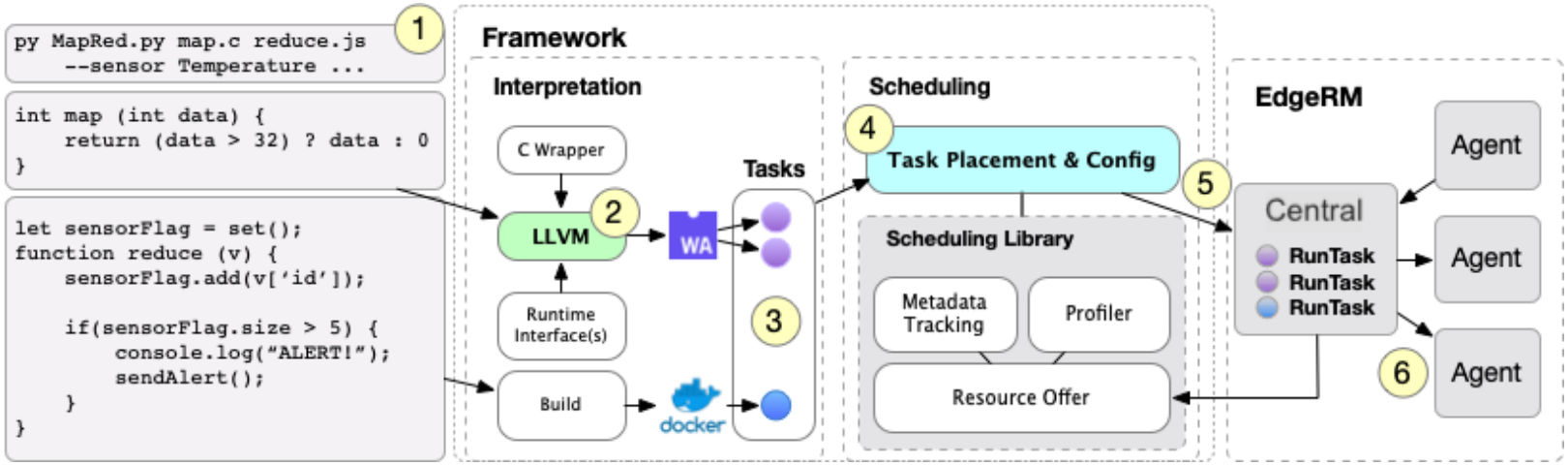


Figure 6.1: A step-by-step workflow of using EdgeRM through the Sensor MapReduce framework (§6.3). (1) A user submits map and reduce jobs to the application framework; (2) The framework's interpreter wraps user code in boilerplate communication code and compiles it into WASM modules and docker containers. (3) These tasks are sent to framework's scheduler, (4) which uses the EdgeRM scheduling library to fetch available resources, plan task placement, and configure tasks (i.e. with source and destination addresses). Active scheduling techniques such as agent profiling are used to assist placement (§6.2). (5) Tasks are issued to the EdgeRM central, (6) and forwarded to EdgeRM Agents to execute.

6.1 End-to-End Workflow

To provide a more grounded basis of discussion for the parts of an application framework, we start by walking through the steps of a simple sensor streaming MapReduce framework which sends "map" tasks to sensor nodes and then "reduces" their data on a publicly accessible cloud server that is part of the cluster. We go into deeper discussion of the framework itself in Section 6.3, and use this section primarily to illustrate the architecture of a framework. A flow diagram of this framework is presented in Figure 6.1. Because EdgeRM supports multiple frameworks, the exact details of each framework will differ, but Figure 6.1 provides one example of how a framework may function.

A framework user begins by writing *map* and *reduce* functions and issuing the MapReduce job via a command line interface. In this case, the MapReduce framework begins execution upon job submission, but future frameworks could run persistently and accept jobs over an API. After receiving a job, the framework constructs map and reduce tasks that are suitable for running in the EdgeRM cluster. The map function is inserted into a baseline map program which handles sensor sampling and results generation from the map function, and the resulting code is compiled into a WebAssembly module via clang/LLVM. The stubs for the WASM system interface are compiled into the module, and the resulting WASM module will only run in a WASM runtime with a matching system interface. Some parts of the map module, such as the destination IP address and port, are left configurable via environment variables which can be set at the time of task issuance The reduce function is wrapped with baseline communication code to receive data from the map function, sort the reduce keys, execute the map function, and host results that can be fetched by the framework. It is then built into a small container. Currently this framework can parse map programs written in C, and reduce programs written in JavaScript, but this is not a requirement.

After the final container and WASM module are built, they are sent to the MapReduce scheduler. The scheduler uses an assistive library which we present in Section 6.2 to interact with EdgeRM. The scheduler starts by requesting offers from EdgeRM and filtering those offers for resources that meet the required tasks. First the scheduler searches for for a publicly-accessible node that can run a container for the reduce task. When the scheduler receives an offer for such a node, it issues this task requesting a fixed amount of memory, CPU, and a network port, waits for confirmation from EdgeRM that the task is running, and notes its address and port. The scheduler then continuously requests offers from EdgeRM for nodes that have the specified sensor resource, can support the specified sample rate, and have sufficient memory to execute the map function. Upon receiving a sufficient offer, it issues the map task to the sensor with environment variables specifying the address and port of the reduce container. The MapReduce framework continuously monitors the cluster for new nodes with sufficient resources for the map task, restarting and re-issuing tasks that fail. The framework also queries the reduce container for results, presenting them to the user. We note that no data from the MapReduce framework flows through EdgeRM, EdgeRM only facilitates the scheduling and execution of tasks and the allocation of resources such that the MapReduce framework can setup independent data flows.

6.2 Framework Components and Meta-Frameworks

Because the building of frameworks and schedulers for distributed and resource-constrained context could be burdensome, we seek to provide tools and guidance for the components of an application framework that may be common or generally useful across frameworks. These components aim to provide support for the more difficult parts of using EdgeRM and writing applications for distributed sensors including handling heterogeneity between devices in the cluster and collecting information useful for scheduling beyond simple resource sufficiency. We also provide practical support for interacting with EdgeRM such that framework builders do not need to re-implement basic interaction flows.

Framework Support Library

We start with a framework support library. This currently exists as a Python library, but could be extended to other languages. The library assists with requesting resource offers, filtering offers based on task requirements, and issuing one or more task execution requests on the provided resource offer. The library also helps frameworks monitor and collect information on already running or recently stopped tasks, including error or termination statuses. Because frameworks exists as clients in a client-server model, the library can be run from anywhere that can access the EdgeRM central for the cluster.

Active Scheduling

As mentioned in Section 5.1 scheduling decisions in edge environments are often not strictly made due to resource availability, but also due to a devices' physical location, the network topology, events sensed by other devices, and many other possible optimizations. Additionally, in a cluster of heterogeneous devices, the capabilities of a unit of resources are necessarily also heterogeneous (i.e. 1 microcontroller CPU is not equivalent to 1 server CPU, and network links have very different capacity and delay).

To address this problem we propose *active scheduling*. Active scheduling is the process of a framework scheduling meta-tasks that are not designed directly to serve any one application, but which can be used to inform the scheduling decisions for all applications sent to the framework. By scheduling meta-tasks, frameworks can discover dynamic information about nodes and their environment without the requiring that information be actively advertised by nodes or collected by EdgeRM. Placing this burden on the framework rather than EdgeRM is architecturally advantageous because EdgeRM could never enumerate the full set of information that is useful for all schedulers, and the preemptive collection of such information by nodes could use unnecessary resources. The downsides of pushing this task onto the frameworks are framework complexity, and the inability to share information collected through active scheduling between frameworks, although we discuss a potential solution to both of these issues below. There are several high-level classes of active scheduling that we have, and an overview of these use cases is presented in Figure 6.2.

Context Monitoring

A framework may only want to schedule a task on a node in specific scenarios. For mobile nodes, this could be when they enter a specific location, or for very rare events, a framework may only schedule a task after the event has happened occurred. Frameworks may also collect information about the underlying distribution of sensor data to understand which nodes are contributing most to the data stream and which can be excluded without greatly impacting the results. These scheduling scenarios can be achieved by deploying small active scheduling tasks to monitor or query the context of nodes in the network without deploying the full task on all nodes. This optimization would allow frameworks to save valuable resources, by waiting to deploy a full task until the condition or context for that task has been met. Context monitoring tasks may also be shared among multiple applications served by a framework to further save resources.

Networking and Failure Domains

One key piece of scheduling information in distributed sensor networks is the network topology and the potential failure models of a network. This information is necessary to get some of the benefits of local computation discussed in Section 3.3. While some of this information could be annotated in the attributes or metadata of static devices using a home or network name, a more reliable and more dynamic approach would be to automatically discover the network topology. A framework could deploy an active scheduling task to perform a scan of the local network, or test the pairwise reachability and network routes between nodes to ensure adequate connectivity for the task at hand.

Performance Profiling

To help handle device heterogeneity, we propose the use of active scheduling to probe the performance and resource utilization of running specific tasks on a specific node. This could be used to discover and update the capacity of a network link, the speed of a processor, or the energy utilization of sampling a specific sensor. Especially as the amount of local computation grows, an understanding of the latency or energy of running a specific task, such as a machine learning model, will be critical to knowing not only if the task can meet application demands, but also whether a task can reasonably be run within resource constraints. While a framework can analyze the number of instructions necessary to execute a specific code path, it will not know if those instructions can be executed within a given CPU allocation without some notion of a CPUs performance. Active scheduling enables frameworks to solve this problem independently of EdgeRM.

Metadata

The last type of information that frameworks and their schedulers need to make scheduling decisions, and that applications need to perform processing and aggregation tasks, is metadata.

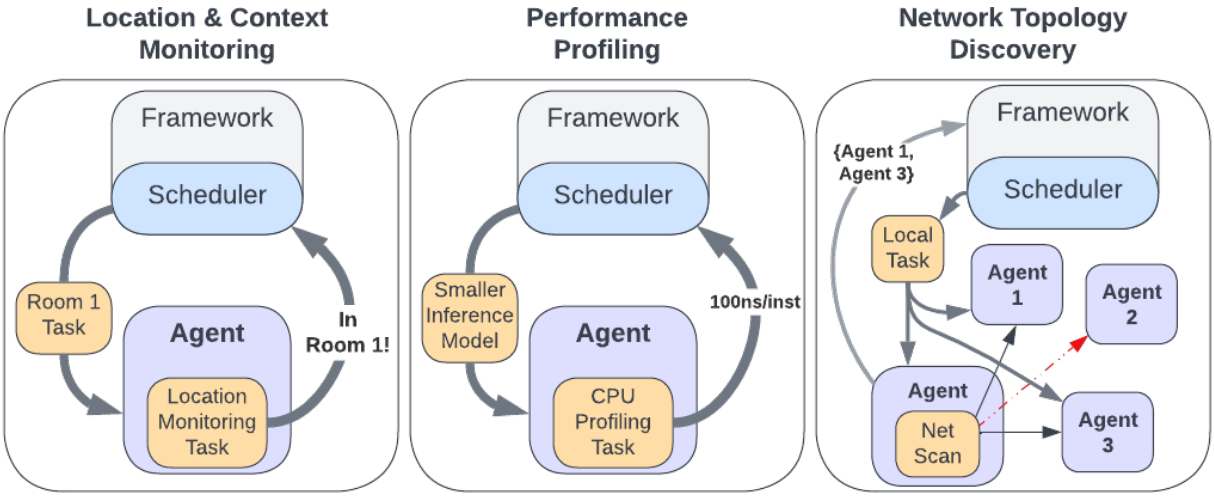


Figure 6.2: Use cases for active scheduling in EdgeRM. (Left) Location and context monitoring tasks can be used to optimize the deployment of larger or more resource intensive tasks pausing their deployment until a condition is met such as the location of a device changing or a sensor returning a specific data value. (Center) Performance profiling enables to frameworks to measure processor performance, network throughput, or other dynamic agent qualities. This allows for schedulers to understand the relative performance of nodes in the case of great node heterogeneity. (Right) Network topology detection can help schedulers place tasks within local networks such that they keep operating in cases of wide-area network failure or to preserve the privacy of local data. All active scheduling allows schedulers to adapt to the resource-constrained sensing context by allowing them to collect information necessary to assist with scheduling that cannot be known or is difficult to annotate at the time of deployment.

Metadata exists in the space between information that could be annotated in attributes or collected by active scheduling. Often metadata is created at the time of sensor deployment, such as noting the home, room, location or other information about the data that a sensor is collecting. Because EdgeRM only manages resources, and because metadata may not be known by the device itself, frameworks are responsible for collecting it, making scheduling decisions based on it, and merging it with sensor data streams as appropriate.

While not prescriptive as a part of EdgeRM, we propose using EdgeRM device attributes as a layer of indirection for metadata storage that is external. Specifically, at the time of programming, a device could be configured with an external metadata storage location and format. Upon deployment, this metadata storage could be updated with the device's information. We implement this layer of indirection using a JDBC URI [170], which specifies the location and credentials of metadata about a sensor. A framework can then collect this metadata for the specified device ID and use that information for scheduling or for merging with application data. This metadata system is implemented in the the framework

support library described in Section 6.2. We make no attempt to specify the type or format of metadata as many studies have been undertaken about metadata standardization [171]. However we note that some standardization will be necessary for frameworks to properly interpret the metadata, even if a layer of indirection is in place.

Long-running Tasks and Orchestration

In most example frameworks built for EdgeRM we find that some tasks are long-running (i.e. servers, databases, messaging brokers) and some tasks are more ephemeral (sensing, filtering, processing). As many other projects such as Kubernetes and the work leading to its creation have discovered, long running tasks have their own needs for monitoring, restarting upon failure, auto-scaling, and log aggregation. We do not hope that it is necessary that framework builders recreate these features to support what has become a common application paradigm.

To solve this problem we propose that either as part of the framework support library, or as a separate framework, frameworks can launch long-running tasks which are monitored and restarted as necessary. We have considered implementing some subset of the Kubernetes API specification to support the notion of long-running deployments. To effectively monitor these tasks, however, either the framework itself or a separate monitoring process or framework would need to be similarly reliable. This pushes us to consider the idea of several core meta-frameworks, both to implement long-running tasks as well as other common components among application frameworks.

Meta-Frameworks

There is a decision to be made between allowing all of these components to exist as part of a library that can be imported and used by a specific framework, or as a standalone program, executed as a meta-framework with tasks submitted to it by other application frameworks. The former architecture is advantageous for fairness of resource allocation, as resources are more easily attributable to the executing framework. However, as meta-frameworks, these services could more easily distribute the resource burden of active scheduling across any framework using the service, and they could be designed to be long-running such that the reliability burden of application frameworks is lessened. As meta-frameworks, these services may be architecturally separate from EdgeRM, but become de facto components of an EdgeRM deployment. Currently many of these services exist as part of the framework support library but in the future they may be deployed as standalone programs alongside EdgeRM; this separation between components still gives frameworks the freedom to deploy their own tasks to assist with scheduling and monitoring.

6.3 Framework Implementations

We implement several application frameworks which use EdgeRM to program clusters of resource-constrained sensors, local gateways, and cloud servers. These frameworks use the framework support library as well as several of the components described in Section 6.2. While these frameworks are not being proposed as ideal programming models for distributed, resource-constrained sensor networks, they demonstrate key abstractions of utility sensing, including multiple users sharing the resources of the cluster to perform distinct tasks. We implement a simple sensor sampling and filtering framework, the MapReduce framework described in Section 6.1, a basic SQL framework, and we port an existing edge computing framework, DDFLow to use EdgeRM.

Sensor Sampling and Filtering Framework

The sensor sampling framework allows users to specify the sensor to sample, the sampling rate, and several optional filters, and then issues tasks to sample the sensor for several minutes before exiting. It consists of a container which runs a CoAP server to receive and host client-specific sensor results and a WebAssembly task which samples, filters, and forwards data to the CoAP server. If the selected sensor is a camera, an optional argument allows for image classification via a Docker container implementation using YOLOv3 [172].

A visual web interface provides users with a view of all devices and tasks within the EdgeRM cluster, allowing them to select a sensor and issue a request. The framework is written statelessly; upon user request, the framework searches for an existing CoAP server container, and, if one is not already executing, starts one on a publicly-accessible node. Once these infrastructure tasks are running, the framework searches for the device-of-interest within the resource offer. The sensor fetch task is then issued using EdgeRM with environment variables to specify the sensor, sample rate, filters, and CoAP server address. This generic framework has been used by dozens of users simultaneously.

This framework demonstrates both the multiprogramming of resource-constrained sensors and the ability of a single framework to collect and serve data to multiple distinct clients. It also demonstrates the network profiling component by attempting to schedule the image classification locally if the camera sensor is chosen.

Streaming SQL

As an extension of the sensor sampling and filtering framework, we implement a very simple streaming SQL front-end that supports SELECT and WHERE statements. Rather than deploying a query parser on the sensor as with TinyDB, the framework parses these SQL statements and translates them into the same tasks used in the sensor sample and filter framework. This framework could be extended to support join operations and many of the same optimization described in TinyDB [121]. This was possible and easy to implement

because the WASM runtime and its underlying virtualization layer is enforcing isolation, rather than a local SQL query parser.

Sensor MapReduce

As initially described in Section 6.1 and illustrated in Figure 6.1, we have developed a stream MapReduce framework targeting resource-constrained sensors. Users can develop *map* functions which can be deployed on all sensors or specific sensors based on metadata filters. The output of these map functions is forwarded to a reduce function which can aggregate and publish the resulting data. A utility also displays the results of the reduce function on the user's terminal.

Map tasks are compiled and issued as WASM modules, while reducers are issued via Docker containers. The MapReduce framework continuously monitors the cluster to adjust deployed tasks based on the available resources, such as a newly registered sensor device, with support for killing and/or re-issuing MapReduce jobs. The framework leverages Active Scheduling principles incorporated into the EdgeRM scheduling library, including network profiling information, to schedule Reduce tasks to the nearest available and accessible server endpoint.

Porting an Edge Computing Framework

Retargeting existing edge computing systems to EdgeRM enables immediate benefits including isolation and multi-framework tenancy. With some modifications, retargeting may even be able to support the extension of programming frameworks for gateway-class machines to include resource-constrained sensors. To this end, we ported the DDFlow system to EdgeRM to validate the practicality and ease of doing so [136].

DDFlow allows developers to declaratively construct edge computing applications as directed dataflow graphs of microservice tasks to be distributed across a computing cluster. As is commonly the case in this setting, the intended target devices are gateways, wireless access points, and other near-edge devices. Porting the (1) client interface and (2) device webserver to a container allows deployment using the EdgeRM abstraction over the same suite of devices. This method of porting allowed much of DDFlow to remain untouched including its original scheduler because multiple DDFlow tasks where deployed with its container (a single EdgeRM task). The full porting effort was accomplished in 279 lines of code and about a day of development effort.

Given the natural decomposition of distributed applications into containers [137, 138, 161], porting edge computing systems to EdgeRM is a straightforward process. Given the growth of languages which can be compiled into WASM modules, we also hope that many of these applications can be easily extended to resource-constrained sensors by compiling their existing tasks into WASM modules and issuing those tasks to nodes that only support the WASM runtime.

6.4 Reflection on Building and Using Application Frameworks for EdgeRM

We did not perform user studies on the ease of programming resource constrained sensors using these application frameworks, nor did we have multiple sets of developers port their programming frameworks to or develop new programming frameworks for EdgeRM. In lieu of this attempt at quantitative evidence, we provide our perspective on the experience of both building frameworks for EdgeRM and using these frameworks to collect sensor data.

The first and most bold conclusion of our experience is that EdgeRM significantly eased the deployment of code to resource-constrained sensors. This was especially highlighted during the development of base code for the frameworks and the development of complex map tasks for the MapReduce framework. New code could be deployed confidently and without the need to check for errors because the EdgeRM agent and its WASM runtime would isolate failures. Code could be iterated on quickly because when it failed. EdgeRM reports the reason that a task failed, either by collecting the program's exit message directly or reporting an internal error such as the use of too many resources or making an invalid call to the runtime, easing rapid iteration. When the new task was deployed, it was distributed quickly to the sensor nodes because the WASM modules are small compared to a code update for the entire sensor, often fitting in one or two packets, and all sensors in the network attempt to execute the code immediately (if selected) meaning errors are caught quickly. Moreover, multiple developers could be iterating on their frameworks and deploying code to these sensors simultaneously.

In describing this experience it's somewhat difficult to pinpoint the difference between this and past methods of deploying code, however it clearly sits at the intersection of confidence, interactivity, and direct feedback. No other general-purpose programming method for sensor nodes has all of these qualities. This is in no small part due to the effort of developing, iterating on, and debugging the EdgeRM agent code itself. However, once that is built, deploying code to a cluster of sensors does start to mimic the feeling of deploying data analysis code in PySpark or infrastructure in Kubernetes. The programming models need fine tuning, but one can create working code quickly because code can be quickly iterated on with confidence; it "feels like" utility sensing.

The second conclusion is that the components describe in Section 6.2 need more iteration and development, and need higher level abstractions. Abstractions like "will this WASM module run on this sensor" and "make sure this task runs indefinitely or is rescheduled with these minimum resources until I request it is killed." While the frameworks we built work well, especially the stateless frameworks such as the SQL query parser and sensor sampling framework, and the tasks they deploy feel robust and iterable, the frameworks themselves are infrastructure for the applications, and developing reliable infrastructure is difficult. Long-running frameworks could themselves be monitored, restarted, or launch themselves in EdgeRM with a long-running state controller; basic basic schedulers for EdgeRM frameworks could be made modular so that they can be shared among multiple frameworks, only to be modified where necessary for a specific application. We will leave these improvements

to future work, along with the porting of past macroprogramming frameworks discussed in Chapter 4 to use EdgeRM. Porting these frameworks and developing new ones will be significantly easier with the existence of a reliable resource management abstraction that works under resource-constraints.

Chapter 7

Conclusion

We began this work with the goal of making distributed, resource-constrained sensor networks easier to deploy and program by people who were steeped in the art and science of embedded systems engineering. This led us to create Signpost, which lowered the barrier to building and deploying new sensors, but did not alleviate the burden of writing sensing applications or collecting data from the sensors themselves. To fully enable the vision of easy-to-use sensing, the sensors themselves, and not just the hardware platform they rely on, needs to be robust infrastructure suitable for sharing.

Between the development of Signpost and the writing of this dissertation, several key technology advancements have enabled the creation of such a robust platform and abstraction. The low-power processors found on most of these sensors have become larger, faster, and more efficient, and they are now on par with with the processors that ran the mainframes used to build the first multiprogramming operating systems. Technologies such as WASM and hardware memory protection enabled multiprogramming and more robust resource isolation, even under severe memory constraints. Using these technologies, we are now able to organize resource-constrained sensors into a cluster of sensor resources that can be shared and programmed just like modern cloud computing clusters. By adopting the techniques of past programming models for sensors, we are able to build application frameworks specifically suited to programming clusters of sensors.

Still, there is work to be done to ensure the full vision is realized. Isolation mechanisms for low-power processors and new operating systems and runtimes which leverage them will continue to make sensor multiprogramming more efficient. Standardized sensor access interfaces and metadata naming will be required for distributed sensor applications to be as portable as containers in the cloud. Already-common data processing systems and the languages they use should be extended to fluidly incorporate sensors into their existing computing clusters. We expect that ongoing research will address some of these problems in time, and engineering effort will be necessary to integrate these solutions and make the systems proposed here usable by a broader audience.

While programming a sensor cluster is not as easy as writing a few lines of data analysis in a high-level dataflow language, this resource management abstraction provides a basis on

which such programming frameworks can be quickly developed, evaluated, and improved. It allows the sensors to be used as a shared utility, and we believe this is a necessary step towards giving future city planners, environmental scientists, and economists direct and efficient access to data which can help them discover and adapt to our ever-changing world.

Bibliography

[1] United Nations Department of Economic and Social Affairs. *World Urbanization Prospects: The 2014 Revision*. 2014.

[2] National Science Foundation. *NSF commits more than $60 million to Smart Cities Initiative*. 2016.

[3] European Commission. *Using EU funding mechanism for Smart Cities*. 2013.

[4] China Business Review. *Smart City Development in China*. 2014.

[5] City of Columbus. *Smart Columbus*. https://www.columbus.gov/smartcolumbus/. 2017.

[6] City of Denver. *Denver – A Smart City*. http://www.denvergov.org/content/denvergov/en/transportation-mobility/smart-city.html. 2017.

[7] Yongling Li, Yanliu Lin, and Stan Geertman. "The development of smart cities in China". In: *The 14th International Conference on Computers in Urban Planning and Urban Management, Cambridge, MA USA*. CUPUM'15. 2015.

[8] Charlie Mydlarz, Justin Salamon, and Juan Pablo Bello. "The implementation of low-cost urban acoustic monitoring devices". In: *Applied Acoustics* 117 (2017).

[9] Charles E Catlett et al. "Array of things: a scientific research instrument in the public way: platform design and early lessons learned". In: *Proceedings of the 2nd International Workshop on Science of Smart City Operations and Platforms Engineering*. SCOPE'17. ACM. 2017.

[10] Yun Cheng et al. "AirCloud: a cloud-based air-quality monitoring system for everyone". In: *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*. SenSys'14. ACM. 2014.

[11] A. Ledeczi et al. "Multiple simultaneous acoustic source localization in urban terrain". In: *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.* IPSN'05. 2005.

[12] Lewis Girod et al. "The Design and Implementation of a Self-calibrating Distributed Acoustic Sensing Platform". In: *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*. SenSys'06. ACM, 2006.

[13] Kai Li, Chau Yuen, and Salil Kanhere. "SenseFlow: An Experimental Study of People Tracking". In: *Proceedings of the 6th ACM Workshop on Real World Wireless Sensor Networks*. RealWSN'15. ACM, 2015.

[14] Omid Abari et al. "Caraoke: An e-toll transponder network for smart cities". In: *ACM SIGCOMM Computer Communication Review*. Vol. 45. SIGCOMM'15. ACM. 2015.

[15] Eric Bouillet et al. "Fusing Traffic Sensor Data for Real-time Road Conditions". In: *Proceedings of First International Workshop on Sensing and Big Data Mining*. SENSEMINE'13. ACM, 2013.

[16] Rijurekha Sen et al. "Kyun Queue: A Sensor Network System to Monitor Road Traffic Queues". In: *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*. SenSys'12. ACM, 2012.

[17] Ian Rose and Matt Welsh. "Mapping the Urban Wireless Landscape with Argos". In: *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*. SenSys'10. 2010.

[18] Nageswara SV Rao et al. "Identification of low-level point radiation sources using a sensor network". In: *Proceedings of the 7th international conference on Information processing in sensor networks*. IPSN'08. IEEE Computer Society. 2008.

[19] Jeffrey B Basara et al. "Overview of the Oklahoma City Micronet". In: *Eighth Symposium on the Urban Environment*. 8URBAN. 2009.

[20] Bradley G Illston et al. "Design and deployment of traffic signal stations within the Oklahoma City Micronet". In: *Eighth Symposium on the Urban Environment*. 8URBAN. 2009.

[21] Peter Ney et al. "SeaGlass: Enabling City-Wide IMSI-Catcher Detection". In: *Proceedings on Privacy Enhancing Technologies*. PoPETs'17 2017 (2017).

[22] Jeffrey A Burke et al. "Participatory sensing". In: WSW'06. 2006.

[23] Andrew T Campbell et al. "People-centric urban sensing". In: *Proceedings of the 2nd annual international workshop on Wireless internet*. WICON'06. ACM. 2006.

[24] Uichin Lee et al. "Efficient data harvesting in mobile sensor platforms". In: *Pervasive Computing and Communications Workshops, 2006. PerCom Workshops 2006. Fourth Annual IEEE International Conference on*. PerCom'06. IEEE. 2006.

[25] Bret Hull et al. "CarTel: a distributed mobile sensor computing system". In: *Proceedings of the 4th international conference on Embedded networked sensor systems*. SenSys'06. ACM. 2006.

[26] Elizabeth Bales et al. "Citisense: Mobile air quality sensing for individuals and communities design and deployment of the citisense mobile air-quality system". In: *Pervasive Computing Technologies for Healthcare (PervasiveHealth), 2012 6th International Conference on*. PervasiveHealth'12. IEEE. 2012.

[27] Prabal Dutta et al. "Common sense: Participatory urban sensing using a network of handheld air quality monitors". In: *Proceedings of the 7th ACM conference on embedded networked sensor systems*. SenSys'09. ACM. 2009.

[28] Srinivas Devarakonda et al. "Real-time air quality monitoring through mobile sensing in metropolitan areas". In: *Proceedings of the 2nd ACM SIGKDD international workshop on urban computing*. SIGKDD'15. ACM. 2013.

[29] Rajib Kumar Rana et al. "Ear-phone: An end-to-end participatory urban noise mapping system". In: *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. IPSN'10. ACM. 2010.

[30] Nicolas Maisonneuve et al. "NoiseTube: Measuring and mapping noise pollution with mobile phones". In: *Information technologies in environmental engineering*. ITEE'09 (2009).

[31] Mark Bilandzic et al. "Laermometer: A mobile noise mapping application". In: *Proceedings of the 5th Nordic conference on Human-computer interaction: building bridges*. NordiCHI'08. ACM. 2008.

[32] Prashanth Mohan, Venkata N Padmanabhan, and Ramachandran Ramjee. "Nericell: rich monitoring of road and traffic conditions using mobile smartphones". In: *Proceedings of the 6th ACM conference on Embedded network sensor systems*. SenSys'08. ACM. 2008.

[33] Arvind Thiagarajan et al. "VTrack: accurate, energy-aware road traffic delay estimation using mobile phones". In: *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*. SenSys'09. ACM. 2009.

[34] Daniel B Work et al. "Mobile Millennium Demonstration-Participatory Traffic Estimation Using Mobile Phones". In: (2009).

[35] Uichin Lee et al. "Mobeyes: smart mobs for urban monitoring with a vehicular sensor network". In: *IEEE Wireless Communications* 13 (2006).

[36] Suhas Mathur et al. "Parknet: drive-by sensing of road-side parking statistics". In: *Proceedings of the 8th international conference on Mobile systems, applications, and services*. MobiSys'10. ACM. 2010.

[37] Jakob Eriksson et al. "The pothole patrol: using a mobile sensor network for road surface monitoring". In: *Proceedings of the 6th international conference on Mobile systems, applications, and services*. MobiSys'08. ACM. 2008.

[38] Rohan Narayana Murty et al. "Citysense: An urban-scale wireless sensor network and testbed". In: *Technologies for Homeland Security, 2008 IEEE Conference on*. HST'08. IEEE. 2008.

[39] Matt Welsh. Personal communication. Apr. 2016.

[40] *MSP430F15x, MSP430F16x, MSP430F161x MIXED SIGNAL MICROCONTROLLER*. MSP430F16. Texas Instruments. Mar. 2011. URL: https://www.ti.com/lit/ds/symlink/msp430f1611.pdf.

[41] *Apollo MCU Datasheet*. Apollo. Rev. 1.00. Ambiq Micro. Mar. 2017. URL: https://contentportal.ambiq.com/documents/20123/388400/Apollo-Datasheet.pdf.

[42] *Low Power 2.4 GHz Transceiver for ZigBee, IEEE 802.15.4, 6LoWPAN, RF4CE and ISM Applications*. AT86RF230. Atmel Corporation. Feb. 2009. URL: http://ww1.microchip.com/downloads/en/devicedoc/doc5131.pdf.

[43] *nRF51822 Product Specification*. nRF51822. Rev. 3.4. Nordic Semiconductor. Jan. 2016. URL: https://infocenter.nordicsemi.com/pdf/nRF51822_PS_v3.4.pdf.

[44] Goojin Jeong et al. "Prospective materials and applications for Li secondary batteries". In: *Energy & Environmental Science* 4 (2011).

[45] NREL. *Solar Maps*. http://www.nrel.gov/gis/solar.html. July 2016.

[46] LoRa Alliance. *LoRa Alliance*. https://www.lora-alliance.org/. 2017.

[47] Intel. *Intel Edison Compute Module*. Edison. Rev. 004. Jan. 2015.

[48] Bharath Sundararaman, Ugo Buy, and Ajay D Kshemkalyani. "Clock synchronization for wireless sensor networks: a survey". In: *Ad hoc networks* (2005).

[49] Joshua Adkins et al. "Energy Isolation Required for Multi-tenant Energy Harvesting Platforms". In: *Proceedings of the Fifth ACM International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*. ENSsys'17. ACM. 2017.

[50] Federal Highway Administration. *Manual on uniform traffic control devices*. 2009.

[51] U-blox. *SARA-U2 Series Datasheet*. Rev. R14. Dec. 2016.

[52] Multitech. *MultiConnect xDot*. Sept. 2017.

[53] Amit Levy et al. "Multiprogramming a 64kB Computer Safely and Efficiently". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Association for Computing Machinery, 2017.

[54] Arduino. *Arduino website*. https://www.arduino.cc. 2017.

[55] Arm. *Mbed OS developer website*. https://os.mbed.com. 2017.

[56] Benjamin Y.H. Liu and Richard C. Jordan. "The long-term average performance of flat-plate solar-energy collectors: With design data for the U.S., its outlying possessions and Canada". In: *Solar Energy* (1963). URL: http://www.sciencedirect.com/science/article/pii/0038092X63900069.

[57] S. Wilcox. *National Solar Radiation Database 1991-2005 Update: User's Manual*. Tech. rep. National Renewable Energy Laboratory, Apr. 2007. URL: https://www.nrel.gov/docs/fy07osti/41364.pdf.

[58] Norman Abramson. "THE ALOHA SYSTEM: Another Alternative for Computer Communications". In: *Proceedings of the November 17-19, 1970, Fall Joint Computer Conference*. AFIPS'70 (Fall). ACM, 1970.

[59] Joseph Polastre, Robert Szewczyk, and David Culler. "Telos: enabling ultra-low power wireless research". In: *Proceedings of the 4th international symposium on Information processing in sensor networks*. IPSN'05. IEEE Press. 2005.

[60] Alan Mainwaring et al. "Wireless sensor networks for habitat monitoring". In: *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*. WSNA'02. Acm. 2002.

[61] *Cortex M4 Technical Reference Manual*. ARM Limited. Mar. 2010. URL: https://developer.arm.com/documentation/100166/0001.

[62] *Arm Ethos-U Processor Series*. ARM Limited. Oct. 2020. URL: https://armkeil.blob.core.windows.net/developer/Files/pdf/arm-ethos-u-processor-series-brief-v2.pdf.

[63] *Artificial Intelligence Microcontroller with Ultra-Low-Power Convolutional Neural Network Accelerator*. Maxim Integrated. May 2021. URL: https://datasheets.maximintegrated.com/en/ds/MAX78000.pdf.

[64] Emil Björnson and Erik G. Larsson. "How Energy-Efficient Can a Wireless Communication System Become?" In: *2018 52nd Asilomar Conference on Signals, Systems, and Computers*. 2018.

[65] *Multi-Core Bluetooth® 5.2 SoC Family with System PMU*. DA1469x. Rev. 3.3. Dialog Semiconductor. Apr. 2022. URL: https://www.renesas.com/us/en/document/dst/da1469x-datasheet?r=1564821.

[66] *nRF52840 Product Specification*. nRF52840. Rev. 1.7. Nordic Semiconductor. Nov. 2021. URL: https://infocenter.nordicsemi.com/pdf/nRF52840_PS_v1.7.pdf.

[67] *Apollo4 Blue SoC*. Apollo4 Blue. Rev. 1.2.0. Ambiq Micro. Aug. 2022. URL: https://contentportal.ambiq.com/documents/20123/388405/Apollo4-Blue-SoC-Datasheet.pdf.

[68] *CC2652R SimpleLink Multiprotocol 2.4 GHz Wireless MCU*. CC2652. Texas Instruments. Mar. 2021. URL: https://www.ti.com/lit/ds/symlink/cc2652r.pdf.

[69] Ubilite. *UBI206 Specifications*. https://www.ubilite.com/product/. 2022.

[70] *ESP32 Series Datasheet*. ESP32. Rev. 4.0. Espressif Systems. 2022. URL: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[71] *Long Range, Low Power, sub-GHz RF Transceive*. SX1262. Rev. 2.1. Semtech. Dec. 2021. URL: https://semtech.my.salesforce.com/sfc/p/#E0000000JelG/a/2R000000Un7F/yT.fKdAr9ZAo3cJLc4F2cBdUsMftpT2vsOICP7NmvMo.

[72] *SX1276/77/78/79 - 137 MHz to 1020 MHz Low Power Long Range Transceiver*. SX1276. Rev. 7. Semtech. May 2020. URL: https://semtech.my.salesforce.com/sfc/p/#E0000000JelG/a/2R0000001Rbr/6EfVZUorrpoKFfvaF_Fkpgp5kzjiNyiAbqcpqh9qSjE.

[73] Le Wang and Jukka Manner. "Energy Consumption Analysis of WLAN, 2G and 3G interfaces". In: *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*. 2010.

[74] Junxian Huang et al. "A Close Examination of Performance and Power Characteristics of 4G LTE Networks". In: *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*. MobiSys '12. 2012.

[75] Robert Falkenberg, Benjamin Sliwa, and Christian Wietfeld. "Rushing Full Speed with LTE-Advanced Is Economical - A Power Consumption Analysis". In: *2017 IEEE 85th Vehicular Technology Conference (VTC Spring)*. 2017.

[76] *nRF9160 Product Specification*. nRF9160. Rev. 2.1. Nordic Semiconductor. Oct. 2021. URL: https://infocenter.nordicsemi.com/pdf/nRF9160_PS_v2.1.pdf.

[77] *SARA-R5 series LTE-M / NB-IoT modules with secure cloud*. SARA-R5. ublox. July 2022. URL: https://content.u-blox.com/sites/default/files/SARA-R5_DataSheet_UBX-19016638.pdf.

[78] M. Hata. "Empirical formula for propagation loss in land mobile radio services". In: *IEEE Transactions on Vehicular Technology* (1980).

[79] Ambuj Varshney, Wenqing Yan, and Prabal Dutta. "Judo: Addressing the Energy Asymmetry of Wireless Embedded Systems through Tunnel Diode Based Wireless Transmitters". In: *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. MobiSys '22. Association for Computing Machinery, 2022.

[80] Pengyu Zhang et al. "EkhoNet: High-Speed Ultra Low-Power Backscatter for Next Generation Sensors". In: 19 (2015).

[81] Carlos Pérez-Penichet et al. "Battery-Free 802.15.4 Receiver". In: *Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks*. IPSN '18. IEEE Press, 2018.

[82] Dinesh Bharadia et al. "BackFi: High Throughput WiFi Backscatter". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. Association for Computing Machinery, 2015.

[83] Bryce Kellogg et al. "PASSIVE WI-FI: Bringing Low Power to Wi-Fi Transmissions". In: *GetMobile: Mobile Comp. and Comm.* 20 (2017).

[84] Bryce Kellogg et al. "Wi-Fi Backscatter: Internet Connectivity for RF-Powered Devices". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Association for Computing Machinery, 2014.

[85] Vamsi Talla et al. "LoRa Backscatter: Enabling The Vision of Ubiquitous Connectivity". In: *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 1 (2017).

[86] Pengyu Zhang and Deepak Ganesan. "Enabling Bit-by-Bit Backscatter Communication in Severe Energy Harvesting Environments". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. USENIX Association, 2014.

[87] Ambuj Varshney, Andreas Soleiman, and Thiemo Voigt. "TunnelScatter: Low Power Communication for Sensor Tags Using Tunnel Diodes". In: *The 25th Annual International Conference on Mobile Computing and Networking*. MobiCom '19. Association for Computing Machinery, 2019.

[88] Bo Zhai et al. "Energy-efficient subthreshold processor design". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17 (2009).

[89] *Apollo and Apollo Blue System on Chips*. Ambiq Micro. 2022. URL: https://ambiq.com/wp-content/uploads/2020/10/Apollo-SoC-Family-Brochure.pdf.

[90] Aakanksha Chowdhery et al. "Visual Wake Words Dataset". In: *CoRR* abs/1906.05721 (2019).

[91] Robert David et al. "TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems". In: *CoRR* abs/2010.08678 (2020).

[92] *The DSP capabilities of ARM Cortex-M4 and Cortex-M7 Processors*. ARM Limited. Nov. 2016. URL: https://community.arm.com/cfs-file/__key/communityserver-blogs-components-weblogfiles/00-00-00-21-42/7563.ARM-white-paper-_2D00_-DSP-capabilities-of-Cortex_2D00_M4-and-Cortex_2D00_M7.pdf.

[93] M. Claypool et al. "Network Requirements for 3D Flying in a Zoomable Brain Database". In: (1998).

[94] Yundong Zhang et al. "Hello Edge: Keyword Spotting on Microcontrollers". In: *CoRR* abs/1711.07128 (2017).

[95] Tess Despres et al. "Where the Sidewalk Ends: Privacy of Opportunistic Backhaul". In: *Proceedings of the 15th European Workshop on Systems Security*. EuroSec '22. Association for Computing Machinery, 2022.

[96] Russell Ford, Changkyu Kim, and Sundeep Rangan. "Opportunistic third-party backhaul for cellular wireless networks". In: *2013 Asilomar Conference on Signals, Systems and Computers*. 2013.

[97] Amir Haleem et al. "Helium A Decentralized Wireless Network". In: (2018). URL: http://whitepaper.helium.com/.

[98] Google. *Google Fi Plans*. https://fi.google.com/about/plans/?pli=1. 2022.

[99] Hologram. *Hologram Pricing*. https://www.hologram.io/pricing/. 2022.

[100] Gierad Laput, Yang Zhang, and Chris Harrison. "Synthetic sensors: Towards general-purpose sensing". In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. 2017.

[101] Jelena Čulić Gambiroža et al. "Predicting Low-Cost Gas Sensor Readings From Transients Using Long Short-Term Memory Neural Networks". In: *IEEE Internet of Things Journal* (2020).

[102] Qiuqi Zhang et al. "Fast Measurement With Chemical Sensors Based on Sliding Window Sampling and Mixed-Feature Extraction". In: *IEEE Sensors Journal* 20 (2020).

[103] Javier Burgués and Santiago Marco. "Wind-Independent Estimation of Gas Source Distance From Transient Features of Metal Oxide Sensor Signals". In: *IEEE Access* 7 (2019).

[104] Yuxun Zhou et al. "Abnormal event detection with high resolution micro-PMU data". In: *2016 Power Systems Computation Conference (PSCC)*. 2016.

[105] Armin Aligholian et al. "Event Detection in Micro-PMU Data: A Generative Adversarial Network Scoring Method". In: *2020 IEEE Power & Energy Society General Meeting (PESGM)*. IEEE, 2020.

[106] Pinyarash Pinyoanuntapong et al. *EdgeML: Towards Network-Accelerated Federated Learning over Wireless Edge*. 2021.

[107] Solmaz Niknam, Harpreet S. Dhillon, and Jeffery H. Reed. *Federated Learning for Wireless Communications: Motivation, Opportunities and Challenges*. 2019.

[108] Zachary S Bischof, Fabian E Bustamante, and Nick Feamster. "Characterizing and improving the reliability of broadband internet access". In: *arXiv preprint arXiv:1709.09349* (2017).

[109] Weijia He et al. "When Smart Devices Are Stupid: Negative Experiences Using Home Smart Devices". In: 2019.

[110] Karissa Bell. "What to do when internet outages ruin your cool smart home". In: (2017).

[111] Matei Zaharia et al. "Apache Spark: A Unified Engine for Big Data Processing". In: *Commun. ACM* (2016).

[112] Serena Zheng et al. "User Perceptions of Smart Home IoT Privacy". In: *Proc. ACM Hum.-Comput. Interact.* CSCW (2018).

[113] Julie Haney, Susanne Furman, and Yasemin Acar. *Research Report: User Perceptions of Smart Home Privacy and Security*. 2020.

[114] Edgar Gabriel et al. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation". In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*. 2004.

[115] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux journal* 2014 (2014).

[116] Cheol-Ho Hong and Blesson Varghese. "Resource management in fog/edge computing: a survey on architectures, infrastructure, and algorithms". In: *ACM Computing Surveys (CSUR)* 52 (2019).

[117] Bytecode Alliance. *WASM Micro Runtime*. https://github.com/bytecodealliance/wasm-micro-runtime. 2020.

[118] Philip Levis and David Culler. "Maté: A Tiny Virtual Machine for Sensor Networks". In: *Architectural Support for Programming Languages and Operating Systems*. 2002.

[119] Niels Brouwers, Peter Corke, and Koen Langendoen. "Darjeeling, a Java Compatible Virtual Machine for Microcontrollers". In: Companion '08. Association for Computing Machinery, 2008.

[120] Ryan Newton, Greg Morrisett, and Matt Welsh. "The regiment macroprogramming system". In: *2007 6th International Symposium on Information Processing in Sensor Networks*. IEEE. 2007.

[121] Samuel R Madden et al. "TinyDB: an acquisitional query processing system for sensor networks". In: *ACM Transactions on database systems (TODS)* 30 (2005).

[122] Laurynas Riliskis, James Hong, and Philip Levis. "Ravel: Programming IoT Applications as Distributed Models, Views, and Controllers". In: *Proceedings of the 2015 International Workshop on Internet of Things towards Applications*. IoT-App '15. Association for Computing Machinery, 2015.

[123] Philip Levis et al. "TinyOS: An operating system for sensor networks". In: *Ambient intelligence*. Springer, 2005.

[124] Ting Liu and Margaret Martonosi. "Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems". In: *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP* 38 (2003).

[125] Chih-Chieh Han et al. "A dynamic operating system for sensor nodes". In: *Proceedings of the 3rd international conference on Mobile systems, applications, and services*. 2005.

[126] Athanassios Boulis et al. "SensorWare: Programming sensor networks beyond code update and querying". In: *Pervasive and mobile computing* 3 (2007).

[127] Andreas Haas et al. "Bringing the Web up to Speed with WebAssembly". In: *SIGPLAN Not.* 52 (2017).

[128] Gregor Peach et al. "eWASM: Practical Software Fault Isolation for Reliable Embedded Devices". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39 (2020).

[129] Damien George. *MicroPython*. https://micropython.org/. 2020.

[130] Rajnish Kumar et al. "DFuse: A framework for distributed data fusion". In: *Proceedings of the 1st international conference on Embedded networked sensor systems*. 2003.

[131] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. "Macro-programming wireless sensor networks using Kairos". In: *International Conference on Distributed Computing in Sensor Systems*. Springer. 2005.

[132] Vinod Kumar Vavilapalli et al. "Apache Hadoop YARN: Yet Another Resource Negotiator". In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. Association for Computing Machinery, 2013.

[133] Brendan Burns et al. "Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade". In: 14 (2016).

[134] Benjamin Hindman et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center". In: NSDI'11. USENIX Association, 2011.

[135] Hypriot. *Docker Pirates ARMed with explosive stuff*. https://blog.hypriot.com/. 2020.

[136] Joseph Noor et al. "DDFlow: Visualized Declarative Programming for Heterogeneous IoT Networks". In: *Proceedings of the International Conference on Internet of Things Design and Implementation*. IoTDI '19. Montreal, Quebec, Canada: Association for Computing Machinery, 2019.

[137] Nan Wang et al. "ENORM: A framework for edge node resource management". In: *IEEE transactions on services computing* (2017).

[138] Peng Liu, Dale Willis, and Suman Banerjee. "Paradrop: Enabling lightweight multi-tenancy at the network's extreme edge". In: *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. 2016.

[139] Amazon Web Services, Inc. *AWS IoT Greengrass*. https://aws.amazon.com/greengrass/. 2020.

[140] Microsoft Azure. *Azure IoT – Internet of Things Platform*. https://azure.microsoft.com/en-us/overview/iot/. 2020.

[141] The Linux Foundation. *Zephyr OS*. https://www.zephyrproject.org/. 2020.

[142] *The FreeRTOS Refernce Manual*. Amazon Inc. 2017. URL: %5Curl%7Bhttps://www.freertos.org/fr-content-src/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf%7D.

[143] Emmanuel Baccelli et al. "RIOT OS: Towards an OS for the Internet of Things". In: *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 2013.

[144] Robert Wahbe et al. "Efficient Software-Based Fault Isolation". In: *SIGOPS Oper. Syst. Rev.* 27 (1993).

[145] Faisal Aslam et al. "Introducing TakaTuka: A Java Virtualmachine for Motes". In: *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*. SenSys '08. Association for Computing Machinery, 2008.

[146] Jose H. Solorzano. *TinyVM*. https://tinyvm.sourceforge.net/. 2000.

[147] Michael P. Andersen, Gabe Fierro, and David E. Culler. "System Design for a Synergistic, Low Power Mote/BLE Embedded Platform". In: *Proceedings of the 15th International Conference on Information Processing in Sensor Networks*. IPSN '16. IEEE Press, 2016.

[148] Dennis M. Ritchie and Ken Thompson. "The UNIX Time-Sharing System". In: *Commun. ACM* 17 (1974).

[149] Volodymyr Shymanskyy. *WASM3 Performance*. https://github.com/wasm3/wasm3/blob/master/docs/Performance.md. 2020.

[150] Abhishek Verma et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. 2015.

[151] Balbir Singh and Vaidyanathan Srinivasan. "Containers : Challenges with the memory resource controller and its performance". In: 2010.

[152] Malte Schwarzkopf et al. "Omega: flexible, scalable schedulers for large compute clusters". In: *SIGOPS European Conference on Computer Systems (EuroSys)*. 2013.

[153] Konstantin Shvachko et al. "The Hadoop Distributed File System". In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 2010.

[154] Kubernetes. *Device Plugins*. https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/.

[155] Douglas Thain, Todd Tannenbaum, and Miron Livny. "Distributed computing in practice: the Condor experience". In: *Concurrency and computation: practice and experience* 17 (2005).

[156] Mathew Ryden et al. "Nebula: Distributed edge cloud for data intensive computing". In: *2014 IEEE International Conference on Cloud Engineering*. IEEE. 2014.

[157] Karim Habak et al. "Femto clouds: Leveraging mobile devices to provide cloud service at the edge". In: *2015 IEEE 8th international conference on cloud computing*. IEEE. 2015.

[158] Cong Shi et al. "Serendipity: Enabling remote computing among intermittently connected mobile devices". In: *Proceedings of the thirteenth ACM international symposium on Mobile Ad Hoc Networking and Computing*. 2012.

[159] Eduardo Cuervo et al. "MAUI: making smartphones last longer with code offload". In: *Proceedings of the 8th international conference on Mobile systems, applications, and services*. 2010.

[160] Roberto Morabito et al. "Consolidate IoT edge computing with lightweight virtualization". In: *IEEE Network* 32 (2018).

[161] Paolo Bellavista and Alessandro Zanni. "Feasibility of fog computing deployment based on docker containerization over raspberrypi". In: *Proceedings of the 18th international conference on distributed computing and networking*. 2017.

[162] Bytecode Alliance. *WebAssembly System Interface*. https://wasi.dev/. 2020.

[163] Andreas Haas et al. "Bringing the web up to speed with WebAssembly". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017.

[164] Ali Ghodsi et al. "Dominant resource fairness: Fair allocation of multiple resource types". In: *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. 2011.

[165] Particle Industries. *Particle Platform*. https://www.particle.io/iot-platform/. 2020.

[166] Raspberry Pi Foundation. *FAQs - Raspberry Pi Documentation*. https://www.raspberrypi.org/documentation/faqs/. 2020.

[167] Wenyong Huang. *WASM Micro Runtime Performance*. https://github.com/bytecodealliance/wasm-micro-runtime/wiki/Performance. 2020.

[168] P. Dutta et al. "Energy Metering for Free: Augmenting Switching Regulators for Real-Time Monitoring". In: *2008 International Conference on Information Processing in Sensor Networks*. ISPN'08. 2008.

[169] *Nanopower Buck-Boost DC/DC with Integrated Coulomb Counter*. LTC3335. Linear Technology. URL: https://www.analog.com/media/en/technical-documentation/data-sheets/3335f.pdf.

[170] Maydene Fisher, Jon Ellis, and Jonathan Bruce. *JDBC API tutorial and reference*. Addison-Wesley Professional, 2003.

[171] Gabe Fierro et al. "Shepherding Metadata Through the Building Lifecycle". In: *Proceedings of the 7th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*. BuildSys '20. Association for Computing Machinery, 2020.

[172] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: *arXiv* (2018).

[173] *nRF5340 Product Specification*. nRF5340. Rev. 1.3. Nordic Semiconductor. Oct. 2022. URL: https://infocenter.nordicsemi.com/pdf/nRF5340_PS_v1.3.pdf.

[174] *Ultra Low Power Wi-Fi SoC*. DA16200. Rev. 3.5. Dialog Semiconductor. June 2022. URL: https://www.renesas.com/us/en/document/dst/da16200-datasheet.

[175] *Bluetooth Low Energy 4.2 SoC with FLASH*. DA14680. Rev. 3.1. Dialog Semiconductor. Jan. 2022. URL: https://www.renesas.com/us/en/document/dst/da14680-01-datasheet?r=1600471.

[176] *Apollo3 Blue Plus SoC*. Apollo3. Rev. 1.3. Ambiq Micro. Oct. 2022. URL: https://contentportal.ambiq.com/documents/20123/388390/Apollo3-Blue-Plus-SoC-Datasheet.pdf.

[177] *Apollo2 Blue Datasheet*. Apollo2 Blue. Rev. 1.0. Ambiq Micro. May 2019. URL: https://ambiq.com/wp-content/uploads/2020/10/Apollo2-Blue-MCU-Datasheet.pdf.

[178] *Bluetooth® 5.0 SoC with Audio Interface*. DA14585. Rev. 3.4. Dialog Semiconductor. Mar. 2022. URL: https://www.renesas.com/us/en/document/dst/da14585-datasheet.

[179] *CC2650 SimpleLink Multistandard Wireless MCU*. CC2650. Texas Instruments. July 2016. URL: https://www.ti.com/lit/ds/symlink/cc2650.pdf.

[180] *SmartMesh IP Node 2.4GHz 802.15.4e Wireless Mote-on-Chip*. LTC5800. Dust Networks. Dec. 2015. URL: https://www.analog.com/media/en/technical-documentation/data-sheets/5800ipmfa.pdf.

[181] *Bluetooth Low Energy 4.2 SoC*. DA14580. Rev. 3.5. Dialog Semiconductor. Jan. 2022. URL: https://www.renesas.com/us/en/document/dst/da14580-datasheet.

[182] *CC2538 Powerful Wireless Microcontroller System-On-Chip for 2.4-GHz IEEE 802.15.4, 6LoWPAN, and ZigBee Applications*. CC2538. Texas Instruments. Apr. 2015. URL: https://www.ti.com/lit/ds/symlink/cc2538.pdf.

[183] *Low Power, 2.4GHz Transceiver for ZigBee, RF4CE, IEEE 802.15.4, 6LoWPAN, and ISM Applications*. AT86RF233. Atmel Corporation. July 2014. URL: http://ww1.microchip.com/downloads/en/devicedoc/atmel-8351-mcu_wireless-at86rf233_datasheet.pdf.

[184] *CC2520 DATASHEET 2.4 GHZ IEEE 802.15.4/ZIGBEE® RF TRANSCEIVER*. CC2520. Texas Instruments. Dec. 2007. URL: https://www.ti.com/lit/ds/symlink/cc2520.pdf.

[185] *Low-Power Sub-1 GHz RF Transceiver*. CC1101. Texas Instruments. Nov. 2013. URL: https://www.ti.com/lit/ds/symlink/cc1101.pdf.

[186] *2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver*. CC2420. Texas Instruments. Feb. 2013. URL: https://www.ti.com/lit/ds/symlink/cc2420.pdf.

[187] *Single Chip Very Low Power RF Transceiver*. CC1000. Texas Instruments. Jan. 2007. URL: https://www.ti.com/lit/ds/symlink/cc1000.pdf.

[188] *TR1000 916.5Mhz Hybrid Transciever*. TR1000. muRata Electronics. Apr. 2015. URL: https://www.mouser.com/datasheet/2/281/tr1000-785050.pdf.

[189] Omar Abdelatty et al. "A Low Power Bluetooth Low-Energy Transmitter with a 10.5nJ Startup-Energy Crystal Oscillator". In: *ESSCIRC 2019 - IEEE 45th European Solid State Circuits Conference (ESSCIRC)*. 2019.

[190] Mojtaba Sharifzadeh et al. "A fully integrated multi-mode high-efficiency transmitter for IoT applications in 40nm CMOS". In: *2018 IEEE Custom Integrated Circuits Conference (CICC)*. 2018.

[191] Li-Xuan Chuo et al. "7.4 A 915MHz asymmetric radio using Q-enhanced amplifier for a fully integrated 3×3×3mm3 wireless sensor node with 20m non-line-of-sight communication". In: *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. 2017.

[192] Yining Zhang et al. "A 1.9-mW 750-kb/s 2.4-GHz F-OOK Transmitter With Symmetric FM Template and High-Point Modulation PLL". In: *IEEE Journal of Solid-State Circuits* 52 (2017).

[193] Feng-Wei Kuo et al. "A Bluetooth Low-Energy Transceiver With 3.7-mW All-Digital Transmitter, 2.75-mW High-IF Discrete-Time Receiver, and TX/RX Switchable On-Chip Matching Network". In: *IEEE Journal of Solid-State Circuits* 52 (2017).

[194] Yao-Hong Liu et al. "13.2 A 3.7mW-RX 4.4mW-TX fully integrated Bluetooth Low-Energy/IEEE802.15.4/proprietary SoC with an ADPLL-based fast frequency offset compensation in 40nm CMOS". In: *2015 IEEE International Solid-State Circuits Conference - (ISSCC) Digest of Technical Papers*. 2015.

[195] Alexandre Siligaris et al. "A low power 60-GHz 2.2-Gbps UWB transceiver with integrated antennas for short range communications". In: *2013 IEEE Radio Frequency Integrated Circuits Symposium (RFIC)*. 2013.

[196] Sudipto Chakraborty et al. "An ultra low power, reconfigurable, multi-standard transceiver using fully digital PLL". In: *2013 Symposium on VLSI Circuits*. 2013.

[197] Yao-Hong Liu et al. "A 1.9nJ/b 2.4GHz multistandard (Bluetooth Low Energy/Zigbee/IEEE802.15.6) transceiver for personal/body-area networks". In: *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*. 2013.

[198] Alan Wong et al. "A 1V 5mA multimode IEEE 802.15.6/bluetooth low-energy WBAN transceiver for biotelemetry applications". In: *2012 IEEE International Solid-State Circuits Conference*. 2012.

[199] Marco Crepaldi et al. "An Ultra-Wideband Impulse-Radio Transceiver Chipset Using Synchronized-OOK Modulation". In: *IEEE Journal of Solid-State Circuits* 46 (2011).

[200] James Ayers et al. "A 2.4GHz wireless transceiver with 0.95nJ/b link energy for multi-hop battery-freewireless sensor networks". In: *2010 Symposium on VLSI Circuits*. 2010.

[201] G. Retz et al. "A highly integrated low-power 2.4GHz transceiver using a direct-conversion diversity receiver in 0.18µm CMOS for IEEE802.15.4 WPAN". In: *2009 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*. 2009.

[202] Wolfram Kluge et al. "A Fully Integrated 2.4-GHz IEEE 802.15.4-Compliant Transceiver for ZigBee™ Applications". In: *IEEE Journal of Solid-State Circuits* 41 (2006).

[203] Trung-Kien Nguyen et al. "A Low-Power RF Direct-Conversion Receiver/Transmitter for 2.4-GHz-Band IEEE 802.15.4 Standard in 0.18- μm CMOS Technology". In: *IEEE Transactions on Microwave Theory and Techniques* 54 (2006).

[204] *Ultra-low-power Arm Cortex-M33 32-bit MCU+TrustZone+FPU,240 DMIPS, up to 2 MB Flash memory, 786 KB SRAM, crypto*. STM32U585xx. Rev. 6. ST Micro. June 2022. URL: https://www.st.com/resource/en/datasheet/stm32u585ai.pdf.

[205] *Apollo4 Plus SoC*. Apollo4Plus. Rev. 1.1. Ambiq Micro. Aug. 2022. URL: https://contentportal.ambiq.com/documents/20123/388415/Apollo4-Plus-SoC-Datasheet.pdf.

[206] *Apollo 4 SoC*. Apollo 4. Rev. 1.2. Ambiq Micro. Aug. 2022. URL: https://contentportal.ambiq.com/documents/20123/388400/Apollo4-SoC-Datasheet.pdf.

[207] *SiFive FE310-G000 Preliminary Datasheet*. SiFive FE310-G000. Rev. 1.5. SiFive. Sept. 2017. URL: https://sifive.cdn.prismic.io/sifive%5C%2Ffeb6f967-ff96-418f-9af4-a7f3b7fd1dfc_fe310-g000-ds.pdf.

[208] *Apollo2 MCU Datasheet*. Apollo2. Rev. 1.2. Ambiq Micro. Sept. 2020. URL: https://contentportal.ambiq.com/documents/20123/388370/Apollo2-MCU-Datasheet.pdf.

[209] *Ultra-low-power Arm Cortex-M4 32-bit MCU+FPU, 100DMIPS, up to 1MB Flash, 128 KB SRAM, USB OTG FS, LCD, ext. SMPS*. STM32L476xx. Rev. 8. ST Micro. June 2019. URL: https://www.st.com/resource/en/datasheet/stm32l476je.pdf.

[210] *MSP430FR596x, MSP430FR594x Mixed-Signal Microcontrollers*. MSP430FR59. Texas Instruments. Aug. 2018. URL: https://www.ti.com/lit/ds/symlink/msp430fr5969.pdf.

[211] *ATSAM-ARM-based Flash MCU, SAM4L*. SAM4L. Atmel Corporation. Nov. 2016. URL: https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/DataSheets/Atmel-42023-ARM-Microcontroller-ATSAM4L-Low-Power-LCD_Datasheet.pdf.

[212] *Medium-density access line, ARM-based 32-bit MCU swith 64 or 128 KB Flash, 6 timers, ADC and 7 communication interfaces*. STM32F101. Rev. 17. ST Micro. June 2016. URL: https://www.st.com/resource/en/datasheet/stm32f101r8.pdf.

[213] *ARM® Cortex-M3 Stellaris ARM Cortex-M3S 100 Microcontroller IC 32-Bit Single-Core 20MHz 8KB (8K x 8) FLASH 48-LQFP (7x7)*. LM3S101. Texas Instruments.

[214] *MIXED SIGNAL MICROCONTROLLER*. MSP430F2012. Texas Instruments. Dec. 2012. URL: https://www.ti.com/lit/ds/symlink/msp430f2012.pdf.

[215] *8-bit Atmel Microcontroller with 128KBytes In-System Programmable Flash*. ATMEGA128. Atmel Corporation. June 2011. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/doc2467.pdf.

[216] *8-bit Microcontroller with 16K Bytes In-System Programmable Flash*. ATMEGA163. Atmel Corporation. Feb. 2003. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/doc1142.pdf.

[217] *8-bit Microcontroller with 8K Bytes In-System Programmable Flash*. AT90S85. Atmel Corporation. Nov. 2001. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/DOC1041.PDF.

Appendix A

Radio and Processor Energy Datasets

The following datasets were used to draw conclusions about the trade-off between local computation and offloading data to the cloud. See https://github.com/adkinsjd/radio-processor-dataset for a digital version.

Table A.1: Power consumption and efficiency of commercial wireless radios focusing on low-power and particularly efficient or commonly used models over the last 25 years. When multiple transmission powers or data rates were available the most efficient were chosen. Link budget calculated with the receive sensitivity at a standard 1% packet error rate. Efficiency may be slightly higher by pairing transmitters with more sensitive receivers available in a given year. Distance is calculated using the Hata Model for all radios to aid in comparability, however this introduces error at higher frequencies and in indoor settings [78].

Name	Year	Standard	Data Rate	TX Power (dBm)	TX Power Draw (mW)	Energy per bit (nJ)	Band	Link Budget (dBm)	Est. Range (m)	Efficiency (pJ/b/meter)
UBI206 [69]	2022	WiFi	6 Mbps	6	20	3.3	2.4 GHz	97	69	48.3
UBI206 [69]	2022	WiFi	54 Mbps	6	20	3.3	2.4 GHz	77	6.8	53.7
nRF5340 [173]	2021	BLE	1 Mbps	0	13.9	13.9	2.4 GHz	97.5	73	190
DA16200 [174]	2020	WiFi	1 Mbps	9.5	255	255	2.4 GHz	109	275	927
DA16200 [174]	2020	WiFi	6 Mbps	8	255	42.5	2.4 GHz	99	87	489
DA16200 [174]	2020	WiFi	54 Mbps	14	630	11.6	2.4 GHz	90	31	376
DA1469 [65]	2020	BLE	1 Mbps	0	9	9	2.4 GHz	97	69	130
Apollo4 [67]	2020	BLE	1 Mbps	0	15.6	15.6	2.4 GHz	95	55	284
DA14683 [175]	2019	BLE	1 Mbps	0	10.3	10.3	2.4 GHz	94.5	52	198
Apollo3 [176]	2018	BLE	1 Mbps	0	15.6	15.6	2.4 GHz	95	55	284
CC2652 [68]	2018	802.15.4	250 kbps	0	18.9	75.6	2.4 GHz	100	97	779
CC2652 [68]	2018	BLE	1 Mbps	0	18.9	18.9	2.4 GHz	97	69	274
nRF52840 [66]	2017	802.15.4	250 kbps	0	15.9	63.6	2.4 GHz	100	97	256
nRF52840 [66]	2017	BLE	1 Mbps	0	15.9	15.9	2.4 GHz	96	61	261
Apollo2 [177]	2017	BLE	1 Mbps	0	15.6	15.6	2.4 GHz	95	55	284
ESP32 [70]	2017	WiFi	6 Mbps	19.5	840	140	2.4 GHz	112.5	411	341
ESP32 [70]	2017	WiFi	54 Mbps	16	540	10	2.4 GHz	87	22	455
SX1262 [71]	2017	LoRa	980 bps	14	76.5	78,000	868 MHz	143	39,000	2009
SX1262 [71]	2017	LoRa	21,900 bps	14	76.5	3,500	868 MHz	131	9,700	358
DA14585 [178]	2016	BLE	1 Mbps	0	10.2	10.2	2.4 GHz	93	44	232
CC2650 [179]	2016	802.15.4	250 kbps	0	18.3	73.2	2.4 GHz	100	97	755
CC2650 [179]	2016	BLE	1 Mbps	0	18.3	18.3	2.4 GHz	97	69	265
SX1276 [72]	2016	N/A	1,200 bps	7	66	55,000	868 MHz	130	8,300	6,600
SX1276 [72]	2016	N/A	250 kbps	7	66	264	868 MHz	104	414	638

Name	Year	Standard	Data Rate	TX Power (dBm)	TX Power Draw (mW)	Energy per bit (nJ)	Band	Link Budget (dBm)	Est. Range (m)	Efficiency (pJ/b/meter)
SX1276 [72]	2016	LoRa	980 bps	7	66	67,000	868 MHz	139	23,000	2896
SX1276 [72]	2016	LoRa	12,500 bps	7	66	5280	868 MHz	126	5,200	1014
LTC5800 [180]	2015	802.15.4	250 kbps	0	16.2	64.8	2.4 GHz	93	44	1473
DA14580 [181]	2015	BLE	1 Mbps	0	10.3	10.3	2.4 GHz	93	44	234
nRF51822 [43]	2012	BLE	1 Mbps	0	39	39	2.4 GHz	93	44	886
CC2538 [182]	2012	802.15.4	250 kbps	0	72	288	2.4 GHz	97	69	4174
AT86RF233 [183]	2012	802.15.4	250 kbps	0	34.4	141.6	2.4 GHz	101	109	1299
AT86RF233 [183]	2012	802.15.4	250 kbps	0	34.4	141.6	2.4 GHz	101	109	1299
CC2520 [184]	2007	802.15.4	250 kbps	0	77.4	309.6	2.4 GHz	98	77	4021
AT86RF230 [42]	2006	802.15.4	250 kbps	0	43.5	174	2.4 GHz	101	109	1596
CC1101 [185]	2005	N/A	250 kbps	0	47.1	188.4	915 MHz	95	147	1282
CC1101 [185]	2005	N/A	500 kbps	0	47.1	94.2	915 MHz	90	83	1135
CC2420 [186]	2003	802.15.4	250 kbps	0	52.2	208.8	2.4 GHz	95	5	3796
CC1000 [187]	2002	N/A	2400 bps	0	34.8	14,500	915 MHz	110	825	17,500
TR1000 [188]	1998	N/A	115 kbps	0	36	312.5	915 MHz	97	185	1689

Table A.2: Power consumption and efficiency of research radios focusing on low-power and particularly efficient examples. Priority is given to searching for standards-compliant radios. Research falls broadly into two categories: (1) fabricated radios that optimize the circuitry to make traditional active radio transmitters and receivers more efficient and (2) new communication topologies such as passive and back-scatter radios. For both, link budget is calculated with the receive sensitivity at a standard 1% packet error rate. Distance is calculated using the Hata Model for all radios to aid in comparability, however this introduces error at higher frequencies and in indoor settings [78]. When receive sensitivity is not available (such as when research is focused on transmitter optimization) a receive sensitivity is used from the most recent past year. In back-scatter radios packet error rates are often not available, and in these cases distance is taken directly from the communication distances used in the paper's evaluation. It's particularly difficult to compare the efficiency of a passive radio to that of an active radio. Often small changes to the methodology of the back-scatter radio may greatly change the results. Still, we feel it is important to attempt comparison so that dramatic shifts in the efficiency of wireless communication can be anticipated. Note that some marked efficiencies are estimated efficiency only and may not reflect results of a fabricated chip.

Name	Year	Standard	Data Rate	TX Power (dBm)	TX Power Draw (mW)	Energy per bit (nJ)	Band	Link Budget (dBm)	Est. Range (m)	Efficiency (pJ/b/meter)
Varshney et al. [79][a]	2022	N/A	50 kbps	-	0.1	2	868 MHz	-	500	4
Varshney et al. [79][a]	2022	N/A	100 kbps	-	0.1	1	868 MHz	-	105	10
Varshney et al. [87][a]	2019	N/A	1,000 bps	-	0.057	57	868 MHz	-	18	3167
Varshney et al. [87][a]	2019	N/A	2,900 bps	-	0.057	20	2.4 GHz	-	15	1311
Abdellaty et al. [189]	2019	BLE	1 Mbps	-10	2.2	2.2	2.4 GHz	85	17	128
Sharifzadeh et al. [190]	2018	BLE	1 Mbps	-0	4.5	4.5	2.4 GHz	95	55	82
Pérez-Penichet et al. [81][a]	2018	802.15.4	250 kbps	-	0.361	66	2.4 GHz	-	2.5	576
Chuo et al. [191]	2017	N/A	30.3 kbps	-26.1	2	66	915 MHz	93.9	129	512
Talla et al. [85][a]	2017	LoRa	200 bps	-	0.009	47	915 MHz	-	100	12
Talla et al. [85][a]	2017	LoRa	8 kbps	-	0.009	1.18	915 MHz	-	100	200
Y. Zhang et al. [192]	2017	N/A	750 kbps	-10	1.9	2.53	2.4 GHz	88	24	105
Kuo et al. [193]	2017	BLE	1 Mbps	0	3.7	3.7	2.4 GHz	95	55	67
Kellog et al. [83][a]	2016	WiFi	1 Mbps	-	0.015	0.015	2.4 GHz	-	7.6	2
Liu et al. [194]	2015	BLE	1 Mbps	1	4.2	4.2	2.4 GHz	95	55	76
Liu et al. [194]	2015	N/A	2 Mbps	1	4.2	2.1	2.4 GHz	95	55	38
Bharadia et al. [82][a]	2015	N/A	100 kbps	-	-	0.014	2.4 GHz	-	7	2
Bharadia et al. [82][a]	2015	N/A	5 Mbps	-	-	0.009	2.4 GHz	-	1	9
P. Zhang et al. [80][a]	2015	N/A	200 kbps	-	0.077	0.39	915 MHz	-	2.8	139
Kellog et al. [84][a]	2014	N/A	10 kbps	-	0.009	0.9	2.4 GHz	-	2.5	360
Siligaris et al. [195]	2013	N/A	500 Mbps	-	0.025	0.05	60 GHz	-	0.022	2273
Chakraborty et al. [196]	2013	BLE	1 Mbps	0	10.5	10.5	2.4 GHz	100	97	108
Liu et al. [197]	2013	BLE	1 Mbps	0	5.4	5.4	2.4 GHz	98	77	70
Liu et al. [197]	2013	N/A	2 Mbps	0	5.4	2.7	2.4 GHz	96	61	44

Name	Year	Standard	Data Rate	TX Power (dBm)	TX Power Draw (mW)	Energy per bit (nJ)	Band	Link Budget (dBm)	Est. Range (m)	Efficiency (pJ/b/meter)
Wong et al. [198]	2012	BLE	1 Mbps	0	8.9	8.9	2.4 GHz	94	49	182
Crepaldi et al. [199]	2011	N/A	1 Mbps	-14	0.253	0.25	3.5 GHz	48	1	250
Ayers et al. [200]	2010	N/A	1 Mbps	-5.2	1.5	1.5	2.4 GHz	80.8	11	136
Retz et al. [201]	2009	802.15.4	250 kbps	3	38.4	129.6	2.4 GHz	104	154	842
Kluge et al. [202]	2006	802.15.4	250 kbps	3	28.6	113.1	2.4 GHz	104	154	734
Nguyen et al. [203]	2006	802.15.4	250 kbps	0	5.4	21.6	2.4 GHz	101	109	198

[a] Passive or back-scatter radio.

Name	Manufacturer	Year	Architecture	Frequency (MHz)	Efficiency (μA/MHz)
STM32U5 [204]	ST Micro	2022	ARM Cortex M33	160	19
Appollo 4 Plus [205]	Ambiq Micro	2022	ARM Cortex M4	192	4
Apollo 4 [206]	Ambiq Micro	2022	ARM Cortex M4	192	3
nRF5340 [173]	Nordic Semiconductor	2021	ARM Cortex M33	128	55
Apollo 3 [176]	Ambiq Micro	2018	ARM Cortex M4	96	6
SiFive FE310 [207]	SiFive	2017	RISC5	320	500
Apollo 2 [208]	Ambiq Micro	2017	ARM Cortex M4	48	13
nRF52840 [66]	Nordic Semiconductor	2017	ARM Cortex M4	64	56
STM32L4 [209]	ST Micro	2016	ARM Cortex M4	80	39
Apollo 1 [41]	Ambiq Micro	2016	ARM Cortex M4	24	35
DA1468x [175]	Dialog Semiconductor	2016	ARM Cortex M0	96	64
CC2650 [179]	Texas Instruments	2015	ARM Cortex M3	48	61
MSP430FR59 [210]	Texas Instruments	2014	MSP430	16	100
nRF51822 [43]	Nordic Semiconductor	2012	ARM Cortex M0	32	150
SAM4L [211]	Atmel Corporation	2012	ARM Cortex M4	48	90
CC2538 [182]	Texas Instruments	2012	ARM Cortex M3	32	406
STM32F101 [212]	ST Micro	2007	ARM Cortex M3	36	358
LM3S101 [213]	Texas Instruments	2006	ARM Cortex M3	20	2250
MSP430F2012 [214]	Texas Instruments	2005	MSP430	16	220
MSP430F16 [40]	Texas Instruments	2004	MSP430	8	600
ATMEGA128L [215]	Atmel Corporation	2001	AVR	8	1375
ATMEGA128 [215]	Atmel Corporation	2001	AVR	16	2375
ATMEGA163L [216]	Atmel Corporation	2000	AVR	4	1250
ATMEGA163 [216]	Atmel Corporation	2000	AVR	8	1875
AT90S8525 [217]	Atmel Corporation	1998	AVR	8	1250

Table A.3: Processing frequency and energy efficiency of common and particularly performant embedded processors. Clearly not all processors are captured, but an attempt was made to find and include processors that pushed the optimal efficiency forward in a given year and include new processors of a specific architecture, such as early ARM Cortex M and RISC5 processors. Efficiency is reported for the most efficient mode of each processor from each processor's datasheet. Efficiencies are reported at 3.3V. Not all efficiency measurement methodologies are comparable, with some manufacturers disabling flash or running insignificant code to improve efficiency. An attempt was made to include efficiency of some not insignificant code running from flash, however this was not always possible and these changes in methodology could decrease the efficiency of some processors by a as much as factor of 2-3x.