System Design for a Synergistic, Low Power Mote/BLE Embedded Platform

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

***Abstract*—Modern IoT prototyping platforms fall short in terms of energy efficiency, connectivity and software programming practices. We present the design of a new hardware and software platform that addresses these shortcomings by bringing together Mobile, Wearable, Maker and Wireless Sensor Network technologies to enable rapid prototyping with a high degree of synergy and energy efficiency. This is achieved in part by leveraging the Memory Protection Unit on modern microcontrollers along with a novel syscall interface to provide kernel / user isolation and a clean concurrency model. Such a design allows a wide range of languages to be used for application development without significant adaptation. We demonstrate how careful choice of application language allows the naturally asynchronous nature of embedded programming to be expressed cleanly and powerfully. Finally we evaluate the platform in several integrated use cases, providing examples of the capabilities introduced by Synergy.**

I. Introduction

Significant hardware innovation has occurred recently under the IoT banner, particularly in the Maker domain around Arduino and mbed platforms. There has been a move towards more powerful processors offering a wider array of peripherals, with the necessary ease of use being provided by software abstractions, rather than the intrinsic simplicity of 8-bit processors used by earlier hobbyist prototyping platforms. Despite this shift, however, the emphasis remains largely on producing stand-alone devices with poor power characteristics, weak software architecture, communication as an add-on, and little to no concurrency. As design for the Internet of Things takes the foreground, the question of how to resolve these shortcomings must be addressed. This paper argues that a solution can be found in the unification of several fields, with their associated best practices, and we present a bottom-up platform design that achieves this synergy, by holistic hardware and firmware design.

The problem of energy efficiency, communication and concurrency in embedded networked systems has long been at the heart of wireless embedded sensor network design. Here, important guidelines such as idle power, duty cycle and wake-up times are well established [27][16]. The community reached an important plateau in the 2004-8 time frame with hardware devices stabilizing around 16-bit microcontrollers, IEEE 802.15.4 radio modules and robust routing protocols. The issue of connectivity was addressed by many innovations in routing that became incorporated into an IETF standard[20], [33]. The energy cost of communications was addressed by idle listening mechanisms that were incorporated in the IEEE 802.15.4E/G standards. For the most part, any problem demanding energy efficiency resulted in a design resembling that of the prototypical energy-efficient mote: the Mica [19].

Independently, the question of how to interface with humans has seen an explosion of innovation with smartphones and wearables becoming ubiquitous in consumer settings, relying on recent improvements to Bluetooth (Bluetooth Low Energy or Bluetooth Smart[3]) to provide a wireless peripheral link between them.

Again, completely independently, the issue of taming large asynchronous systems in software design has fueled innovation in the web services arena with frameworks introducing patterns for asynchronous programming that offer alternatives to the classic events versus threads debate, particularly *await* from `c#` and `node.js`.

After several generations, embedded 32-bit microcontrollers recently obtained the idle power and fast wake-up characteristics needed to unify these disparate domains, and in doing so resolve the problems faced by modern IoT development platforms. It is now possible to construct a highly capable system following the best practices of the Maker domain, while retaining a low power profile that rivals the best-in-class of the wireless sensor networks domain. The additional computational resources allow for novel software abstractions that simplify energy efficient asynchronous program development, and provide a unified API for both BLE and IP over 802.15.4.

A high level overview of the study is presented by Figure 1, which also provides a framework for the contributions of the work. At the core is a new module, *Storm*, that incorporates the first generation of ARM Cortex M4 to concurrently achieve mote-class power characteristics and modern Maker-class computation and peripheral capabilities. Storm combines this with a state-of-the-art LoWPAN radio and flash. Besides processing power and storage, such modern MCUs bring certain systems aspects of microprocessors into the embedded domain, particularly memory protection, but in a distinct form; for example, there is no MMU.

Drawing on these advances, we develop a true system kernel based on TinyOS [25], with the embedded application in a distinct protection domain, dynamically loaded independent of the kernel, and interacting with it via syscalls. This allows an investigation of a novel syscall interface that projects structured event driven programming into the application tier, while allowing tight low-level real-time processing without the complexity and overhead of preemptive threads.

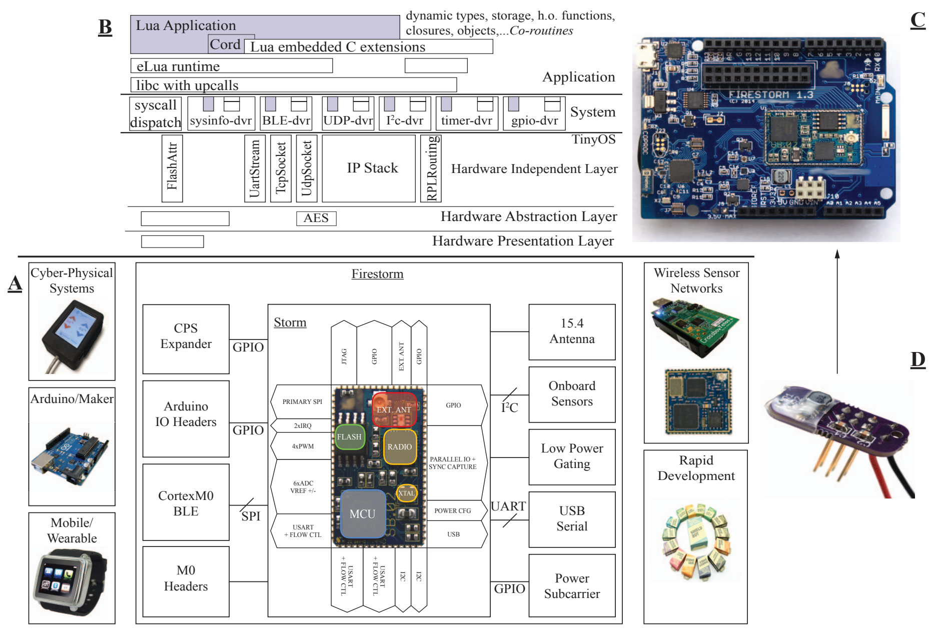


Fig. 1: An overview of the Storm, Firestorm and connected domains of technologies

The Storm module advances the 3P's methodology (prototype, pilot, production) promoted by Dutta et al[16] with a logically and physically coherent presentation of external interfaces. Further, we extend the 3P's into software by implementing an embedded scripting language, Lua, in the application tier. Embedded network applications can be prototyped interactively in Lua then critical components migrated to C, extending the Lua runtime. As the language runtime is built on newlib, any libC based language can be easily ported. It avoids the austere constraints of Mate' like approaches[24] and the complexity of embedded VMs[30][31]. It opens the door to propagating mobile code capsules across the network, an empowering capability in the IoT domain allowing scripting of physical devices and spaces. In the context of a higher level language with closures, we find that the `await` pattern can be utilized to address longstanding issues with event-driven execution without resorting to preemptive multithreading.

We extend the platform integration to engage active Maker innovation communities by building a carrier, *Firestorm*, that is physically and electrically compliant with Arduino shields. These expansion boards support prototyping, but also provide drop-in network infrastructure, such as IP border routers[33], without requiring external Linux boxes, a system weakpoint observed in prior WSN literature[1], [23]. Learning from WSN platform design [27], a dedicated set of sensors (accelerometer, magnetometer, temperature, thermopile, light) support classic sensor network investigations out of the-box. To integrate the final domain – mobile and wearables – the carrier contains a state-of-the-art BLE SoC (with an independently programmable ARM Cortex M0) that supports a dynamically shifting interaction model from BLE central to BLE peripheral. The LoWPAN and the BLE modules are symmetric, so either could be the primary orchestrator of the system; in this study we focus on the Storm-centric perspective. We then provide an in-depth analysis of the power consumption of the BLE domain in comparison with 802.15.4.

Utilizing this overall system design, we evaluate Synergy through a family of case studies. The first illustrates a cross-network application in the Internet of Things domain involving a smartphone interacting with a smart appliance (a fan) that is also interacting with a remote (temperature) sensor over UDP/IPv6/LoWPAN. The second creates a unified service architecture that is projected simultaneously into the BLE and IP domains, the latter engaging LoWPAN, WiFi and cellular, as needed.

II. Storm module

Off-the-shelf hardware platforms have been progressing rapidly in several areas, but moving backwards in others. The three major aspects of a platform are its computational capabilities, its peripheral capabilities and its energy efficiency. Over the past few years we have seen tremendous advances in the the first two categories with off-the-shelf platforms such as mbed, Arduino Zero and Raspberry Pi offering sufficient computation and IO for most problems in the IoT design space. Unfortunately, the third category – energy efficiency – has not advanced. The most energy efficient platform has remained essentially the TelosB[27], an aged WSN mote designed more than a decade ago. Ironically, the problem of energy efficiency is even more relevant now, with wearables being constrained by form factor, driving battery sizes ever lower. In essence, the gap between available development platforms, and the requirements of production is growing.

To facilitate a study into the unification of wireless sensor networks, mobile and wearable devices and the maker space, it is necessary to reconsider the platform design in its entirety. The lessons learned from the various fields must be incorporated early, in the design phase. The resulting module presented here represents an answer to the question "how can we retain the energy efficiency of a WSN mote, while matching the advances in computational power and IO capabilities present in popular modern platforms?" This section describes the design of new platform, Storm, that offers the functionality of a high power system like mbed, while providing the energy efficiency and connectivity of a WSN mote.

Table I shows the important characteristics of Storm compared to leading mote and maker platforms. Many other mote platforms are omitted because their energy profiles are similar or worse than the widely used platforms listed. Likewise, many Maker platforms are omitted as the given platforms are representative. The table shows that we can construct a system that lies between the not-very-capable but energy efficient WSN motes and the capable but energy-inefficient modern prototyping platforms such as Arduino and mbed, a design point that was not possible until recent MCU and SoC generations.

A. Component selection

The selection of the components in a hardware platform determines its dynamic range (from extremely low-power idle mode to feature-rich full-power mode). Care must be taken in this process as a lack of balance results in a platform of limited use. It is this characteristic that undermines the usefulness of platforms like mbed or Rasberry Pi, that excel at the high end but are incapable of operating at the energy efficient end. Similarly, there are existing WSN motes that excel at energy efficiency but lack the computational power for innovation in software architecture.

It is clear that high-performance platforms are trending towards 32-bit ARM Cortex M microcontrollers, with almost all major vendors of maker and wearable technology having Cortex offerings. Previous low-power WSN platforms opted for 8-bit or 16-bit processors, but this choice was not because of intrinsic advantages, but rather the lack of sufficiently low power 32-bit microcontrollers. This is no longer true. At the time of initial design (late 2013), there did not exist an energy efficient Cortex M SoC with an integrated 802.15.4 radio, but there did exist a Cortex M4 that possessed both decent computational power, and exceptional energy efficiency characteristics. Therefore, we decided to proceed with a discrete component based design, as all of the work built on top of it – software programming paradigms for example – would simply transfer over to a SoC when it became available in the future. The resulting module consists of a microcontroller, external flash and radio, following the pattern established by early WSN platforms. As of 2015, there are some SoC offerings that merit investigation, especially the Ti CC2650 which could potentially unify 802.15.4 and BLE in a single chip. The use of a SoC rather than discrete components is orthogonal to this work.

The microcontroller on the Storm module is an Atmel ATSAM4LC8CA, a 48 Mhz Cortex M4 with 512KB of flash and 64KB of SRAM. Most importantly, it can compete with the energy efficiency of best-in-class wireless sensor network platforms. The release of this class of MCU heralds a new epoch in ultra-low-power system design where the benefits of 32-bit processors with the associated increase in flash, RAM and peripherals become relevant.

The AT86RF233 radio is an improvement of the AT86RF230 used by many wireless sensor network platforms. Radio chips have always been concerned with energy efficiency, so the advance here lies in its ability to perform automatic retransmission of unacknowledged packets. This reduces whole-system energy consumption by allowing the MCU to sleep during retransmissions, a significant savings.

The flash chip is an AT45DB641E. This series of external flash was the first to operate over a wide voltage range from 1.7V to 3.6V. Previously, the minimum voltage of the flash was the limiting factor in the system, preventing motes like Telos from operating at the lower voltages supported by the microcontroller. As illustrated by Table II and III, operating at a lower voltage offers significant whole-system energy savings.

B. Form factor

The Storm is constructed as a small (1" by 1/2") single-sided solder-on-module with castellated edges. This module contains all the difficult-to-engineer analog circuitry for the antenna switch, and is assembled en-masse by specialized equipment. Using the module requires no special equipment or knowledge. Furthermore, the exported IO pins are reordered into logical groups (with pins of the same module grouped together) greatly simplifying carrier routing constraints.

This encapsulation approach is well motivated by [16]; separating concerns between a *core* module and *carrier* boards allows for innovative application specific peripheral circuity while still isolating researchers from the task of designing the core functionality which requires *deep expertise* and specialized equipment. It also mirrors the approach of systems like mbed and Arduino which similarly encapsulate the complexity into pre-built module, but export IO for application-specific circuitry.

C. Evaluation and Comparison

We briefly evaluate the Storm module with regards to power consumption, both idle and per unit computation. Table I shows the capabilities of Storm with regard to memory capacity and operating conditions. Additional evaluation of the Storm in the context of a carrier is presented below in Section III.

Table II shows the measured idle power of the Storm and, for comparison, the TelosB with varying supply voltages in the lowest sleep mode that retains a running RTC with alarm interrupts. The Telos is used as a reference because it represents the best-published extreme on the WSN mote energy efficiency spectrum, despite its age. As shown in Table I, detailed idle power comparisons with modern development platforms is meaningless. The result, (Table II) shows that there are significant idle energy savings to be had by controlling the whole-system voltage, a technique that is absent in IoT development platforms.

Platform	GPIO	Pr. Flash(KB)	Ram(KB)	Ext. Flash(Mb)	Performance(DMIPS)	Size (mm^2)	Idle current(μA)
Storm	63	512	64	64	92	416	2.3
TelosB	13	48	10	1	4.4	2040	5.1
MICAz	42	128	4	0.5	5.6	1815	15
Arduino Uno	20	32	2	0	5.6	3663	>40000
Arduino Zero	20	256	32	0	65.4	3663	>1500
mbed LPC1768	32	512	64	16	189	774	>50000

TABLE I: Critical characteristics for several common platforms compared to Storm

Voltage	TelosB μA	Storm μA
3.300	8.8	13.0
3.000	7.2	7.5
2.700	5.9	5.1
2.500	5.1	4.1
2.400	4.6	3.8
2.100	3.5	2.6
1.800	2.6	2.3

TABLE II: Idle power comparison between a TelosB and a Storm, measured side by side

To observe active power characteristics, we measure the power consumption of the device while it is computing the sum of squares over a set of samples. Both 4Mhz and 48Mhz configurations were tested. The results are presented in Table III. This table illustrates that despite the higher active currents of modern 32-bit processors, the energy per unit computation is substantially lower than that of the Telos' MSP430 which possesses lower active current. Energy per unit computation is largely insensitive to speed, and a wide range increases a system's capability, so long as low idle power is maintained. However, it can be seen that the choice of linear (LDO) or switched mode power supply (buck) for the microcontroller's core voltage as well as whole-system operating voltage significantly affects energy use. The choice of power supply mode depends on usage profile, as the buck converter introduces higher idle currents, whereas the LDO introduces higher active currents. The expected duty cycle for a given application must be considered at design time to determine the ideal configuration.

It is common for Maker platforms to be designed purely for their peak power capacity, with no regard for the idle characteristics, as can be seen in the Arduino Zero whose regulator alone has a quiescent current of 1.5mA, two orders of magnitude more than the Storm's whole-system idle current.

III. Synergistic Platform - Firestorm

The Storm module is designed specifically to ease the effort to create application specific carrier boards. For pilot scale or production deployments, these typically address mechanical and electrical requirements as well as adding sensors. Prototype scales demand flexibility and rapid experimentation. Often, the software architected during the prototype and pilot phase changes very little in the move to production, as any change potentially introduces bugs. As a result, it is imperative that the mechanisms for whole-system energy efficiency are present in the prototyping platform, so that the software can be designed to use them.

We present Firestorm, a Storm carrier design (Figure 1 C) that seeks to do this, through mechanisms drawn from wireless sensor network hardware design. Firestorm is a prototyping platform that retains the rapid development process iconified by Arduino by leveraging the wide array of available Arduino shields. At the same time, it seeks to bring this together with WSNs, mobile devices, wearable devices and demanding cyber-physical systems. Storm makes the physical design of such a carrier relatively easy and the Arduino form factor is spacious, so the core challenge is to accommodate the wide variety of power requirements while retaining extreme energy efficiency. Firestorm is partitioned into several power domains, and power modules can be attached. It also allows the 15.4-centric Cortex-M4 Storm module and the BLE-centric Cortex-M0 SoC to function as peers; either processor could be the central system orchestrator.

A. *Wireless Sensor Networks*

A lesson from WSNs is that possessing connectivity and a small number of sensors out-of-the-box can accelerate development. The power management, data transport and aggregation code for many projects can be developed using the default sensors and switched to application specific sensors as they become available. Firestorm provides a suite of on-board sensors: a Texas Instruments TMP006B ambient and IR temperature sensor, a Freescale FXOS8700 3-axis magnetometer/accelerometer, and an Intersil ISL29035 light sensor. 802.15.4 connectivity is provided by the Storm module, and an external antenna.

To obtain low average power and allow the development of energy efficient software, fine grained power domains are provided so that each subsystem can be duty cycled. Average power consists of the *active power*, which is dominated by a small number of high power components such as the MCUs and the radios, and *idle power* which is affected by every single active and passive component on the board.

Idle current is often overlooked in prototype and pilot development, which complicates the migration to production, possibly requiring a complete redesign. On the Firestorm, the 2μA idle current of the accelerometer and the 1.5μA idle current for the temperature sensor would be significant. To avoid this, the sensors are powered from a separate power rail that can be gated off to reach a lower power mode. In addition, the USB serial circuitry is powered from the USB itself and consumes minimal (75 nA) current when there is no USB connection present. This gating of the sensor power supply saves more than 4μA in idle current consumption (more than 25%, depending on the configuration).

Device	Supply (V)	CFG	Freq (Mhz)	Run time (μS)	Current (mA)	Energy (μJ)
TelosB	3.3	-	4	712.7	2.29	5.386
Storm	3.3	LDO	4	393.0	1.049	1.360
Storm	3.3	BUCK	4	393.5	0.501	0.651
Storm	1.8	LDO	4	393.7	0.896	0.634
Storm	3.3	LDO	48	32.8	13.625	1.479
Storm	**3.3**	**BUCK**	**48**	**32.9**	**8.602**	**0.934**
Storm	1.8	LDO	48	32.9	13.124	0.777

TABLE III: Benchmark power comparison results. The bold line indicates the anticipated common configuration

Peripheral	Storm	Firestorm	Arduino Uno
GPIO / sync IRQ	63	42	20
USB Host	1	0	0
PWM Channels	12	12	6
USART	4	3	1
Primary SPI channels	2	2	1
ADC channels	14	7	6
I^2C channels	2 + 2	1 + 1	1
Async IRQ	9	9	2
16bit PD DAC	2	2	0
10bit DAC	1	1	0
8bit parallel port	1	1	0
AES accelerator	1	1	0

TABLE IV: Storm and Firestorm peripheral and IO capabilities

These techniques are essential for producing a long-lived battery-powered device, increasingly prevalent with the Internet of Things. These methods, long present in wireless sensor networks, have not yet found their way into modern Maker and IoT communities. Typical Arduino devices offer no means to gate the power to the USB subsystem and platforms like mbed with acceptable MCU power efficiency lose those characteristics in the whole-system design. In summary, these energy efficiency techniques do not introduce significant cost; they allow for gradual development of the power-aware software that the Internet of Things demands, starting at the prototype phase.

B. *Arduino*

To fully engage large Maker efforts it is necessary to retain a degree of compatibility with technology utilized there. Thus, the form factor of one of the most common Arduino devices, the Uno [2], is used.

The original Arduino platform offers only a limited set of IO and peripherals, as shown in Table IV. But, with the persistence of these headers across several product generations, hundreds of sensors, actuators and communication peripherals are available with a well defined interface. As Table IV shows, it is possible to advance the capabilities of the system while maintaining pin-compatibility; Firestorm provides the same physical layout and a superset of the pin functionalities. Analog pins on the Arduino headers have analog capabilities on the Firestorm, as do PWM, I2C and USART pins. An additional 22-pin parallel port connector is added to support IO intensive applications, such as the display used in our Personal Environmental Control system shown top left in Figure 1 A.

Compatibility with the Arduino space imposes significant supply voltage requirements. Several Arduino shields require 5V supplies, even though they have 3.3V compatible IO signals. Adding a boost converter to the power supply circuit in the Firestorm so that all shields have a 5V supply available to them, even when run from batteries, incur the energy penalty for the boost converter, even when no shields require that power rail. Furthermore, this boost converter circuit cannot be designed optimally since shields and other pluggable peripherals have diverse power requirements ranging from μA to hundreds of mA (in the case of a WiFi or GSM shield). By over-engineering for the maximum possible power draw, performance is lost at lower power levels. This problem is shared by all systems attempting to improve on energy efficiency while maintaining backwards compatibility with existing peripherals.

The answer to this dilemma lies in modularity. To accommodate highly variable power supply requirements (possibly even varying at runtime), Firestorm provides an interface for an application specific power supply "sub-carrier" board. This sub-carrier interface allows a module to provide the 5V rail and the IO voltage rail when needed, along with a communication channel for runtime configuration. The motivation for the IO rail, which also powers the MCU, stems from the processor being being able to run over a voltage range of 1.8V to 3.3V, with the optimal voltage being application specific. As shown in Table II and Table III, the idle and active power consumption of the system drops with a decrease in voltage. It is possible to optimise average power consumption by switching to 3.3V only when communicating with sensors, and running at 1.8V for the "low" parts of the duty cycle.

Figure 1 D shows a simple subcarrier that provides 400 mA on the 5V rail. The carrier mounts on the reverse side of the board so as not to interfere with any shield that may be affixed to the board. In addition, it uses a through-pcb connection so that it can be low-profile. A well designed subcarrier can sit flush with the back of the main PCB.

C. *BLE*

Bluetooth Low Energy (BLE) is an asymmetric protocol that typically provides an attribute based communication channel between a Central and a Peripheral. There have been several developments in using BLE technology for other transport types, such as tunneling IPv6 [28][32] but for many users seeking compatibility with existing mobile devices and wearables, only the most widely adopted pattern is acceptable. Its wide availability as a mobile phone peripheral warrants inclusion in any next-generation sensor network or smart device that interfaces with people.

BLE does not define what "low energy" must be. Devices exist at multiple points on the power consumption spectrum. Central roles such as mobile phones, typically consume much more power than peripherals, but even the peripheral role exhibits great variability. Smart watches, such as Android Wear devices, have a battery life of around two days. The Fitbit Force has a battery life of around 10 days. Devices such as Tile [14] have a stated battery life of a year. This is a testament to the versatility of BLE and to the complexity of the trade-offs between performance and lifetime inherent in a BLE device.

In order to serve as a versatile research platform across a wide range of applications, Firestorm is designed to operate anywhere in this wide domain of BLE applicability. It is suited for high-bandwidth BLE applications, such as streaming accelerometer data from wearable sensors (e.g. [21]) to low power BLE applications that wish to embed battery backed tags in the environment for months or years.

Integrating BLE with WSNs, wearables and the Arduino/-Maker space creates tensions that need to be resolved: low power operation, asymmetric communication roles and complexity. Table V illustrates the success of our design – with the 802.15.4 radio turned off, advertising a BLE beacon with maximum transmission power (+4 dBm, reliably detected by a phone at more than 180 feet away) at a 200 ms interval uses only *140 μA* more than the standby current (90 μA) on average. With a whole system average current of 230 μA in that configuration, a Firestorm could run from 2 AA batteries for more than 400 days. Naturally, even longer lifetimes could be achieved by decreasing the advertising interval.

Wearables and other existing BLE peripherals require a device to be in central mode to interface with them. The NRF51822 BLE used by the Firestorm is capable of operating in either mode. In addition Nordic provides a soft-device that allows for concurrent peripheral and central mode [12]. At the time of writing, this capability has not been explored in depth. We have, however, constructed a proof-of-concept BLE advertisement-scanning mesh using early versions of the software framework to verify functionality in both modes. We are confident that this capability will enable new avenues of research.

The Arduino/maker space demands ease of use. BLE is a complex protocol, arguably far more so than IP. The process of constructing a Generic Attribute Profile (GATT) is involved and presents a conceptual burden to researchers. We show one method of presenting Bluetooth to the application that reduces this complexity in Section IV.

In summary, although BLE originated in a different space from the Maker and WSN fields, it can be integrated cleanly while meeting the requirements the latter spaces impose.

D. BLE/802.15.4 Evaluation

Although the platform integrates both IEEE 802.15.4 and BLE, it allows either to be studied in isolation, due to use

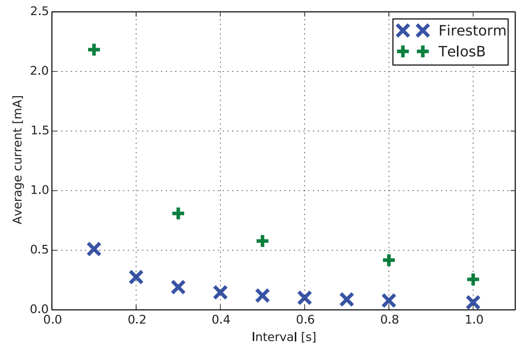


Fig. 2: Average current vs low power listening interval for Firestorm compared to TelosB

of discrete transceivers and MCUs. We look at the power characteristics of a beacon style application and contrast BLE and IEEE 802.15.4. To the best of our knowledge, this is the first such comparison in the literature, beyond preliminary overviews, such as [29]. Figure 3c and 3a show the power consumption of the system while advertising BLE services at 200 ms intervals. Figure 3b shows the transmission of a (larger) 802.15.4 packet; the Lua trace is for reference later in the paper. Figure 2 shows the average RX current of a Firestorm running stock UDPEcho on TinyOS, and how it varies with listening interval.

Note how much more rapidly the BLE SoC is capable of reaching the radio TX state (roughly 2 ms in comparison to roughly 6ms) and how much lower the MCU power consumption is. This plays a large role in BLE's energy efficiency. The slow-to-TX behavior on the TinyOS side seems largely due to software engineering, as the radio itself can transition to the TX state in 100 μs , and the transmission of the packet over SPI would take less than 20 μs if DMA were used.

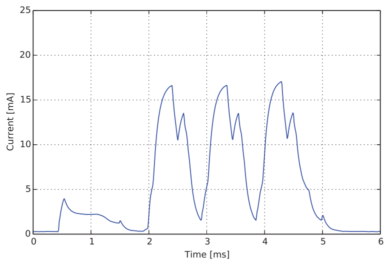
At 2.4V in the buck configuration, the total energy per second for 802.15.4 is 550 μJ over 1 packet (and two LPL intervals). In the same configuration, BLE uses 554 μJ over 5 advertisement groups. Each BLE advertisement group contains 20 bytes of user data sent to three channels, totalling 100 unique bytes sent over the second. The 802.15.4 packet contains 115 bytes, roughly equivalent if the non-user data in BLE advertisements is considered.

When connected to an Android phone that is ranging by polling to determine the RSSI, we see a profile almost identical to Figure 3c. The transmission events occur at 50ms intervals, but we do not observe the high current blocks of packet-trains or listening typical of some low power 802.15.4 MACs.

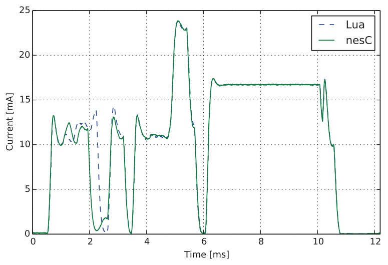
Part of the reason that IEEE 802.15.4 ended up having comparable energy efficiency to BLE, contrary to popular expectation, is that the benchmark of power efficiency in the 802.15.4 literature is out of date. Figure 2 compares the measured 802.15.4 power consumption of the Telos as the listening interval is varied. The Firestorm whole system power is also plotted for reference, and is significantly lower.

E. Power

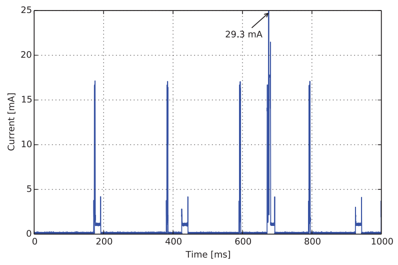
There are several power rails on the Firestorm, as it has to cater to both high functionality attached-to-USB modes of use,



(a) Power trace of an individual BLE advertisement sent to three different channels, note the rapid transition to TX state from idle



(b) Power trace of a 115 byte 802.15.4 packet being sent from a Lua userland and nesC kernel



(c) Power trace of BLE peripheral when advertising with a 200ms interval. The MCU does not contribute significantly, in contrast to current 802.15.4 implementations

Fig. 3: Power traces contrasting BLE and 802.15.4. Normalized by transmitted information, they are roughly equivalent

as well as low power battery backed modes. The Storm module requires only a single supply as it generates the analog supply and core voltage internally. It can, however, generate the core voltage either by using a low dropout linear regulator (LDO) or a buck converter. As explored in [26], the buck converter is more efficient if the processor is active, whereas the LDO is more efficient if the processor is idle. For this reason, the Firestorm allows either configuration to be chosen by adjusting a jumper. The selection can also be made at runtime by the power subcarrier.

By using these techniques when designing the power supplty circuity, the Firestorm's minimum idle current is 9.6 μA at the typical 2xAA voltage of 2.4V (the Storm's idle current at this voltage is 3.8μA). The NRF51822 BLE SoC datasheet claims a 2.6μA sleep current, which brings our theoretical floor to 6.4 μA. The remaining 3.2 μA is due to other components in the system, such as the analog switch that isolates the USB subsystem, and the various high-leakage logic-level-triggered FETs used as ideal diodes and power domain gates.

Table V lists the Firestorm whole-system power consumption under various conditions. Note the significant difference between multiplexed clocks (the application using RTC derived clocks) and dedicated high precision application clocks that preclude the lowest sleep state. Also note the very comparable whole-system power figures for BLE advertising, and 802.15.4 beaconing. They are within 2 μA of each other at 2.4V in the buck configuration. This table is generated by measuring total charge over a length of time and then dividing to get the average current, increasing the precision over direct current measurement.

The key lesson illustrated in Table V is that with careful system design, idle power can be kept remarkably low, even with more capable hardware. Furthermore, the active current is significantly higher than that of lower frequency processors typically used in energy-constrained hardware, so it is imperative that the software be programmed correctly to spend as much time as possible in low power modes.

F. *Symmetry*

An interesting aspect of designing a platform that communicates over BLE and 802.15.4 is that they have several disjoint use cases. Although the Cortex M4 on the Storm is certainly more capable than the Cortex M0 in the NRF51822, it is not clear that one processor would always be the master. For this reason, Firestorm was designed with symmetry of use in mind.

Firstly, the SPI interface between the Storm and the NRF can have either chip as the master. This would allow the M0 to be running application logic, and leave the Storm in ultra low sleep until an 802.15.4 packet needs to be sent, or records need to be stored in flash.

Secondly, the I2C sensor bus containing the accelerometer, magnetometer, light sensor and temperature sensor is connected to both microcontrollers. This means that either can be the master on the bus, or that the bus can be used as an out-of-band channel for additional communication between the processors. For example, the M0 can be the slave on the SPI bus but a master on the I2C bus, allowing it to initiate communication to the Storm when that is required.

This symmetry enables an exploration of the role of BLE and 802.15.4 in a system where they coexist, as it is still a nascent field and the use cases are still being explored.

IV. Rich Embedded Application Tier

The hardware platform presented above allows the exploration of novel approaches to application development that leverage the capabilities of modern microcontrollers. An open question is how best to create a software environment that supports easy, rapid, energy efficient software design. With this in mind, an obvious trade is the reintroduction of a kernel / userland split offering memory protection, independent program loading, and broader software compatibility. Previous WSN operating systems shunned the idea of multiple stacks for application and system code because low power MCU's failed to support this separation efficiently [15]. This has changed. The introduction of a Memory Protection Unit and dual hardware stack pointers into several of the ARM Cortex MCUs permits full isolation with little to no runtime overhead, and an acceptable RAM cost.

Sleep	Clocks	15.4 RX	15.4 TX	BLE	Buck @ 3.3V	Buck @ 2.4V	LDO @ 3.3V	LDO @ 2.4V
enabled	low power	disabled	disabled	disabled	26.0 μA	10.2 μA	25.6 μA	9.6 μA
				standby	91 μA	90 μA	93 μA	84 μA
				advertising	235 μA	231 μA	236 μA	230 μA
		512ms LPL	1 /sec	disabled	197 μA	229 μA	242 μA	257 μA
			disabled	advertising	320 μA	341 μA	346 μA	354 μA
			1 /sec		389 μA	420 μA	429 μA	430 μA
		always on			13.1 mA	12.8 mA	13.0 mA	12.7 mA
	dedicated				17.9 mA	19.3 mA	20.6 mA	20.5 mA
disabled					19.8 mA	21.9 mA	23.6 mA	23.6 mA

TABLE V: Firestorm power consumption

An isolated application tier with its own dedicated memory and stack offers several advantages. Applications can be efficiently preempted to handle the asynchronous tasks posted by kernel interrupts *without any additional code in the application*. Hardware has always supported preemption for interrupt handling, but prioritising the "bottom" ends of interrupt handlers over application logic has required the application to "play along" [18][15].

The arguably more important advantage is that the application need not be designed with the kernel in mind at all, beyond a small well-defined Application Binary Interface in the form of a syscall. We can experiment with different languages, libraries and programming paradigms for application design without re-engineering system services like the network and hardware abstraction layers.

A. Design of a syscall interface

The design of the syscall plays an important role in determining the possible execution modes of the application. If the ABI were POSIX-like, e.g. with the socket interface being read() and write() blocking calls, the kernel could enter sleep state between an invocation and when data is available, saving power. It would, however, limit the ability of the application to implement concurrent logic, sans threads. If a more advanced POSIX-like interface were used, such as select() or epoll(), it would allow more traditional asynchronous programming as found in performant C programs. This comes at the cost of complexity and poor software design as it breaks modularity, requiring a central piece of code to listen for and distribute events.

Our syscall interface overcomes these limitations by distributing events from the kernel directly to the handling logic in the application, facilitating good software design by reducing artificial dependencies between components in the application. A pointer to the handler function and any necessary state is passed with the syscall invocation. Upon completion the kernel triggers the event. For example, an I2C transaction syscall accepts a pointer to the handler that will be invoked when the I/O transaction is complete.

This improves on the split-phase model found in TinyOS, as every invocation of the syscall can accept a unique handler, determined at runtime. The difference is evident when observing a common pattern in TinyOS: complex switch statements in the completion handler that are used to determine the next state (e.g. Figure 5). As custom data cannot be passed from the point

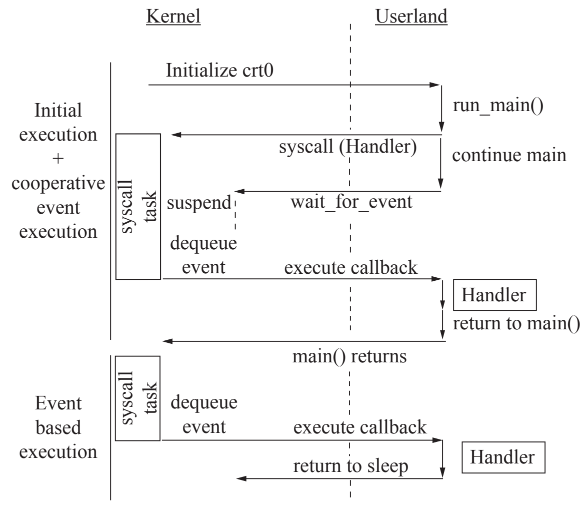


Fig. 4: Execution model of an application that enters fully event based execution

of invocation to the completion handler in TinyOS, global state must be used, and is a source of race-condition bugs. With the syscall interface we have designed, data from the point of invocation can be passed directly to the handler. This emulates handler closures in languages that do not support them.

The syscall event handlers are executed *cooperatively* by the application; they do not interrupt the running application. As shown in Figure 4, a syscall is invoked with the event handler pointer. Later an event is dispatched by executing the `wait_for_event()`, or `check_for_event()` syscall. The `wait` version will sleep if no events are available whereas the `check` version allows the program to continue execution if no events are queued. In the bottom half of the figure, the application returns from main() and enters a fully event based mode where userland execution only occurs in response to an event.

The cooperative, run-to-completion model of callback execution has trade-offs, all of which exist in the context of ECMAScript [6], node.js [11] and other run-to-completion single-threaded execution environments. The primary advantage is simplicity, as it is difficult to introduce a race condition. The primary disadvantage is jitter and latency in the time it takes for an event to be delivered to a handler. Fortunately, the increase in processor speeds has rendered most latency demands easy to meet as the physical-event-based latency constraints have not gotten tighter whereas processor speeds

Driver	Example syscalls
Timer	`start_periodic`, `get_now`
GPIO	`set_mode`, `enable_irq`
SysInfo	`get_MAC`, `get_nodeid`
UDP	`open_socket`, `sendto`
Routing	`add_route`, `lookup_route`
BLE	`add_service`, `add_characteristic`
AES	`encrypt`, `decrypt`
I2C	`read`, `write`
Flash	`read`, `write`

TABLE VI: Currently implemented drivers

continue to increase. For the rare case that this jitter is unacceptable (for some peripheral emulation for example) the logic can be moved into the kernel for stricter guarantees.

This syscall interface, which carries the asynchronous event-based nature of the kernel through to the application, improving energy efficiency, is strictly more capable than the blocking POSIX-like interface. Specifically, if an application calls `wait_for_event()` directly after every syscall invocation, it is functionally equivalent to the blocking version. Such a capability means that applications written in Arduino-style blocking code can still be run, and modified gradually to more powerful execution models.

Any language with a compiler that emits ARM Thumb code can run on top of the kernel and interface with syscalls with *no hardware specific adaptations*. This is because a syscall follows the standard ARM calling convention [17]. As an example, this means that advanced new system languages such as Rust [13] can be used, a topic of ongoing research.

B. Driver design

Apart from the core syscalls offered by the main kernel, such as waiting for an event, most syscalls are implemented as *drivers* – extensions to the kernel that are wired in as standard TinyOS components. This preserves the modularity of the kernel, so that different platforms can contain platform specific logic. We have implemented nine drivers, listed in Table VI. A driver does not take long to write as it is a wrapper around existing TinyOS components.

Although synchronous syscalls (with no callback) have a common behaviour, asynchronous syscalls may have driver-specific logic governing how they are queued and dispatched. By making the queues driver specific, it is possible for different drivers to offer different service agreements on the events, such as "every event has a reserved spot in the queue" which is appropriate for standard asynchronous operations like an I2C transaction, or "best effort with queue depth N" which is appropriate for unbounded operations, such as listening on a socket.

C. LibC support

We implement a full LibC over this syscall interface by adapting newlib [10]. The implications of this are significant: common off-the-shelf embedded applications can be used, without needing to port their hardware abstraction layers to the platform. They can be dropped in, complete with their build environment and the versions of libraries that they ship, compiled for ARM Thumb. An example is the eLua runtime discussed below.

Adapting a third party application to run is simple: remove all the hardware specific code, and write a few LibC support stubs (40 lines of code for newlib based applications). From there, the syscall interfaces can be added: a single line C function per syscall. As this code does not change significantly from application to application, it can be copied verbatim. Applications need not be simple C programs, they can be as complex as full language interpreters, as evidenced by the trivial port of Lua.

D. Lua

Lua [22] is a simple interpreted dynamic language intended for embedding into other applications. It offers closures and coroutines among its many features. It is built entirely on LibC, making it lightweight and portable. The eLua project [7] has combined the Lua runtime with newlib [10] – a popular embedded LibC implementation – and a hardware abstraction layer to allow embedded development.

By writing a few stubs for newlib such as `_sbrk`, `_write` and `_read`, we were able to boot eLua into an interactive prompt with only 40 lines of code. This is on a chip not supported by eLua previously. We are confident that Lua was not a special case; projects such as MicroPython [9] would be equally amenable to running with this syscall interface.

Although Lua was used, any other high level language interpreted language with closures, coroutines and clean interfaces to native code can be used to implement the functionality described below.

E. CORD and Await

A central question we wanted to explore was how the addition of closures in the application tier might simplify writing asynchronous embedded applications. Event handlers can be written in the context of the syscall invocation and can access variables in that lexical scope. While this is a great improvement over the scattered handlers found in TinyOS, it still leads to deeply nested callbacks when there are sequences of asynchronous operations, a scenario colloquially known as *callback hell* [5]. This is a problem for memory use – every callback has access to every variable in its enclosing scope, so the garbage collector cannot free anything until the entire sequence is complete. More significantly, it is a problem for program legibility and complexity.

Alternatively all callbacks could be used synchronously, using the wait-after-syscall method described above. This solves both the legibility and memory problems, producing code that looks like the sequential code typified by Arduino. The problem with this approach is that there can only be one logical thread of execution.

A better approach is the introduction of a mini-scheduler, written in Lua, that uses coroutines to allow concurrent green threads. COROutine Daemon, or CORD, allows multiple logical fibers of execution to run in parallel. With this we can construct a method that converts any asynchronous function into a synchronous one by yielding the coroutine until the callback is invoked. Inspired by the `await` keyword from C# [4], this method receives a function and the parameters with which to invoke it, sans the callback. It then invokes the function with its own callback that receives the parameters and stores them to be passed as return values from the await invocation via Lua's `coroutine.resume()`. The entire CORD file is only 78 lines long, including await and the native support that will be described later.

The significance of `await()` and coroutines is best illustrated by example (Figure 5). With await (Listing 1), functions can be written that encapsulate asynchronous operations and make the interface look synchronous. A closure based model (Listing 2) also allows encapsulating functions to be written, but sequences of asynchronous functions are chained together by nesting. Finally, standard C or nesC (Listing 3) code in the absence of closures leads to highly scattered program logic with an explicit state machine controlling the progression through the asynchronous operations.

F. 3P's for programming

The use of an interpreted language as an application tier brings the concept of 3P's to software development. Lua code can be prototyped interactively via a shell, or pushed in line by line over the network. Individual functions or variables can be tweaked on a running system via encrypted active network capsules. Programs of moderate size (hundreds of lines) can be developed entirely in Lua and pushed into the network to enable dynamic coordination between collections of IoT devices. The presence of a hardware AES accelerator on the MCU allows for rapid (0.68 cycles per byte) encryption and decryption so that secure code dissemination now does not incur a significant energy cost. Using an interpreter in the application tier does not cost much, the dashed line in Figure 3b shows how insignificant is the overhead of sending packets from Lua compared to from the kernel.

As pieces of functionality stabilize and are moved to the pilot stage, they can be migrated into C modules that interface with Lua code. This follows the same pattern as C extensions to Lua on PC. As Lua was designed to be embedded into other applications, the process is straightforward.

The 3P's software pattern not only extends to application logic, but also to driver design. It is possible to interactively develop a new hardware driver in Lua using only a few basic C functions (`peek` and `poke`) and then migrate that to a Lua C implementation for pilot use, and a kernel driver for production use. This is because the kernel can configure the MPU to allow blocks of system memory to be accessible to the application tier at a granularity of individual peripherals. We have found this pattern to be useful in an instructive setting where students learned to configure and control an ADC peripheral from scratch in a single lab, without needing to understand the mechanisms of TinyOS and the kernel. As they progressed, more and more functionality was migrated from Lua to C, allowing comprehension of basic principles to preface comprehension of optimal implementation.

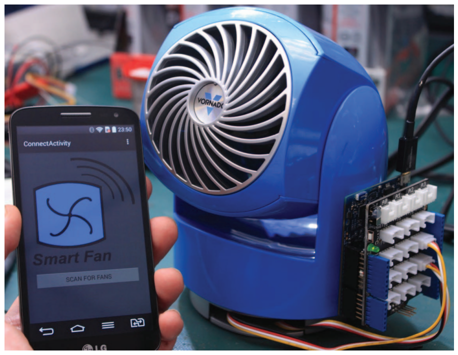


Fig. 6: 802.15.4 and BLE connected prototype smart appliance

V. Evaluation and case studies

We had the opportunity to use this platform to teach an IoT class of roughly twenty students. They were able to rapidly construct applications over ensembles of smart devices with ad-hoc coordination. These ensembles included a smart storybook that discovered nearby devices such as fans, heaters and lights that provided "feel effect" of the environment described in the story, or a meal coordinator that scheduled kitchen appliances to ensure that toast was ready at the same time as the coffee and bacon. Another yielded a standalone LAN-LoWPAN border router with a BLE console and active network style lua capsules for debugging. There are too many to mention, but we present two examples that would not have been possible without the innovations shown in this paper.

A. Cross network application

A simple demonstration of a rapidly assembled cross-network application developed using this synergy is a smart appliance interacting with humans via a mobile phone, and with the environment via distributed sensors communicating over 802.15.4.

This system consists of an off-the-shelf fan that was disassembled, two relays inserted to adjust the fan speed and reassembled. The Grove relays [8] allow for plug-and-play assembly, typical of Maker prototyping tools. This fan exports control via a Bluetooth attribute to an Android app. Additional Firestorms deployed near the fan are discovered and instructed to send temperature readings from their onboard sensors via 802.15.4. The fan can then use this information to make additional control decisions and present it to the user app.

This application is not ambitious, but the entire appliance was disassembled and reassembled in four hours, of which more than three hours were spent on the Android app. The code for the fan is only 37 lines long. The alacrity with which such a cross-network application – involving two network stacks, and GPIO – can be assembled, is empowering.

B. An approachable symmetric service infrastructure

The previous example of a rapidly assembled smart fan shows that a cross network application is easy to develop

Listing (1) Lua with CORD

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

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

Listing (2) Lua Async

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

Listing (3) nesC

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

Fig. 5: I2C Display initialisation under different programming models

on this platform, but there is still a disparity between the functionality offered over Bluetooth and the functionality offered over 802.15.4. It is possible to engineer a specific application such that this is not the case, but it would be nice to abstract away this conceptual burden.

To this end we created a service discovery and utilisation framework with the explicit goal of unifying BLE and 802.15.4. This goal presents some challenges: firstly, BLE is primarily attribute based, not stream based. This means that either a stream type overlay must be built on top of BLE, or an attribute type overlay on top of IP. Secondly, in BLE, the size of each attribute is limited to 20 bytes if one is trying to avoid complexity on the central role. This limits the degree to which one can advertise rich metadata to make services self-describing.

Rather than trying to artificially elevate BLE to offer stream transport – which would complicate the code running on the central role – we opted for creating a set of somewhat restrictive operations on IP that allows parity between the two protocols. This reduces overall complexity significantly.

Figure 7 shows the architecture of this system. BLE discovery of services via advertisements is replicated as discovery via link-local multicast in 802.15.4. The GATT obtained via BLE service discovery is mapped onto a table of UUIDs sent as part of the advertisement payload in 802.15.4.

Additional information to consume the service and translate data from opaque blobs, to machine readable structs or human readable values is encapsulated in a manifest located on GitHub. This manifest also augments BLE, as the protocol is lacking self-description of services. On the phone side, it is accessed at runtime via the internet, whereas for embedded devices that lack the variability introduced by human interaction, the necessary manifest entries are resolved at build time.

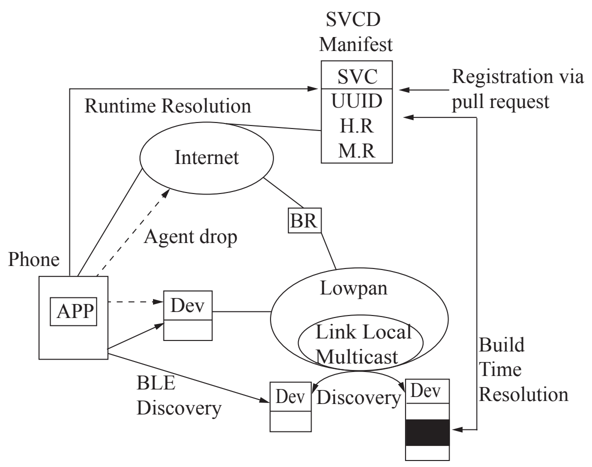


Fig. 7: The SVCD framework

Using this system, services are written with one interface, that is translated to both media. Client interactions via BLE or IP appear identical. This also allows seamless transition between the two, where a phone might discover and initiate relationships between devices over BLE, then drop an agent to the cloud or a local device to persist that relationship over IP when the phone leaves the space. In an instructional setting, students constructed numerous applications as ensembles of such services.

VI. Summary

The platforms that are popular for IoT prototyping and pilot testing have advanced in processing capabilities but remain inadequate in terms of energy efficiency, software architecture and connectivity. Individually, these challenges have been overcome in other domains. By bringing together wireless sensor networks, maker, mobile and wearable technology it is possible to design a platform that retains the best qualities of modern IoT development systems, while addressing the outstanding problems.

Exemplifying this approach, this paper presents Storm, a solder-on module that operates with an energy profile of a best-in-class mote, and the computational and peripheral capabilities of modern maker platforms. It is designed to allow a clear path from prototype, to pilot, to production. Leveraging this, we also present a synergistic carrier platform, Firestorm, that balances the best characteristics of wireless sensor networks, mobile devices, wearables and maker technology. This platform allows in-depth analysis of low power BLE and 802.15.4 techniques while also providing the ideal tool for direct comparison. In addition, Firestorm serves as a design template for future energy efficient IoT development platforms. Using this platform, we create an isolated application tier – implemented with the MPU – showing that it offers accelerated application development and easy adoption of existing embedded frameworks and interpreted languages. We show how one such language – Lua – can elegantly encapsulate the asynchronous syscall interface and reduce application complexity by allowing for pseudo-synchronous programming with coroutines. It also allows for the 3P's model to be applied to software by providing easy prototyping combined with a clear path to production. Finally we showed the power of the platform and application tier together – Synergy – in rapidly prototyping a smart appliance, and developing a symmetric service oriented architecture that unifies BLE and 802.15.4; this allows for agile distributed embedded system composition.

By adopting these whole-system synergistic design principles, we can finally move away from the ubiquitous yet constrained mobile-app-and-wearable incarnation of the Internet of Things to highly interconnected ensembles of long-lived devices, an engine for innovation.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under grant CPS-1239552. Any opinions expressed in this material do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Archived Arch Rock product page. http://web.archive.org/web/20071010023745/http://www.archrock.com/product.

[2] Arduino Uno product page. http://arduino.cc/en/main/arduinoBoardUno.

[3] Bluetooth Smart homepage. http://www.bluetooth.com/Pages/Bluetooth-Smart.aspx.

[4] C# await operator reference. https://msdn.microsoft.com/en-us/library/hh156528.aspx.

[5] Callback Hell website. http://callbackhell.com/.

[6] ECMAScript home page. http://www.ecmascript.org/.

[7] eLua project. http://www.eluaproject.net/.

[8] Grove relay. http://www.seeedstudio.com.

[9] MicroPython - Python for Microcontrollers. http://micropython.org/.

[10] Newlib home page. https://sourceware.org/newlib.

[11] Node.JS home page. https://nodejs.org/.

[12] Nordic S130 soft device. https://www.nordicsemi.com/.

[13] The Rust Language. http://www.rust-lang.org/.

[14] Tile locating tag. https://www.thetileapp.com.

[15] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462, Nov 2004.

[16] P. Dutta, J. Taneja, J. Jeong, X. Jiang, and D. Culler. A building block approach to sensornet systems. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 267–280, New York, NY, USA, 2008. ACM.

[17] R. Earnshaw. Procedure call standard for the arm architecture. *ARM Limited, October*, 2003.

[18] R. Goyette. An analysis and description of the inner workings of the freertos kernel. *Carleton University*, 5, 2007.

[19] J. Hill and D. Culler. Mica: a wireless platform for deeply embedded networks. *Micro, IEEE*, 22(6):12–24, Nov 2002.

[20] J. Hui and P. Thubert. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. RFC 6282 (Proposed Standard), Sept. 2011.

[21] H. Hung, G. Englebienne, and L. Cabrera Quiros. Detecting conversing groups with a single worn accelerometer. In *Proceedings of the 16th International Conference on Multimodal Interaction*, ICMI '14, pages 84–91, New York, NY, USA, 2014. ACM.

[22] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes Filho. Lua-an extensible extension language. *Softw., Pract. Exper.*, 26(6):635–652, 1996.

[23] X. Jiang, S. Dawson-Haggerty, P. Dutta, and D. Culler. Design and implementation of a high-fidelity ac metering network. In *Information Processing in Sensor Networks, 2009. IPSN 2009. International Conference on*, pages 253–264, April 2009.

[24] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. *SIGARCH Comput. Archit. News*, 30(5):85–95, Oct. 2002.

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

[26] T. Nass and A. Andersen. Powering low-power rf products. *Texas Instruments, Design Note DN019*, 2009.

[27] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. In *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, pages 364–369, April 2005.

[28] T. Savolainen and M. Xi. Ipv6 over bluetooth low-energy prototype. In *Aalto University Workshop on Wireless Sensor Systems, Aalto, Finland*, 2012.

[29] M. Siekkinen, M. Hiienkari, J. Nurminen, and J. Nieminen. How low energy is bluetooth low energy? comparative measurements with zigbee/802.15.4. In *Wireless Communications and Networking Conference Workshops (WCNCW), 2012 IEEE*, pages 232–237, April 2012.

[30] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java&#8482; on the bare metal of wireless sensor devices: The squawk java virtual machine. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, pages 78–88, New York, NY, USA, 2006. ACM.

[31] R. B. Smith. Spotworld and the sun spot. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 565–566. ACM, 2007.

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

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