Experiences from a Decade of TinyOS Development

http://www.tinyos.net

Philip Levis
Stanford University
pal@cs.stanford.edu

Abstract

When first written in 2000, TinyOS's users were a handful of academic computer science researchers. A decade later, TinyOS averages 25,000 downloads a year, is in many commercial products, and remains a platform used for a great deal of sensor network, low-power systems, and wireless research.

We focus on how technical and social decisions influenced this success, sometimes in surprising ways. As TinyOS matured, it evolved language extensions to help experts write efficient, robust systems. These extensions revealed insights and novel programming abstractions for embedded software. Using these abstractions, experts could build increasingly complex systems more easily than with other operating systems, making TinyOS the dominant choice.

This success, however, came at a long-term cost. System design decisions that seem good at first can have unforeseen and undesirable implications that play out over the span of years. Today, TinyOS is a stable, self-contained ecosystem that is discouraging to new users. Other systems, such as Arduino and Contiki, by remaining more accessible, have emerged as better solutions for simpler embedded sensing applications.

1. INTRODUCTION

Wireless sensor network research is just over a decade old. Starting as a handful of academic institutions studying networks of tiny, low-power wireless sensing devices, it now has numerous academic conferences and journals that serve a large, worldwide research community. Sensor networks have also grown from research projects to commercial systems. Commercial systems today include ad-hoc wireless smart meter networks, home area networks, and industrial monitoring systems. When Cisco talks about an "Internet of Things," it means the coming Internet with millions or billions of tiny networked devices that interact with and sense the physical environment: sensor networks.

TinyOS is an operating system designed for such embedded devices. It emerged from UC Berkeley in 2000 when sensor network research was beginning, starting as a set of Perl scripts that auto-generated #define statements [23]. Since then, it has evolved to use a C dialect called nesC, has gone through four major revisions, supports tens of sensor network platforms, and has approximately 25,000 downloads per year. TinyOS is the dominant software platform used for sensor network research, enabling hundreds of research results. It is used in numerous commercial products, such as Zolertia [3], Cisco's smart grid systems (formerly Arch Rock), and People Power Company [2].

This paper examines how TinyOS evolved over the past decade. TinyOS is interesting for two reasons. First, like projects such as Xen [11, 44] and OpenFlow [16], TinyOS started as an academic research project that transitioned to significant success and impact outside academia. It managed to make this transition while simultaneously remaining a linchpin of the research community. Second, TinyOS differs from these other examples in that it is a successful, principled, and novel operating system for a new class of computing devices.

This paper examines how technical and social decisions encouraged or restricted the growth of TinyOS and therefore its impact on practice, sometimes in unforeseen ways. For example, fine-grained software components allow users to easily customize the OS with small, local changes. As TinyOS was still forming and being used speculatively in a large number of domains, this easy customization was beneficial. But once core OS services solidified, fine grained components became ultimately harmful, as reading a core system requires leafing through many tiny components.

The paper is divided into four parts. Section 2 describes the two basic principles that have driven TinyOS. The first principle is resource use minimization. The costs of scale and low power operation say that TinyOS code should trade off runtime flexibility or generality for smaller code and data, in contrast to many modern "large" software systems. The bug prevention principle, motivated by the tremendous difficulty of debugging embedded systems, says that TinyOS should be

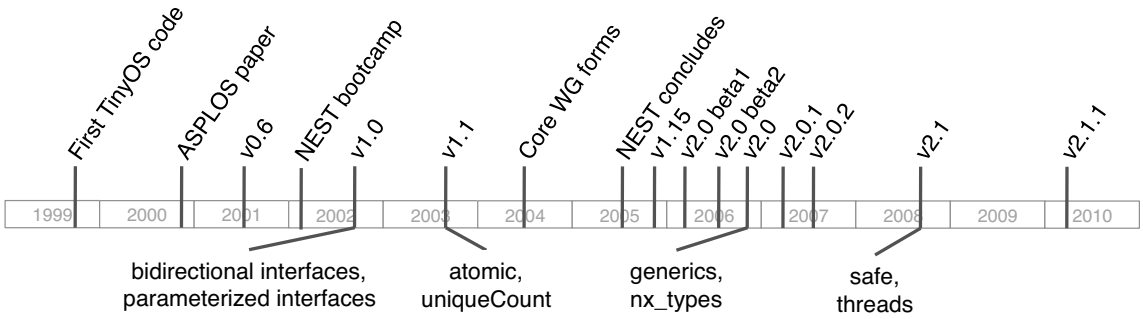

Figure 1: Timeline of major events in TinyOS development from 1999-2010.

structured to make it hard to write bugs, sometimes at the cost of making it generally harder to write code. To help support these principles, TinyOS developers chose to design and use nesC [20], a new C dialect. The language and OS co-evolved, such that it does not make sense to talk about one without the other: when we talk about the evolution of TinyOS, we mean the evolution of both the OS and its language.

Sections 3-6 walk through how four approaches TinyOS took had unforeseen long-term implications. The first two, memory allocation and isolation, relate to the unique properties of embedded software. The second two, components and systems language design, relate to systems software more generally. Section 3 discusses how new language features allowed TinyOS to optimally allocate RAM while simultaneously removing the need for some run-time memory access checks. Section 4 describes how a novel software pattern based on this memory allocation, static virtualization, improves software isolation by making the finite state machine of each virtualized instance completely independent. Section 5 examines how using nesC was critically important to TinyOS's early success, but also how its evolution limited TinyOS from even broader, long-term use. Section 6 looks at the benefits and drawbacks of fine-grained, reusable software components, concluding they are a poor fit for operating systems.

Section 7 examines the TinyOS project from a social perspective: how did the project grow such a large developer community? Open source projects live and die based on their contributors. TinyOS today has a large community of developers and users from all across the world. It examines how this community is structured and how that structure evolved. It presents several pitfalls the project encountered, relating to hiring staff, managing code contributions, and the interactions between academia and industry. It also discusses the role of documentation and target audiences and how the project was able to reduce the barrier to entry caused by its increasing technical complexity.

Section 8 takes a step back to examine lessons from TinyOS that can apply to embedded software, systems more generally, and systems projects. One conclusion is that fine-grained components are good for experimentation but add unnecessary and painful complexity to stable software that expects reuse (e.g., a kernel). A second conclusion is the natural tendency to support long-standing, dedicated users and evolve a system to better meet their needs undermines system adoption. Research wants to push a frontier, but doing so can alienate a broader audience and stifle long-term success. We discuss some ways in which future projects seeking large-scale adoption might avoid these and other pitfalls.

Model	ROM	RAM	Sleep	Price
F2002	1kB	128B	1.3μA	$0.94
F1232	8kB	256B	1.6μA	$2.73
F155	16kB	512B	2.0μA	$6.54
F168	48kB	2048B	2.0μA	$9.11
F1611	48kB	10240B	2.0μA	$12.86

(a) TI MSP430 Microcontrollers

Model	ROM	RAM	Sleep	Price
LM2S600	32kB	8kB	950μA	$2.73
LM3S1608	128kB	32kB	950μA	$4.59
LM3S1968	256kB	64kB	950μA	$6.27

(b) TI ARM CortexM3 Processors

Table 1: A representative sampling of popular processors used in low-power wireless sensors. The price values are from DigiKey's catalog on March 3rd, 2010, when purchased in quantities of 1,000 - 10,000.

2. MINIMIZATION AND PREVENTION

TinyOS's design has two major goals: minimizing resource use and preventing bugs. Both are driven by the unique intersection of requirements that sensor networks pose.

The minimization principle states that TinyOS software should use as few hardware resources as possible.

This means being computationally efficient (minimizing cycle counts and wake time), requiring little state (minimizing RAM) and having very tight code (minimizing code ROM). Traditional computing systems want to be efficient, but they typically trade off some efficiency for flexibility and efficiency in the form of kernel modules, plugins, or other mechanisms. In contrast, TinyOS focuses on producing an ultra-optimized binary that can run unattended for months to years.

Two properties of embedded sensors motivate the minimization principle. The first is energy. Within a device class, parts with more hardware resources draw more power both when awake and when asleep. Since nodes sleep almost all of the time, even small sleep power draws are significant. Table 1 shows a selection of recent microcontrollers. 16-bit MSP430 microcontrollers dominate platforms today, due to their 1.3-2μA sleep draw. An "ultra-low" power 32-bit architecture (ARM Cortex M3), in contrast, has a 950μA sleep current.

As these devices are already designed for ultra-low power operation, there is no low-hanging fruit which will show large improvements in the short term. Furthermore, microcontrollers do not follow Moore's Law due to market and performance considerations that differ from processors. While the first TinyOS prototypes had 8kB of code and 512 bytes of RAM, 48kB of code and 10kB of RAM has been typical for the past seven years.

Harsh energy concerns ("every bit transmitted brings a sensor node one moment closer to death" [36]) cause nodes to spend almost all of their time asleep. Correspondingly, real-time operating systems, such as FreeRTOS [41], eCos [40], and μC/OS-II [32], are a poor fit. Their primary purpose is to schedule use of a limited resource (e.g., a CPU) to meet deadlines, but scheduling is easy when the resource is almost always idle. The other benefit of hard real-time is stability in very precise control systems. This stability breaks down in the presence of an unreliable wireless network and so is typically not useful in practice.

Cost is the second motivation for the minimization principle. While research prototypes use top-end microcontrollers for flexibility (e.g., the bottom row of Table 1(a)), for large scale or commercial use they are overkill and raise prices unnecessarily. Using 16kB of code and 512B of RAM instead of the top-end MSP430 could cut unit costs by $6. For 100,000 units, this $600,000 is well worth the cost of a year of software engineer time to optimize and squeeze overly general code.

Over the first four years of TinyOS development, RAM was generally the most limiting resource. The mica [22] and mica2 [4] platforms have 128kB of ROM and 4kB of RAM, and applications typically hit RAM limits before ROM. Unlike a computer with virtual memory and swap, where a slightly-too-big program will run slowly, there is no margin for error on a microcontroller. A too-big program either has a compile error or crashes almost immediately when the stack overruns data memory.

The prevention principle means preventing bugs through software structure. All software wants to prevent bugs, but TinyOS took a very extreme position due to how astonishingly difficult in-the-field debugging of sensor networks is. Debugging is so difficult that it has prompted a wide range of research [13, 37, 42]. A sensor network is a highly distributed system, where nodes dynamically react to the environment and each other. The limited resources, as well as possible energy constraints, on each device preclude extensive logging or other traditional debugging techniques. Many sensor networks do not even support the equivalent of a TCP connection or other per-node access. How does one debug a node's response to an unknown input?

The sensor network research literature has many papers describing application experiences, from volcanoes [43] to bird burrows [38] to HVAC systems and oil tankers [28] to industrial steam pipe monitoring [46]. Application deployments using early versions of TinyOS almost always report a failure that occurred in bringing the system from lab to deployment, yet are unable to pinpoint the cause of the failure [42]. These experiences by users led TinyOS developers to follow the prevention principle more strongly as it matured. Recent deployment papers that use TinyOS 2.x, such as a hospital application in SenSys 2010 [14] are in comparison unabashed success stories.

To meet these goals, TinyOS and nesC evolved language primitives and programming abstractions to push what are traditionally dynamic, run-time operations into static, compile-time ones. Doing so allowed it to have near-optimal RAM overhead while simultaneously enabling large, complex, and dependable software systems. The next sections examine how TinyOS evolved in four ways: ROM and RAM allocation, code isolation, software components, and language features. Figure 1 shows a timeline of the project between 1999 and 2010 that highlights important organizational and technical events.

3. RAM AND ROM ALLOCATION

TinyOS programs generally require a 10:1 ratio of ROM:RAM ratio. There are exceptions, such as large packet queues or imaging sensors, but a 10:1 rule of thumb is good for predicting whether RAM or ROM will be the limiting resource. For example, TinyOS 1.x was designed predominantly for the mica platform [22], which had an Atmega atm128 microcontroller with 128kB of ROM and 4kB of RAM. Applications on the mica family typically run into RAM limits before ROM. In contrast, the Telos family [35] uses a Texas Instruments

MSP430 with 48kB of ROM and 10kB of RAM; applications on Telos typically run into ROM limits first.

While minimizing CPU cycles is useful, most resource use minimization efforts focused on RAM and ROM. The nesC paper discusses the major techniques used to minimize ROM (inlining and dead code elimination) [20]. RAM reduction, in contrast, was mostly through software structure. RAM received more attention because mica preceded Telos and so applications fought with RAM limits first.

Some design decisions that traded off increased code size for reduced RAM then posed problems for Telos applications. One example of this tradeoff is how a sensor driver configures a chip's analog-to-digital converter (ADC). Configuration options include which pin to sample, the reference voltage, the sample hold time, and the clock source. Before the driver samples the ADC, it must reconfigure it appropriately. Since reconfiguration is very fast (just twiddling a few control bits in registers), ADC software automatically handles the configuration on every sample. A simple way to set these parameters would be for a sensor driver to allocate a structure in RAM with the correct values, which it passes to the ADC software. But this approach means that each sensor driver allocates a structure even though the ADC needs only one of them at any time. This wastes RAM. Instead, TinyOS sensor drivers implement a function that returns their configuration structure directly on the stack (i.e., not a pointer). Rather than maintain the structure in memory, they regenerate it when needed, reducing RAM needs by 4 bytes per client but increasing ROM by 50-60 bytes. This approach worked well for mica, but "ADC bloat" became a common complaint for Telos applications. RAM-conserving and a ROM-conserving APIs look quite different; forcing developers to choose one or the other has the unwanted side effect of making code less portable.

Minimizing the RAM needed by service APIs, in particular, became exceptionally critical. Where in a traditional OS one wants to make system calls fast, in TinyOS we wanted them to require as little RAM state as possible. Take, as an example, the timer service. Many components and systems need timers. Applications need to periodically collect data, routing protocols need to periodically send beacons, and link layers need to manage backoff intervals as well as retransmissions. A complete application can require anywhere from 3 to 15 timers, and each 32-bit timer requires 10 bytes of state (when it started, its interval, and some control bits, such as whether it's a repeating timer). In the best case, the system will allocate 10 bytes for each timer and no more.

The first version of the timer system (pre-1.0) had clients allocate their timer state and pass a pointer into the timer system. On one hand, this meant that applications allocated precisely the right number of timer structures. On the other, it required additional state in each struct: a pointer so the timer implementation could string them into a linked list. The pointer increased the timer structure to 12 bytes, a 20% overhead. Furthermore, the dynamic data structure became a common source of runtime failures due to memory corruption. As each user of the timer service allocated its own structure, a local off-by-one error could corrupt the pointer, breaking the link list. Recall that there was no debugger. After collecting 30 nodes to reprogram them due to a simple memory bug, you don't ever want to again.

In response to difficult experiences debugging timer problems, the second version of the timer system (v1.0) allocated a fixed array of private timer structures. To distinguish different timers, nesC introduced a special function, unique. The nesC compiler evaluates unique at compile time. Each invocation of unique with a given string s returns a unique integer in the range of 0 to $n-1$, where n is the number of times unique is invoked with s. Because there is no binary loading or linking, the nesC compiler parses every call to unique and can compute n correctly. Unique uses the string s as a general way to manage needed sets of unique values. A component that needed a timer allocated a key with unique and passed this key in all calls to the timer system. The second timer implementation used the key to index into its timer structure array.

The second version of the timer system was much more stable, but often wasted even more RAM. Programs made the timer array safely large so calls to unique would not reach past the bounds of the array. This problem was not limited to timers. It existed for ADC sampling, packet queues, and many other components.

The third version of the timer (v1.1) fully minimized RAM through a new nesC function, uniqueCount. Like unique, uniqueCount takes a string and returns an integer. The return value is the number of calls to unique with that string. In the case of timers, for example, the timer service can declare an array of timer state:

```
timer_state_t timers[uniqueCount("Timer")];
```

The unique values can safely access timer state accordingly. Assuming that all timer clients use the correct string (something static virtualization, below, ensures) the timer service can even elide run-time checks that the index parameter is within the size of the array, reducing code size. The final result is that TinyOS today allocates precisely the minimal amount of RAM needed for timers and is 988 bytes of code on mica platforms. If each timer requires 10 bytes of state and there are n timers, it allocates $10n$ bytes of RAM, exactly the minimum required.

4. ISOLATION

Initially, TinyOS did not support dynamic memory allocation of any kind. While the need for more flexible memory allocation became increasingly apparent, so did the dangers of a malloc-like approach. TinyOS 1.1 has many cases where multiple components share a single memory resource. For example, the core OS scheduler provides the abstraction of a "task," a form of deferred procedure call. The scheduler maintains a fixed-size array of tasks to execute. If a component posts a task to a full queue, the post fails. This raises a very difficult failure condition: how does the component repost the task? Since TinyOS has a single stack, the component cannot spin or wait, as the scheduler will not free an entry until the current function returns. Instead, the component must somehow be re-invoked, e.g., by starting a timer. But the timer system uses tasks, and it can drop timers when it cannot post its task.

Packet transmission suffered from a similar problem. In TinyOS 1.x, a transmission request fails if the send queue is full. As this queue is shared across many components, it is possible for one component to fill the queue and starve other senders. Some protocols expect periodic transmissions (e.g., routing beacons) and infer their absence as packet losses. Therefore, the calling semantics in TinyOS 1.x caused several deployments with one badly behaving component to have entire protocols collapse.

We concluded that global, shared memory pools, even when hidden and very limited, were too dangerous for robust software and violated the bug prevention principle. One bad component can create hard to handle failures across the entire system. They lead to hard-to-find or unidentifiable bugs, which are excruciatingly frustrating in embedded platforms with no easy debugging interface.

Over time, it became apparent that a lack of isolation in programming interfaces was a major impediment to writing highly reliable TinyOS 1.x software. By isolating processes, a traditional OS greatly simplifies application implementations. The task queue example shows how TinyOS 1.x components, in contrast, had poor isolation. There were numerous other examples of this limitation, such as the link layer send queue, sensors, and generally almost every OS service except timers. Very robust TinyOS 1.x software therefore had to consider that any operation might fail and handle the error, increasing RAM and ROM use.

TinyOS 2.x improves prevention through better component isolation: it makes each component's interactions to an underlying shared resource completely independent. Each client has a perfectly virtualized instance of the underlying service. For example, the return value of a send packet call is independent of whether other packets are in the transmit queue. Memory allocation for this virtualization, however, needs to occur at compile-time, otherwise it would introduce a source of run-time failure.

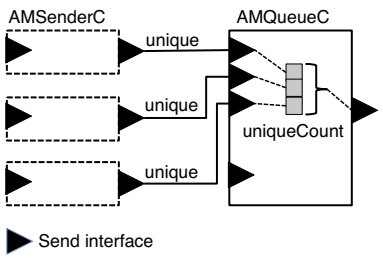

Figure 2: Static virtualization with AMSenderC.

TinyOS 2.x achieves this "static virtualization" behavior by combining generic components and the memory allocation techniques described above in Section 3. Generic components are instantiable nesC components, taking types and constant primitive types as parameters (before 2.x, all nesC components were singletons in a global namespace). Generic components improve code reuse just as Java generic, C++ templates and other similar language mechanisms do.

The basic idea behind static virtualization is that a piece of software can declare a logical (virtualized) instance of a service, such as the ability to send a link layer packet. The behavior of an API is completely independent of all other users of the API. A caller can deterministically know the result of any call, as all transitions in the interface's finite state machine come from that client. This differs from deterministic parallelism [9] in that it is concerned with the behavior of only a single API and avoids shared state.

TinyOS accomplishes static virtualization entirely at compile-time. It uses an abstraction called parameterized interfaces to distinguish between multiple clients, the unique and uniqueCount functions to determine exactly how many clients there are, and generic components to prevent bugs by hiding all of this machinery from the user. In TinyOS 2.x, all APIs to core OS services use static virtualization. For example, to send a link layer packet, a program instantiates an AMSenderC component. AMSenderC has the property that it rejects a valid transmission request if and only if that client already has a transmission request outstanding.

Underneath, AMSenderC connects AMSend to a packet queue, shown in Figure 2. The packet queue has a parameterized Send interface. Each instance of AMSenderC connects to it with a call to unique. The queue uses uniqueCount to allocate the correct number of queue entries. When a component tries to send a packet, the queue checks if the corresponding client's entry in the

queue is occupied. If not, it accepts the packet for transmission; if so, it tells the caller to retry.

Static virtualization is an example of a novel programming abstraction from TinyOS that emerged from the unique requirements that wireless sensors face. We believe it represents a large step forward for highly efficient and dependable embedded software. With static virtualization, software can use an OS service, safely isolated from all other users of the service. Because the behavior of the API is based solely on the calling component, one can statically verify that some components are correct (e.g., with interface contracts [7]). Furthermore, the underlying implementation allocates exactly the amount of RAM needed and has simple, concise code.

5. LANGUAGE/OS CO-DESIGN

Early on in TinyOS development we made the decision to design a language to better support its programming and concurrency model. The nesC language allowed TinyOS to achieve near-optimal resource efficiency (minimization) and a surprisingly low bug rate (prevention). Having a new language also allowed us to evolve and extend features as new problems arose. For example, the language features for static virtualization (parameterized interfaces, unique, uniqueCount, generic components) emerged over a 4 year period. Being able to control both the language and operating system gave the project tremendous flexibility to achieve system design goals.

On one hand, static virtualization is an excellent programming interface. On the other, the software complexity it takes to achieve in nesC turns out to be formidable. Reaching it took a circuitous path through 4 major releases of TinyOS and five years of development. As a result, static virtualization involves emergent, rather than planned, uses of language mechanisms and a handful of programming idioms which are foreign to a new user.

Language evolution is a two-edged sword. As TinyOS became more robust and users began to tackle more challenging software projects, both the OS and nesC language evolved to meet these needs. On one hand, this evolution made it possible to tackle larger and harder problems. On the other, each stage of this evolution added new features, moving TinyOS and nesC further from C and raising the barrier to entry. Furthermore, the most effective software patterns, such as static virtualization, used all of these features in complex and novel ways. By focusing on expert TinyOS users and making it possible to write larger software, TinyOS 2.x became less accessible to new users. Making it harder to write buggy code had the unfortunate result of making it just plain harder to write code.

In retrospect, the focus on expert users missed a great opportunity: hobbyists and the "Maker" do-it-yourself crowd. The past five years have seen a huge growth in simple, DIY electronic projects, spearheaded by Make Magazine [1]. This community has latched onto the Arduino platform [8] for its projects. In comparison to TinyOS, Arduino is feature-poor: programs are single-threaded C programs for simple sensing and actuation. But for hobbyists, the resulting simplicity is extremely desirable. Building a gumball machine that "only dispenses treats when you knock the secret rhythm on its front panel" (an article in a recent issue of Make magazine) doesn't require static virtualization, network types, and compile-time data race detection.

This increase in learning difficulty had more to do with the novel features of nesC and their increased use than APIs or software implementation. Evolving and larger APIs do not increase programmer difficulty in the same way that language features do. A simple program needs to use a limited number of APIs, and the cognitive effort required scales with program complexity. You see this pattern in language communities, but not operating system ones. For example, consider regular expressions in Perl 5 versus Perl 6. Perl 5 regular expressions are similar to those in many other UNIX tools (sed, shells, etc.), so the learning curve for an experienced UNIX can start very gradually. In contrast, Perl 6 regular expressions introduce programming constructions called grammars and rules that require learning from scratch. While the earliest TinyOS programs were mostly C with a bit of nesC to support components, modern code heavily uses many nesC features, making the learning curve very steep.

The steepness of this learning curve has implications to staffing. Academic projects tend have graduate students as their primary developers. This tension between research and engineering can sometimes be solved by hiring staff software engineers. Language evolution, however, complicated this process considerably for TinyOS. Different groups tried several times to hire TinyOS staff programmers, with mixed results. The first staff hire, made early in TinyOS 1.0, contributed a great deal. But he departed in 2005 to work at a sensor network startup. The second was hired in 2004 during the beginning of TinyOS 2.x development. Intel Research tried hiring a software engineer for 1.x: the hire, after a year, produced a single component which had to be thrown away. The third hire had significant experience in event-driven systems, the gulf between Internet services and TinyOS was too wide and he was unable to contribute. In retrospect, hiring staff early in the project, so they can learn the system as it evolves, was much more successful than doing so late, when it had significant and novel complexity.

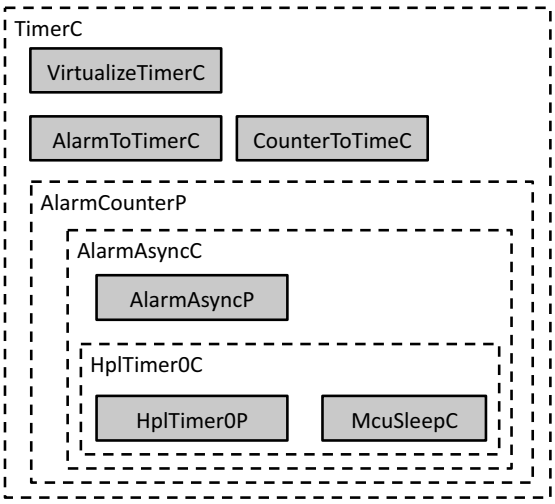

Figure 3: Component structure for the TinyOS 2.x timer implementation. Grey boxes with solid lines are modules (executable code), while white boxes with dashed lines are configurations, components which connect other components together. AlarmCounterP exists to transform its hardware-independent into specific chip implementation (Atm128AlarmAsyncC).

One staff member hired late in the project succeeded in contributing because he was an exceptional and unique case. Before starting as a staff engineer, he was one of the largest contributors to TinyOS 2.x, having researched sensor networks while pursuing a Masters degree. The fact that only someone well inside the community could be a significant contributor later on in the project further demonstrates the barrier to entry that OS/language co-design can create.

Other sensor network OSes have emerged to fill the voids left by TinyOS's evolution. Contiki [5], for example, is written entirely in C and provides a more traditional operating system model of a core kernel and applications which compile against it. While TinyOS is more efficient and cleaner, starting with Contiki is much easier. Today a significant fraction of sensor network research builds on top of Contiki rather than TinyOS.

6. COMPONENTS

The concept of a component is key to TinyOS's programming model. Components separate interface and implementation, provide data privacy, allow code re-use, and provide sufficient linguistic structure for nesC to perform many useful optimizations. For all of these reasons, using components in embedded software is a tremendous improvement over basic C code.

But while components are generally beneficial, they can be used badly. Early on, TinyOS was intended as a research vehicle. We tried to structure software so that it was easy to extend or modify in a small way. Based on user feedback (in particular, early MAC research such as S-MAC [45]), this structure involved many layered compositions of small, lightweight components. For example, if someone wanted to change the MAC timing behavior of the mica platform (carrier sense, backoff), this involves changing one component. Changing the data encoding/decoding involves changing a different component.

The main goal of this approach was to ease experimentation. But taken to its conclusion, fine-grained components have significant drawbacks we did not foresee. Today, the most heavily used radio driver (for the ChipCon CC2420 [15]) is ≈2400 lines of code[1] and 41 different components. The driver consists of 40 files for 2400 lines of code! In a slightly less extreme example, the timer service, shown in Figure 3, involves 8 components that convert a 32kHz counter with compare and overflow interrupts into a millisecond granularity timer component, which becomes the basis for the statically virtualized timer abstraction (another 3 files). What is ultimately less than a kilobyte of code is spread across 11 different files. In terms of prevention and minimization, this is fine. Each small component is easy to verify and debug, and the interfaces between components are designed to avoid the waste of multiple private copies of the same state.

But as Figure 3 suggests, the drawback of fine-grained components emerges when trying to understand a system for the first time. There are so many tiny pieces of functionality spread across files, with numerous levels of indirection, that keeping track of it all can be a headache. The structural complexity is far beyond what the underlying code complexity requires. In the case of the CC2420, one literally has to have 41 different files open at once to see all the code for just one (admittedly very important) driver. When you are implementing the system, all of it makes sense; but to a new user, it's convoluted and complex. A user interface researcher might say this is not a fundamental problem: a good development tool could make browsing this code easy and intuitive. However, we had neither such a tool nor the expertise to build one. While perhaps not a fundamental problem, it is a real and practical one.

For application-level systems, such as GUI toolkits or the Click modular router [27], fine grained components can make sense. Every application is different, and a very flexible toolkit can greatly speed development. But the tradeoffs for an operating system are very different. In the end, there are very few microcontrollers with

[1]We measure lines of code as the number of lines in a file outside of comments that have a semicolon in them.

which one builds a timer system for using the TinyOS timer library, not that many radios which resemble the CC2420 and not that many variations in its use. These libraries are intended to be the basic APIs of an OS; ultimately, application developers want stability, and so there is very little innovation.

Designing generalized fine-grained abstractions can be valuable if you need to integrate multiple, independent changes. For example, one might want to incorporate an alternative MAC protocol (e.g., Funneling MAC [6]) with an alternative packet retransmission scheme (e.g., Partial Packet Recovery [24]). In practice, however, operating system changes are rarely simply localized and rarely compose easily. While implemented as many small components, those components, for sake of code simplicity, end up being tightly coupled.

Our conclusion is that a well designed and carefully implemented operating system is more helpful than an operating system toolkit or operating system software designed with reuse in mind. Our experience with developing more traditional operating systems supports this conclusion. It is easier to take the Linux boot code and modify it for your needs than to work within a component framework for its generalized boot module. We lost sight of the fact that "code reuse" really means within a system, not necessarily across completely independent systems. As both researchers and software engineers, we want to design generalized abstractions, but an excellent artifact is often more useful than a general architecture.

7. COMMUNITY STRUCTURE

TinyOS began as a small research project at UC Berkeley and today has a large, global developer community. Linux's success over HURD in the early 1990s demonstrated that the ability for an open source project to build and maintain an active developer community is as much a result of social interactions and structure as technical concerns.

This section describes how the TinyOS community has evolved socially, focusing on three major considerations: the structure of the community, the relationship between academic and industrial developers, and the effort needed to manage and support users. The prior section described how TinyOS's technical evolution increased its barrier to entry, and this section explores how social mechanisms adopted very late in the project (2007) helped counteract this somewhat.

7.1 Historical Progression

The TinyOS community has gone through two major structural changes, reflecting its major revisions: pre-1.0 from 1999-2002, 1.x from 2002-2005 and 2.x from 2005 to present. We present a very brief overview of these changes as background for later observations and also to acknowledge major contributors.

7.1.1 Pre-1.0

Before version 1.0, TinyOS was a small research project at UC Berkeley [23]. All of the major authors were UC Berkeley students, with some students visiting from UCLA and USC contributing a few components, such as for flooding experiments [19]. At this point in time, there was no real separation between TinyOS development and sensor network research. Research meetings at UC Berkeley discussed major design decisions, and close proximity made social interactions about code similar to most research group codebases.

7.1.2 Building a community: v1.x

When version 1.0 was released, TinyOS had a small community of research users through the DARPA NEST project. These users began to contribute code. In addition to students at UC Berkeley, the TinyOS core system[2] developers included researchers and a staff programmer at Intel Labs Berkeley. The Berkeley NEST project group hired a staff member to organize demonstrations, who began to contribute code.

The TinyOS 1.x core system had 37 developers who checked code into the tree. 23 of the developers were from Berkeley: 16 graduate students, 5 undergraduates and two staff members. 6 were from Intel Research Berkeley, 3 were from Technische Universität Berlin, 2 were from Crossbow, Inc., the company that produced the Berkeley hardware designs, and the last 3 were graduate students from Vanderbilt, UCLA, and Harvard.

Although TinyOS 1.x had many users building systems they sought the community to use, most of the core TinyOS development continued to occur at Berkeley. Code in the main TinyOS tree had to go through regression tests for each release. For most research projects, the responsibility of managing formal releases and performing regular tests on someone else's schedule was much more effort than it was worth. Instead, the research community put code in a separate "contributions" directory. While the core TinyOS 1.x tree had 37 contributors, the contrib directory has 110, spread across over 80 project subdirectories, from Funneling MAC [6] to the Capsule flash storage system [31].

7.1.3 Expanding globally: v2.x

The tight collaboration between Berkeley and TU Berlin was the seed for the core TinyOS development community to expand beyond UC Berkeley. This step forward was auspicious: three of the largest TinyOS contribu-

[2]By "core system" we mean the TinyOS code packaged in a release (the `tos/` directory), not PC-side support tools or other non-nesC code.

tors left Berkeley in the spring of 2005. Two started to work full time at their startup company, Moteiv, while one took a faculty position. Setting up a more formal structure would allow all of them to continue to contribute.

A group of the core developers agreed that TinyOS 1.x had numerous unsolvable structural flaws, mostly relating to reliability (e.g., packet transmission queuing as described in Section 4). They formed the TinyOS 2.x working group in October of 2004. The working group realized that one of the major challenges for new users to TinyOS was its lack of any formal design documents. Because every abstraction in TinyOS was up for review and redesign, small subgroups formed and began to define the new interfaces, documenting them in TinyOS Enhancement Proposals (TEPs), a cross between a Python Enhancement Proposal (PEP) and an Internet Request for Comments (RFC).

The first full release of TinyOS 2.0 took two years. When work started, three companies (Moteiv, Arch Rock, and Crossbow) were all significantly involved in the effort and contributed code. By the time 2.0 was released, however, both Crossbow and Moteiv had dropped out of participation. Arch Rock continued to contribute late into 2007.

A small number of institutions dominate academic contributions. After Berkeley-based development transitioned into Arch Rock, Stanford, and Moteiv, Berkeley development dropped to zero until late 2008 and early 2009. The most consistent and significant academic contributor over time is TU Berlin, which not only wrote many parts of the core OS in 2005-6 but also continued development on extensions (e.g., an 802.15.4 MAC in 2008-10). In 2008, Johns Hopkins contributed a network protocol for reprogramming as well as CC2420 security extensions.

One notable aspect of the commit logs is that they are very bursty. Commits generally relate to a library or contribution, and there are very few background commits, e.g., for fixing bugs. The small number of bugs indicates that TinyOS 2.x has successfully followed the prevention principle. From when the TinyOS 2.x tree moved to Google code in July of 2010 until May 2011, there were 16 bug reports over the approximately 80,000 lines of code of the core system.

7.2 Industry vs. Academia

TinyOS represents a unique point in the design space of open source projects because it deals with embedded systems yet sees very heavy use by the research community. Because debugging embedded code is very difficult, users have a very strong incentive to use existing code rather than write their own: writing a new device driver is a much more daunting task than writing a new protocol. The research community, however, wants to explore ideas on how to improve important systems and so modify existing codes. These conflicting desires between efficient existing codes to use and extensible codes to conduct research has been a continual tension in the TinyOS development community.

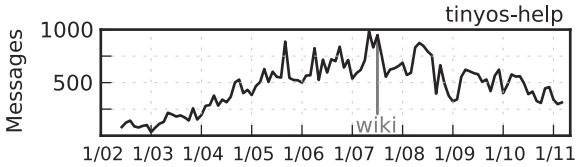

Figure 4: Traffic on tinyos-help mailing list.

TinyOS code development has always been primarily academic. Industrial contributions, however, constitute some of the most critical components of the system. For example, the link layer stack for the ChipCon CC2420 radio, the dominant radio used today, has gone through three iterations. The first was an academic rewrite of the TinyOS 1.x stack; the second was a clean reimplementation by Arch Rock; the third was from Rincon Research and included a low-power mode. While the CC2420 code is only 2,400 lines (3% of the codebase), it is one of the most heavily used, experimented with, and important pieces of code.

Initially, developers who left Berkeley for Moteiv and Arch Rock continued to contribute to TinyOS 2.x, as the companies had expressed commitment to an open-source platform. However, the very different timescales of startups and academia proved to be an irreconcilable tension. Both Moteiv and Arch Rock wanted to settle on a "good enough" platform quickly so they could move on to higher-level services they could sell. The academics, in contrast, saw 2.x as the opportunity to "do things right" and establish a design which would minimize future maintenance. The numerous iterations on the design of very low-level systems, such as power management and locking [26], led both Moteiv and Arch Rock to fork the TinyOS tree and use their own private versions of the codebase.

This forking introduced difficult conflicts of interest. For example, Moteiv released Boomerang, a hybrid version of TinyOS halfway between 1.x and 2.x which supported features in newer Moteiv hardware. Meanwhile, members of Moteiv remained involved in TinyOS 2.x discussions. On one hand, they argued that TinyOS 2.x was going in directions contrary to the needs of their customers. On the other, Moteiv had stopped contributing code towards this end, and changing course to follow these suggestions would slow TinyOS 2.x development and cause more people to use Boomerang.

7.3 Managing and Supporting Users

Today, TinyOS averages approximately 50-100 downloads/day, or 18,000-36,000 per year (the numbers spike on a release). This number does not include developer downloads via CVS, SVN, or git: it is solely downloads of RPMs, debian packages, and VMWare images. We obtained this count by examining web logs for downloads of these three formats from the TinyOS distribution server, pruning by agent to remove search bots, and then counting the number of unique IP addresses. Unfortunately the bi-monthly rotation of web logs (as well as a server replacement in 2010) prevents us from giving detailed download statistics over time.

Managing a user base so large is difficult, especially because every developer is a volunteer. Computer science graduate students have very little motivation to actively support users. Support is also especially challenging due to the fact that TinyOS is used in many university courses, whose students represent a huge variety of technical ability. Developers typically do not mind answering "interesting" questions or responding to bug reports, but questions on more mundane issues such as Java classpath problems, general C programming questions such as "is there array for TinyOS," or "where do I download TinyOS" become wearying after a few months, let alone a decade.

Tinyos-help, the main help mailing list, started in May 2002. Figure 4 shows the posts per month between then and now. There are two interesting trends: first is the annual dip in traffic around the new year, due to the winter holidays. The second is that messages on the mailing list peaked in June of 2007 with 947 messages that month. Since that time, there has been a steady, downward trend. This downward trend does not correlate with a downward trend in downloads.

What happened in 2007 that made mailing list support easier? In July 2007 the Documentation Working Group formed to move TinyOS from a set of static documentation web pages to a documentation wiki that anyone could modify and improve. Over time, documentation since then has continually improved. Anecdotally, our experience with tinyos-help is that the reduction in traffic since then has not been uniform across types of questions. Traffic on -help has become bimodal, either consisting of the most rudimentary questions by posters who have not bothered to look at the documentation (or search the web), or detailed technical questions. Developers ignore the former and typically respond to the latter. The response to a common, recurring question is typically a pointer to the site-specific Google search for the web accessible tinyos-help archives.

While the process of writing tutorials, API reference documents, and programming manuals is neither glamorous nor exciting, the presence of these materials reduced the long-term effort needed to support a large user community.

8. LESSONS LEARNED

In the past decade TinyOS has transitioned from C preprocessor macros maintained by a graduate student at Berkeley to 80,000 lines of code written in a new C dialect by a worldwide community of academic and industrial developers. Arriving at this point involved some good steps and some mistakes. This section tries to answer the question: if we could do it all over with hindsight, what would we do similarly and what would we do differently? Or more generally, how would we recommend growing a systems software research project to be adopted outside academia? We focus on 5 specific decisions TinyOS made: adopting nesC, its focus on software components, its reliance on a research community as users, its collaboration with industry, and how it developed documentation. When possible, we draw parallels between a few other academic software projects.

8.1 Good: Language Extensions

Ultimately, the decision to go with nesC was the right one: nesC's language features allowed developers to write robust code that used very few hardware resources. Had we stayed with C, it seems unlikely that TinyOS-based sensor networks would be as advanced as they are today. Chances are another project, realizing the limitations of C, would have tried an alternative language approach. Furthermore, nesC gave us the flexibility to discover novel programming abstractions that are not possible in C and greatly improve system development, such as static virtualization.

8.2 Bad: How Language Extensions Evolved

While the decision to use nesC was a good one, how TinyOS used it should have evolved differently. On one hand, eating your own proverbial dog food is important: TinyOS developers built applications and systems, giving them experience with the strengths and weaknesses of the system. On the other hand, doing so led to a distorted perception of what was hard or important. Chasing hard, unsolved problems makes sense from a research standpoint. But from a practical standpoint, making it easier to solve hard problems can simultaneously make it harder to solve the easy ones, and this happened with TinyOS.

In retrospect, it would have been better to split the system design and evolution efforts into two halves. The first half would be to make it easier to build larger and more complex systems. The second half would make it easier to build trivial systems. Motivating systems and networking graduate students to take this second approach would most likely fail. But, for example, sup-

pose TinyOS developers had engaged with work at Stanford [21] or Carnegie Mellon [10] on rapid sensor device prototyping. It could have possibly enabled whole new application domains and use for low-power wireless sensing devices. Arduinos, which have moved to fill this capacity, have very limited network capabilities: who knows what interesting new scientific experiments, art pieces, or toys could have appeared if TinyOS were used instead?

On the other hand, nesC's evolution discovered new and better ways to write efficient, bug-free embedded code. Going forward, the right thing to do is to completely redesign the language, or design a new one, to make these concepts basic language structures rather than complex uses of more general features. For example, one could have a way to define a static virtualization that automatically sets up parameterization, unique, and state management: a single file could define the service, rather than the typical case today of at least 4 files.

8.3 Good: Software Components

Components are a significant improvement over basic C code. They provided clean, reusable interfaces, data privacy, and enabled many tools for checking and verifying TinyOS code. They encourage clean system decompositions, which enabled small groups of programmers to build intricate, complex systems, ranging from shooter localization applications [30] to the Tenet programming system [33].

8.4 Bad: Software Component Architectures

Faced with such capabilities, however, the inevitable academic tendency is to generalize and define architectures for core services, such as TinyOS' network layer architecture (NLA) [18]. But these generalizations, in practice, turn out to usually be much more effort than they are worth. If there is only a small handful of implementations for any given abstraction (e.g., forwarding policy, link estimator), the structural complexity that generalization adds is detrimental. "Don't generalize; generalizations are generally wrong." [29] In practice, clean, easy-to-understand code without too much structural complexity can be easier to copy and modify. We should have started with fine-grained components, then over time transitioned to more monolithic implementations as they stabilized.

8.5 Good: Initial Users

Without the NEST project, TinyOS would not have moved far past Berkeley and a few other schools. NEST motivated Berkeley developers to embark on sometimes boring software engineering projects: others would use their code and so the work had impact. It also led to software from outside Berkeley that others could use, extend, and compare against. Finally it created momentum and interest in the form of social memory and knowledge. When a researcher thinks about using TinyOS, chances are they know another person or group who is already doing so, whom they can learn from.

While it's obvious that getting an initial group of users is critical, how does one do so? There are two basic mechanisms. The first is to promote use internally, among other groups or researchers who might find the system useful. One notably successful example is the Click modular router [27]. Click, originating at MIT, has been used in many research projects from that institution, such as Roofnet [12] and wireless network coding [25]. These demonstrations of Click's success as not only a research project but also a practical tool have helped it now be used by many researchers and companies.

The second approach, which is generally more easily successful, is to have a funding agency give grants to work on the system. For the NEST project, using TinyOS was essentially a requirement. Of course, this has drawbacks as well. Some NEST participants still resent that they had to use TinyOS. Other examples that followed this approach are less extreme, such as DHash++ [17] as part of IRIS, or PlanetLab [34] and Intel.

8.6 Bad: Focusing on Experts

In retrospect, focusing on growth within the research community exacerbated TinyOS' focus on technical complexity. The project, just as with technical directions, should have also focused on broadening participation. But in seeking research impact, the project sought impact predominantly with researchers.

While it is possible to achieve impact by having users outside the research community (X from MIT, BSD from Berkeley, Mach from CMU, Xen from Cambridge, and more recently OpenFlow from Stanford, are notable examples), this is especially difficult for embedded software. Embedded systems are often closed, single-vendor, vertically integrated systems where the vendor gives few if any real details on the underlying technology. Anecdotally, through we know of many companies who use TinyOS in products, only a tiny handful will say so on the record.

8.7 Bad: Early Industrial Involvement

When effort on TinyOS 2.x started, several companies were involved in the design process. Each of them, however, dropped out within nine months as their development timescale was much, much faster than academia. Frustrated by the long discussions and numerous design iterations, both Moteiv and Arch Rock forked from the main tree to develop their own branches. The frustration was also due to differing goals: both Arch Rock and

Moteiv wanted to focus on the hardware platform they both used, while academic groups representing multiple hardware devices wanted more generality.

What is especially revealing is that many of the early criticisms from Crossbow and Moteiv on the programmability of TinyOS 2.x were, in retrospect, completely correct. While we believe involving industry in early design was a mistake, TinyOS would have benefited from more carefully listening to the requirements industrial collaborators presented. Instead, early industrial partners departed the project in frustration.

8.8 Good: Late Industrial Involvement

Once the core design was complete in early 2006, however, companies such as Rincon Research, Handhelds.org, Zolertia, and Shockfish began to join the project and contribute significantly. This code was typically drivers for their platforms, although it also included a few utility libraries. Given a well defined structure and precise, stable interfaces, commercial engineers were willing to participate and contribute without having to accommodate what must at times seem like philosophical debates about hypothetical universes. The OpenFlow project at Stanford lends additional evidence that incorporating commercial contributors later, not earlier, is a better approach than the one TinyOS tried. The original designs of OpenFlow and Xen originated within Stanford and Cambridge. Over time, the projects enlisted industrial partners who are willing to implement, extend, and use the system.

8.9 Good: Diverse Documentation

As a user community grows, documentation is absolutely critical to keeping down the support effort needed. Writing documentation can be time consuming, but is worth the long-term time savings in answering questions. TinyOS ultimately gravitated towards three forms of documentation: tutorials, for getting started, TEPs, which are API and implementation references, and a TinyOS programming manual (over 200 pages) that goes into excruciating detail on advanced programming and software engineering techniques. Tutorials acclimate a new user to how to write a program and use some simple functionality; TEPs explain most of the system functionality, for when a user wants to build something new; the programming manual helps when a user wants to write a reasonably large and complex piece of software.

One sometimes frustrating result of good documentation is that you hear very little from users: no news is good news. After the TinyOS documentation wiki started, some developer wondered whether the slow reduction in questions was due to the TinyOS community slowly fading away. But the number of downloads indicates otherwise: more people are downloading TinyOS, but fewer are asking questions about it.

8.10 Bad: Only Developer Documentation

It's challenging for someone to write documentation intended for an audience with a vastly different technical background. When TinyOS was early in its evolution and not yet very complex, documentation written by its developers was reasonably accessible to other C programmers. But as the system become more complex and developer expertise increased, tutorials became simultaneously longer and more obtuse.

In retrospect, TinyOS was far too late in transitioning documentation to a wiki. There was a bit of a control concern: if you open documentation to the masses, they might write something incorrect. But generally, for every mistake, there will be ten additions that are correct. If a user thinks a certain piece of documentation is needed, trust that thought. For example, one of the earliest community documentation contributions, the second link on `http:/docs.tinyos.net`, is a tiny page that demonstrates the simplest TinyOS program. We initially thought that something so minor should be in a tutorial, or deeper in the site, but in retrospect realized we should leave it to users to decide.

However, one cannot simply create a wiki and expect users to populate it with content for free. Developers have to heavily seed the documentation effort. Users, like everyone else, are much more motivated to improve something that's there than to create out of whole cloth.

9. CONCLUSION

A decade is a long time, especially for an academic project. TinyOS was able to transition from the academic halls of UC Berkeley into a worldwide community of developers and users. Getting to this point involved tens of thousands of hours of work by hundreds of contributors. In retrospect, some decisions that seemed sound at the time had significant negative long-term implications that we did not foresee. For example, while designing language extensions for better operating systems programming is valuable, co-evolving those extensions with the OS can alienate new users, limiting the long-term benefits of the work.

TinyOS has been a critical enabler for wireless sensor network research and engineering, the benefits of which we see in efforts like the IETF developing standards for connecting low-power wireless sensors to the Internet [39]. As computing increasingly pervades society, the ability for universities to transition research into practical, real-world impact and benefits will remain important and valuable. Our hope is that the lessons we learned so may help others trying to do so in the future.

Acknowledgments

TinyOS is the collaborative work of many developers, too many to list here, who all deserve credit for its success. I'd like to especially acknowledge Jason Hill, David Gay, Cory Sharp, Joe Polastre, Vlado Handziski, Jan Heinrich-Hauer, Kevin Klues, David Moss, Omprakash Gnawali, Jonathan Hui, John Regehr, Matt Welsh, Alec Woo, Robert Szewczyk, Kamin Whitehouse, Philip Buonadonna, Ben Greenstein, and Miklos Maroti. In addition, David Culler's leadership, Eric Brewer's language design insights, and Shankar Sastry's application knowledge all set the trajectory that led to the operating system's longevity and success. Last but certainly not least, TinyOS would never have succeeded without its users, their bug reports, feature requests, and hard work that helped define the early years of sensor network research.

I'd also like to thank the program committees of SOSP 2011 and OSDI 2012. Their reviews provided excellent advice on what aspects of TinyOS's history could be most beneficial to other researchers and engineers.

This work was supported by generous gifts from Microsoft Research, Intel Research, DoCoMo Capital, Foundation Capital, the National Science Foundation under grants #0615308 ("CSR-EHS"), #0627126 ("NeTS-NOSS"), and #0846014 ("CAREER"), as well as a Stanford Terman Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

10. REFERENCES

[1] Make magazine. `http://makezine.com`.

[2] People Power Company. `http://www.peoplepowerco.com`.

[3] Zolertia Wireless Sensor Networks. `http://www.zolertia.com`.

[4] Mica2 schematics. `http://webs.cs.berkeley.edu/tos/hardware/design/ORCAD_FILES/MICA2/6310-0306-01ACLEAN.pdf`, Mar. 2003.

[5] Adam Dunkels and Björn Grünvall and and Thiemo Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (IEEE EmNetS-I)*, 2004.

[6] G.-S. Ahn, S. G. Hong, E. Miluzzo, A. T. Campbell, and F. Cuomo. Funneling-MAC: a Localized, Sink-Oriented MAC for Boosting Fidelity in Sensor Networks. In *Proceedings of the Fourth International Conference on Embedded Networked Sensor Systems (SenSys)*, 2006.

[7] W. Archer, P. Levis, and J. Regehr. Interface Contracts for TinyOS. In *Proceedings of the Sixth International Conference on Information Processing in Sensor Networks (IPSN)*, 2007.

[8] Arduino Team. Arduino home page. `http://www.arduino.cc`.

[9] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient System-Enforced Deterministic Parallelism. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[10] D. Avrahami and S. E. Hudson. Forming Interactivity: A Tool for Rapid Prototyping of Physical Interactive Products. In *Proceedings of the Fourth Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques (DIS)*, 2002.

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[12] J. Bicket, D. Aguayo, S. Biswas, and R. Morris. Architecture and Evaluation of an Unplanned 802.11b Mesh Network. In *Proceedings of the Eleventh Annual International Conference on Mobile Computing and Networking (Mobicom)*, 2005.

[13] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo. Declarative Tracepoints: a Programmable and Application Independent Debugging System for Wireless Sensor Networks. In *Proceedings of the Sixth International Conference on Embedded Network Sensor Systems (SenSys)*, 2008.

[14] O. Chipara, C. Lu, T. C. Bailey, and G.-C. Roman. Reliable Clinical Monitoring Using Wireless Sensor Networks: Experiences in a Step-Down Hospital Unit. In *Proceedings of the Eighth International Conference on Embedded Networked Sensor Systems (SenSys)*, 2010.

[15] ChipCon Inc. CC2420 Data Sheet. `http://www.chipcon.com/files/CC2420_Data_Sheet_1_4.pdf`, 2006.

[16] O. S. Consortium. Openflow. `http://www.openflow.org/`.

[17] F. Dabek. *A Distributed Hash Table*. PhD thesis, Cambridge, MA, USA, 2005.

[18] C. T. Ee, R. Fonseca, S. Kim, D. Moon, A. Tavakoli, D. Culler, S. Shenker, and I. Stoica. A Modular Network Layer for Sensorsets. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[19] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. An Empirical Study of Epidemic Algorithms in Large Scale Multihop Wireless Networks. UCLA Computer Science Technical Report UCLA/CSD-TR 02-0013, 2002.

[20] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[21] B. Hartmann, S. R. Klemmer, M. Bernstein, L. Abdulla, B. Burr, A. Robinson-Mosher, and J. Gee. Reflective Physical Prototyping through Integrated Design, Test, and Analysis. In *Proceedings of the Nineteenth Annual ACM Symposium on User Interface Software and Technology (UIST)*, 2006.

[22] J. Hill and D. E. Culler. Mica: a Wireless Platform for Deeply Embedded Networks. *IEEE Micro*, 22(6):12–24, 2002.

[23] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, 2000.

[24] K. Jamieson and H. Balakrishnan. PPR: Partial Packet Recovery for Wireless Networks. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, 2007.

[25] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Médard, and J. Crowcroft. XORs in the Air: Practical Wireless Network Coding. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, 2006.

[26] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis. Integrating Concurrency Control and Energy Management in Device Drivers. In *Proceedings of Twenty-First ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[27] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[28] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra,

M. Flanigan, N. Kushalnagar, L. Nachman, and M. Yarvis. Design and Deployment of Industrial Sensor Networks: Experiences from a Semiconductor Plant and the North Sea. In *Proceedings of the Third International Conference on Embedded Networked Sensor Systems (SenSys)*, 2005.

[29] B. W. Lampson. Hints for Computer System Design. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles (SOSP)*, 1983.

[30] M. Maroti, G. Simon, A. Ledeczi, and J. Sztipanovits. Shooter localization in urban terrain. *Computer*, 37(8):60–61, Aug. 2004.

[31] G. Mathur, P. Desnoyers, P. Chukiu, D. Ganesan, and P. Shenoy. Ultra-Low Power Data Storage for Sensor Networks. *ACM Transactions on Sensor Networks*, 5(4):33:1–33:34, 2009.

[32] Micrium. The uC/OS-II Kernel. `http://micrium.com/page/products/rtos/os-ii`.

[33] J. Paek, B. Greenstein, O. Gnawali, K.-Y. Jang, A. Joki, M. Vieira, J. Hicks, D. Estrin, R. Govindan, and E. Kohler. The Tenet Architecture for Tiered Sensor Networks. *ACM Transactions on Sensor Networks*, 6(4):34:1–34:44, 2010.

[34] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences Building PlanetLab. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[35] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling Ultra-Low Power Wireless Research. In *Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks (IPSN)*, 2005.

[36] G. Pottie. Casting the Wireless Sensor Net. MIT Technology Review, July 2003.

[37] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the Sensor Network Debugger. In *Proceedings of the Third International Conference on Embedded Networked Sensor Systems (SenSys)*, 2005.

[38] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. An Analysis of a Large Scale Habitat Monitoring Application. In *Proceedings of the Second International Conference on Embedded Networked Sensor Systems (SenSys)*, 2004.

[39] T. Winter and P. Thubert (editors). RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. Internet Network Working Group RFC6550, March 2012.

[40] The eCos project. eCos v2.0 Embedded Operating System. `http://ecos.sourceware.org`.

[41] The FreeRTOS project. FreeRTOS – Free professional grade RTOS. `http://www.freertos.org`.

[42] M. Wachs, J. I. Choi, J. W. Lee, K. Srinivasan, Z. Chen, M. Jain, and P. Levis. Visibility: a New Metric for Protocol Design. In *Proceedings of the Fifth International Conference on Embedded Networked Sensor Systems (SenSys)*, 2007.

[43] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and Yield in a Volcano Monitoring Sensor Network. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[44] Xen.org community. The Xen Hypervisor Project. `http://xen.org`.

[45] W. Ye, J. Heidemann, and D. Estrin. An Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *In Proceedings of the Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2002.

[46] C. Zhang, A. Syed, Y. H. Cho, and J. Heidemann. Steam-Powered Sensing. In *Proceedings of the Ninth International Conference on Embedded Networked Sensor Systems (SenSys)*, 2011.