TinyOS: An Operating System for Sensor Networks

Philip Levis[1], Sam Madden[1,2,3], Joseph Polastre[1], Robert Szewczyk[1], Kamin Whitehouse[1], Alec Woo[1], David Gay[2], Jason Hill[4], Matt Welsh[2,5], Eric Brewer[1] and David Culler[1]

[1] EECS Department, University of California, Berkeley, Berkeley, California 94720 {pal, madden, polastre, szewczyk, kamin, awoo, brewer, culler}@cs.berkeley.edu
[2] Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley, California 94704 {madden, dgay, mdw}@intel-research.net
[3] CSAIL, MIT, Cambridge, MA 02139, madden@csail.mit.edu
[4] JLH Labs, 35231 Camino Capistrano, Capistrano Beach, CA 92624, jhill@jlhlabs.com
[5] Division of Engineering and Applied Sciences, Harvard University, Cambridge, MA 02138, mdw@eecs.harvard.edu

Abstract

We present TinyOS, a flexible, application-specific operating system for sensor networks. Sensor networks consist of (potentially) thousands of tiny, low-power nodes, each of which execute concurrent, reactive programs that must operate with severe memory and power constraints. The sensor network challenges of limited resources, event-centric concurrent applications, and low-power operation drive the design of TinyOS. Our solution combines flexible, fine-grain components with an execution model that supports complex yet safe concurrent operations. TinyOS meets these challenges well and has become the platform of choice for sensor network research; it is in use by over a hundred groups worldwide, and supports a broad range of applications and research topics. We provide a qualitative and quantitative evaluation of the system, showing that it supports complex, concurrent programs with very low memory requirements (many applications fit within 16KB of memory, and the core OS is 400 bytes) and efficient, low-power operation. We present our experiences with TinyOS as a platform for sensor network innovation and applications.

1 Introduction

Advances in networking and integration have enabled small, flexible, low-cost nodes that interact with their environment and with each other through sensors, actuators and communication. Single-chip systems are now emerging that integrate a low-power CPU and memory, radio or optical communication [75], and MEMS-based on-chip sensors. The low cost of these systems enables embedded networks of thousands of nodes [18] for applications ranging from environmental and habitat monitoring [11, 51], seismic analysis of structures [10], and object localization and tracking [68].

Sensor networks are a very active research space, with ongoing work on networking [22, 38, 83], application support [25, 27, 49], radio management [8, 84], and security [9, 45, 61, 81], as a partial list. A primary goal of TinyOS is to enable and accelerate this innovation.

Four broad requirements motivate the design of TinyOS:

1) Limited resources: Motes have very limited physical resources, due to the goals of small size, low cost, and low power consumption. Current motes consist of about a 1-MIPS processor and tens of kilobytes of storage. We do not expect new technology to remove these limitations: the benefits of Moore's Law will be applied to reduce size and cost, rather than increase capability. Although our current motes are measured in square centimeters, a version is in fabrication that measures less than 5 mm^2.

2) Reactive Concurrency: In a typical sensor network application, a node is responsible for sampling aspects of its environment through sensors, perhaps manipulating it through actuators, performing local data processing, transmitting data, routing data for others, and participating in various distributed processing tasks, such as statistical aggregation or feature recognition. Many of these events, such as radio management, require real-time responses. This requires an approach to concurrency management that reduces potential bugs while respecting resource and timing constraints.

3) Flexibility: The variation in hardware and applications and the rate of innovation require a flexible OS that is both application-specific to reduce space and power, and independent of the boundary between hardware and software. In addition, the OS should support fine-grain modularity and interpositioning to simplify reuse and innovation.

4) Low Power: Demands of size and cost, as well as untethered operation make low-power operation a key goal of mote design. Battery density doubles roughly every 50 years, which makes power an ongoing challenge. Although energy harvesting offers many promising solutions, at the very small scale of motes we can harvest only microwatts of power. This is insufficient for continuous operation of even the most energy-efficient designs. Given the broad range of applications for sensor networks, TinyOS must not only address extremely low-power operation, but also provide a great deal of flexibility in power-management and duty-cycle strategies.

In our approach to these requirements we focus on two broad principles:

Event Centric: Like the applications, the solution must be event centric. The normal operation is the reactive execution of concurrent events.

Platform for Innovation: The space of networked sensors is novel and complex: we therefore focus on flexibility and enabling innovation, rather then the "right" OS from the beginning.

TinyOS is a tiny (fewer than 400 bytes), flexible operating system built from a set of reusable components that are assembled into an application-specific system. TinyOS supports an event-driven concurrency model based on split-phase interfaces, asynchronous *events*, and deferred computation called *tasks*. TinyOS is implemented in the NesC language [24], which supports the TinyOS component and concurrency model as well as extensive cross-component optimizations and compile-time race detection. TinyOS has enabled both innovation in sensor network systems and a wide variety of applications. TinyOS has been under development for several years and is currently in its third generation involving several iterations of hardware, radio stacks, and programming tools. Over one hundred groups worldwide use it, including several companies within their products.

This paper details the design and motivation of TinyOS, including its novel approaches to components and concurrency, a qualitative and quantitative evaluation of the operating system, and the presentation of our experience with it as a platform for innovation and real applications. This paper makes the following contributions. First, we present the design and programming model of TinyOS, including support for concurrency and flexible composition. Second, we evaluate TinyOS in terms of its performance, small size, lightweight concurrency, flexibility, and support for low power operation. Third, we discuss our experience with TinyOS, illustrating its design through three applications: environmental monitoring, object tracking, and a declarative query processor. Our previous work on TinyOS discussed an early system architecture [30] and language design issues [24], but did not present the operating system design in detail, provide an in-depth evaluation, or discuss our extensive experience with the system over the last several years.

Section 2 presents an overview of TinyOS, including the component and execution models, and the support for concurrency. Section 3 shows how the design meets our four requirements. Sections 4 and 5 cover some of the enabled innovations and applications, while Section 6 covers related work. Section 7 presents our conclusions.

2 TinyOS

TinyOS has a component-based programming model, codified by the NesC language [24], a dialect of C. TinyOS is not an OS in the traditional sense; it is a programming framework for embedded systems and set of components that enable building an application-specific OS into each application. A typical application is about 15K in size, of which the base OS is about 400 bytes; the largest application, a database-like query system, is about 64K bytes.

2.1 Overview

A TinyOS program is a graph of components, each of which is an independent computational entity that exposes one or more *interfaces*. Components have three computational abstractions: *commands*, *events*, and *tasks*. Commands and events are

Interface	**Description**
`ADC`	Sensor hardware interface
`Clock`	Hardware clock
`EEPROMRead/Write`	EEPROM read and write
`HardwareId`	Hardware ID access
`I2C`	Interface to I2C bus
`Leds`	Red/yellow/green LEDs
`MAC`	Radio MAC layer
`Mic`	Microphone interface
`Pot`	Hardware potentiometer for transmit power
`Random`	Random number generator
`ReceiveMsg`	Receive Active Message
`SendMsg`	Send Active Message
`StdControl`	Init, start, and stop components
`Time`	Get current time
`TinySec`	Lightweight encryption/decryption
`WatchDog`	Watchdog timer control

Fig. 1. Core interfaces provided by TinyOS.

mechanisms for inter-component communication, while tasks are used to express intra-component concurrency.

A *command* is typically a request to a component to perform some service, such as initiating a sensor reading, while an *event* signals the completion of that service. Events may also be signaled asynchronously, for example, due to hardware interrupts or message arrival. From a traditional OS perspective, commands are analogous to downcalls and events to upcalls. Commands and events cannot block: rather, a request for service is *split phase* in that the request for service (the command) and the completion signal (the corresponding event) are decoupled. The command returns immediately and the event signals completion at a later time.

Rather than performing a computation immediately, commands and event handlers may post a *task*, a function executed by the TinyOS scheduler at a later time. This allows commands and events to be responsive, returning immediately while deferring extensive computation to tasks. While tasks may perform significant computation, their basic execution model is run-to-completion, rather than to run indefinitely; this allows tasks to be much lighter-weight than threads. Tasks represent internal concurrency within a component and may only access state within that component. The standard TinyOS task scheduler uses a non-preemptive, FIFO scheduling policy; Section 2.3 presents the TinyOS execution model in detail.

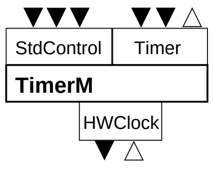
TinyOS abstracts all hardware resources as components. For example, calling the `getData()` command on a sensor component will cause it to later signal a `dataReady()` event when the hardware interrupt fires. While many components are entirely software-based, the combination of split-phase operations and tasks makes this distinction transparent to the programmer. For example, consider a component that encrypts a buffer of data. In a hardware implementation, the command would instruct the encryption hardware to perform the operation, while a software implementation would post a task to encrypt the data on the CPU. In both cases an event signals that the encryption operation is complete.

The current version of TinyOS provides a large number of components to application developers, including abstractions for sensors, single-hop networking, ad-hoc routing, power management, timers, and non-volatile storage. A developer composes an application by writing components and wiring them to TinyOS components that provide implementations of the required services. Section 2.2 describes how developers write components and wire them in NesC. Figure 1 lists a number of core interfaces that are available to application developers. Many different components may implement a given interface.

2.2 Component Model

TinyOS's programming model, provided by the NesC language, centers around the notion of *components* that encapsulate a specific set of services, specified by *interfaces*. TinyOS itself simply consists of a set of reusable system components along with a task scheduler. An application connects components using a *wiring specification* that is independent of component implementations. This wiring specification defines the complete set of components that the application uses.

The compiler eliminates the penalty of small, fine-grained components by whole-program (application plus operating system) analysis and inlining. Unused components and functionality are not included in the application binary. Inlining occurs across component boundaries and improves both size and efficiency; Section 3.1 evaluates these optimizations.



```
module TimerM {
  provides {
    interface StdControl;
    interface Timer[uint8_t id];
  }
  uses interface Clock;
}
implementation {
  ... a dialect of C ...
}
```

Fig. 2. Specification and graphical depiction of the `TimerM` component. *Provided interfaces are shown above the `TimerM` component and used interfaces are below. Downward arrows depict commands and upward arrows depict events.*

A component has two classes of interfaces: those it *provides* and those it *uses*. These interfaces define how the component directly interacts with other components. An interface generally models some service (e.g., sending a message) and is specified by an *interface type*. Figure 2 shows a simplified form of the `TimerM` component, part of the TinyOS timer service, that provides the `StdControl` and `Timer` interfaces and uses a `Clock` interface (all shown in Figure 3). A component can provide or use the same interface type several times as long as it gives each instance a separate name.

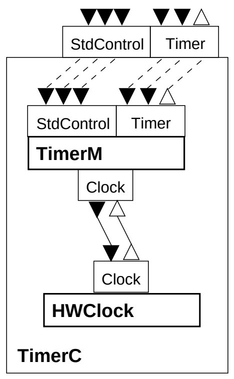
Interfaces are *bidirectional* and contain both *commands* and *events*. A command is a function that is implemented by the providers of an interface, an event is a function that is implemented by its users. For instance, the `Timer` interface (Figure 3) defines `start` and `stop` commands and a `fired` event. Although the interaction between the timer and its client could have been provided via two separate interfaces (one for its commands and another for its events), grouping them in the same interface makes the specification much clearer and helps prevent bugs when wiring components together.

```
interface StdControl {
  command result_t init();
  command result_t start();
  command result_t stop();
}

interface Timer {
  command result_t start(char type, uint32_t interval);
  command result_t stop();
  event result_t fired();
}

interface Clock {
  command result_t setRate(char interval, char scale);
  event result_t fire();
}

interface SendMsg {
  command result_t send(uint16_t address,
                        uint8_t length,
                        TOS_MsgPtr msg);
  event result_t sendDone(TOS_MsgPtr msg,
                          result_t success);
}
```

Fig. 3. Sample TinyOS interface types.

NesC has two types of components: *modules* and *configurations*. Modules provide code and are written in a dialect of C with extensions for calling and implementing commands and events. A module declares private state variables and data buffers, which only it can reference. Configurations are used to wire other components together, connecting interfaces used by components to interfaces provided by others. Figure 4 illustrates the TinyOS timer service, which is a configuration (`TimerC`) that wires the timer module (`TimerM`) to the hardware clock component (`HWClock`). Configurations allow multiple components to be aggregated together into a single "supercomponent" that exposes a single set of interfaces. For example, the TinyOS networking stack is a configuration wiring together 21 separate modules and 10 subconfigurations.

Each component has its own interface namespace, which it uses to refer to the commands and events that it uses. When wiring interfaces together, a configuration makes the connection between the local name of an interface used by one component to the local name of the interface provided by another. That is, a component invokes an interface without referring explicitly to its implementation. This makes it easy to



```
configuration TimerC {
  provides {
    interface StdControl;
    interface Timer[uint8_t id];
  }
}
implementation {
  components TimerM, HWClock;

  StdControl = TimerM.StdControl;
  Timer = TimerM.Timer;

  TimerM.Clk -> HWClock.Clock;
}
```

Fig. 4. TinyOS's Timer Service: the `TimerC` configuration.

perform interpositioning by introducing a new component in the component graph that uses and provides the same interface.

Interfaces can be wired multiple times; for example, in Figure 5 the `StdControl` interface of `Main` is wired to `Photo`, `TimerC`, and `Multihop`. This fan-out is transparent to the caller. NesC allows fan-out as long as the return type has a function for combining the results of all the calls. For example, for `result_t`, this is a logical-AND; a fan-out returns failure if any subcall fails.

A component can provide a *parameterized interface* that exports many instances of the same interface, parameterized by some identifier (typically a small integer). For example, the the `Timer` interface in Figure 2 is parameterized with an 8-bit `id`, which is passed to the commands and events of that interface as an extra parameter. In this case, the parameterized interface allows the single `Timer` component to implement multiple separate timer interfaces, one for each client component. A client of a parameterized interface must specify the ID as a constant in the wiring configuration; to avoid conflicts in ID selection, NesC provides a special `unique` keyword that selects a unique identifier for each client.

Every TinyOS application is described by a *top-level configuration* that wires together the components used. An example is shown graphically in Figure 5: `SurgeC` is a simple application that periodically (`TimerC`) acquires light sensor readings (`Photo`) and sends them back to a base station using multi-hop routing (`Multihop`).

NesC imposes some limitations on C to improve code efficiency and robustness. First, the language prohibits function pointers, allowing the compiler to know the precise call graph of a program. This enables cross-component optimizations for entire call paths, which can remove the overhead of cross-module calls as well as inline code for small components into its callers. Section 3.1 evaluates these optimizations on boundary crossing overheads. Second, the language does not support dynamic memory allocation; components statically declare all of a program's state, which prevents memory fragmentation as well as runtime allocation failures. The restriction sounds more onerous than it is in practice; the component abstraction eliminates

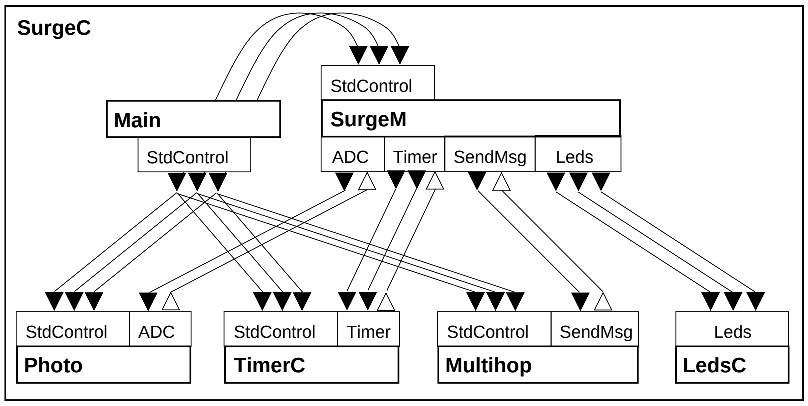


Fig. 5. The top-level configuration for the Surge application.

many of the needs for dynamic allocation. In the few rare instances that it is truly needed (e.g., TinyDB, discussed in Section 5.3), a memory pool component can be shared by a set of cooperating components.

2.3 Execution Model and Concurrency

The event-centric domain of sensor networks requires fine-grain concurrency; events can arrive at any time and must interact cleanly with the ongoing computation. This is a classic systems problem that has two broad approaches: 1) atomically enqueueing work on arrival to run later, as in Click [41] and most message-passing systems, and 2) executing a handler immediately in the style of active messages [74]. Because some of these events are time critical, such as start-symbol detection, we chose the latter approach. NesC can detect data races statically, which eliminates a large class of complex bugs.

The core of the execution model consists of run-to-completion *tasks* that represent the ongoing computation, and interrupt handlers that are signaled asynchronously by hardware. Tasks are an explicit entity in the language; a program submits a task to the scheduler for execution with the `post` operator. The scheduler can execute tasks in any order, but must obey the run-to-completion rule. The standard TinyOS scheduler follows a FIFO policy, but we have implemented other policies including earliest-deadline first.

Because tasks are not preempted and run to completion, they are atomic with respect to each other. However, tasks are not atomic with respect to interrupt handlers or to commands and events they invoke. To facilitate the detection of race conditions, we distinguish synchronous and asynchronous code:

Synchronous Code (SC): code that is only reachable from tasks.
Asynchronous Code (AC): code that is reachable from at least one interrupt handler.

The traditional OS approach toward AC is to minimize it and prevent user-level code from being AC. This would be too restrictive for TinyOS. Component writers need to interact with a wide range of real-time hardware, which is not possible in general with the approach of queuing work for later. For example, in the networking stack there are components that interface with the radio at the bit level, the byte level, and via hardware signal-strength indicators. A primary goal is to allow developers to build responsive concurrent data structures that can safely share data between AC and SC; components often have a mix of SC and AC code.

Although non-preemption eliminates races among tasks, there are still potential races between SC and AC, as well as between AC and AC. In general, any update to shared state that is *reachable from AC* is a potential data race. To reinstate atomicity in such cases, the programmer has two options: convert all of the conflicting code to tasks (SC only), or use *atomic sections* to update the shared state. An atomic section is a small code sequence that NesC ensures will run atomically. The current implementation turns off interrupts during the atomic section and ensures that it has no loops. Section 3.2 covers an example use of an atomic section to remove a data race. The basic invariant NesC must enforce is as follows:

> **Race-Free Invariant***: Any update to shared state is either SC-only or occurs in an atomic section.*

The NesC compiler enforces this invariant at compile time, preventing nearly all data races. It is possible to introduce a race condition that the compiler cannot detect, but it must span multiple atomic sections or tasks and use storage in intermediate variables.

The practical impact of data race prevention is substantial. First, it eliminates a class of very painful non-deterministic bugs. Second, it means that composition can essentially ignore concurrency. It does not matter which components generate concurrency or how they are wired together: the compiler will catch any sharing violations at compile time. Strong compile-time analysis enables a wide variety of concurrent data structures and synchronization primitives. We have several variations of concurrent queues and state machines. In turn, this makes it easy to handle time-critical actions directly in an event handler, even when they update shared state. For example, radio events are always dealt with in the interrupt handler until a whole packet has arrived, at which point the handler posts a task. Section 3.2 contains an evaluation of the concurrency checking and its ability to catch data races.

2.4 Active Messages

A critical aspect of TinyOS's design is its networking architecture, which we detail here. The core TinyOS communication abstraction is based on Active Messages (AM) [74], which are small (36-byte) packets associated with a 1-byte handler ID. Upon reception of an Active Message, a node dispatches the message (using an event) to one or more handlers that are registered to receive messages of that type. Handler registration is accomplished using static wiring and a parameterized interface, as described above.

AM provides an unreliable, single-hop datagram protocol, and provides a unified communication interface to both the radio and the built-in serial port (for wired nodes such as basestations). Higher-level protocols providing multihop communication, larger ADUs, or other features are readily built on top of the AM interface. Variants of the basic AM stack exist that incorporate lightweight, link-level security (see Section 4.1). AM's event-driven nature and tight coupling of computation and communication make the abstraction well suited to the sensor network domain.

2.5 Implementation Status

TinyOS supports a wide range of hardware platforms and has been used on several generations of sensor motes. Supported processors include the Atmel AT90L-series, Atmel ATmega-series, and Texas Instruments MSP-series processors. TinyOS includes hardware support for the RFM TR1000 and Chipcon CC1000 radios, as well as as well as several custom radio chipsets. TinyOS applications may be compiled to run on any of these platforms without modification. Work is underway (by others) to port TinyOS to ARM, Intel 8051 and Hitachi processors and to support Bluetooth radios.

TinyOS supports an extensive development environment that incorporates visualization, debugging, and support tools as well as a fine-grained simulation environment. Desktops, laptops, and palmtops can serve as proxies between sensor networks and wired networks, allowing integration with server side tools implemented in Java, C, or MATLAB, as well as interfaces to database engines such as PostgreSQL. NesC includes a tool that generates code to marshal between Active Message packet formats and Java classes.

TinyOS includes TOSSIM, a high-fidelity mote simulator that compiles directly from TinyOS NesC code, scaling to thousands of simulated nodes. TOSSIM gives the programmer an omniscient view of the network and greater debugging capabilities. Server-side applications can connect to a TOSSIM proxy just as if it were a real sensor network, easing the transition between the simulation environment and actual deployments. TinyOS also provides JTAG support integrated with `gdb` for debugging applications directly on the mote.

3 Meeting the Four Key Requirements

In this section, we show how the design of TinyOS, particularly its component model and execution model, addresses our four key requirements: limited resources, reactive concurrency, flexibility and low power. This section quantifies basic aspects of resource usage and performance, including storage usage, execution overhead, observed concurrency, and effectiveness of whole-system optimization.

3.1 Limited Resources

We look at three metrics to evaluate whether TinyOS applications are lightweight in space and time: (1) the footprint of real applications should be small, (2) the compiler

Application	**Size**			**Structure**		
	Optimized	Unoptimized	Reduction	Tasks	Events	Modules
Blink	683	1791	61%	0	2	8
Blink LEDs						
GenericBase	4278	6208	31%	3	21	19
Radio-to-UART packet router						
CntToLeds	6121	9449	35%	1	7	13
Display counter on LEDs						
CntToRfm	9859	13969	29%	4	31	27
Send counter as radio packet						
Habitat monitoring	11415	19181	40%	9	38	32
Periodic environmental sampling						
Surge	14794	20645	22%	9	40	34
Ad-hoc multihop routing demo						
Maté	23741	25907	8%	15	51	39
Small virtual machine						
Object tracking	23525	37195	36%	15	39	32
Track object in sensor field						
TinyDB	63726	71269	10%	18	193	91
SQL-like query interface						

Fig. 6. Size and structure of selected TinyOS applications.

should reduce code size through optimization, and (3) the overhead for fine-grain modules should be low.

Absolute Size: A TinyOS program's component graph defines which components it needs to work. Because components are resolved at compile time, compiling an application builds an application-specific version of TinyOS: the resulting image contains exactly the required OS services.

As shown in Figure 6, TinyOS and its applications are small. The base TinyOS operating system is less than 400 bytes and associated C runtime primitives (including floating-point libraries) fit in just over 1KB. `Blink` represents the footprint for a minimal application using the base OS and a primitive hardware timer. `CntToLeds` incorporates a more sophisticated timer service which requires additional memory. `GenericBase` captures the footprint of the radio stack while `CntToRfm` incorporates both the radio stack and the generic timer, which is the case for many real applications. Most applications fit in less than 16KB, while the largest TinyOS application, TinyDB, fits in about 64KB.

Footprint Optimization: TinyOS goes beyond standard techniques to reduce code size (e.g., stripping the symbol table). It uses whole-program compilation to prune dead code, and cross-component optimizations remove redundant operations and module-crossing overhead. Figure 6 shows the reduction in size achieved by these optimizations on a range of applications. Size improvements range from 8% for Maté, to 40% for habitat monitoring, to over 60% for simple applications.

Component Overhead: To be efficient, TinyOS must minimize the overhead for module crossings. Since there are no virtual functions or address-space crossings,

Cycles	Optimized	Unoptimized	Reduction
Work	371	520	29%
Boundary crossing	109	258	57%
Non-interrupt	*8*	*194*	*95%*
Interrupt	*101*	*64*	*-36%*
Total	**480**	**778**	**38%**

Fig. 7. Optimization effects on clock event handling. *This figure shows the breakdown, in CPU cycles, for both work and boundary crossing for clock event handling, which requires 7 module crossings. Optimization reduces the overall cycle count by 38%.*

the basic boundary crossing is at most a regular procedure call. On Atmel-based platforms, this costs about eight clock cycles.

Using whole-program analysis, NesC removes many of these boundary crossings and optimizes entire call paths by applying extensive cross-component optimizations, including constant propagation and common subexpression elimination. For example, NesC can typically inline an entire component into its caller.

In the TinyOS timer component, triggering a timer event crosses seven component boundaries. Figure 7 shows cycle counts for this event chain with and without cross-component optimizations. The optimization saves not only 57% of the boundary overhead, but also 29% of the work, for a total savings of 38%. The increase in the crossing overhead for the interrupt occurs because the inlining requires the handler to save more registers; however, the total time spent in the handler goes down. The only remaining boundary crossing is the one for posting the task at the end of the handler.

Anecdotally, the code produced via whole-program optimization is smaller and faster than not only unoptimized code, but also the original hand-written C code that predates the NesC language.

3.2 Reactive Concurrency

We evaluate TinyOS's support for concurrency by looking at four metrics: (1) the concurrency exhibited by applications, (2) our support for race detection at compile time, (3) context switching times, and (4) the handling of concurrent events with real-time constraints.

Exhibited Concurrency: TinyOS's component model makes it simple to express the complex concurrent actions in sensor network applications. The sample applications in Figure 6 have an average of 8 tasks and 47 events, each of which represents a potentially concurrent activity. Moreover, these applications exhibit an average of 43% of the code (measured in bytes) reachable from an interrupt context.

As an example of a high-concurrency application, we consider TinyDB, covered in Section 5.3, an in-network query processing engine that allows users to pose queries that collect, combine and filter data from a network of sensors. TinyDB supports multiple concurrent queries, each of which collects data from sensors, applies some number of transformations, and sends it up a multihop routing tree to a basestation where the user receives results. The 18 tasks and 193 events within TinyDB

Component	**Type**	**Data-race variables**
`RandomLFSR`	System	1
`UARTM`	System	1
`AMStandard`	System	2
`AMPromiscious`	System	2
`BAPBaseM`	Application	2
`ChirpM`	Application	2
`MicaHighSpeedRadioM`	System	2
`TestTimerM`	Application	2
`ChannelMonC`	System	3
`NoCrcPacket`	System	3
`OscilloscopeM`	Application	3
`QueuedSend`	System	3
`SurgeM`	Application	3
`SenseLightToLogM`	Application	3
`TestTemp`	Application	3
`MultihopM`	System	10
`eepromM`	System	17
`TinyAlloc`	System	18
`IdentC`	Application	23
Total		103

Fig. 8. Component locations of race condition variables.

perform several concurrent operations, such as maintenance of the routing tables, multihop routing, time synchronization, sensor recalibration, in addition to the core functionality of sampling and processing sensor data.

Race Detection: The NesC compiler reports errors if shared variables may be involved in a data race. To evaluate race detection, we examine the reported errors for accuracy.

Initially, TinyOS included neither an explicit `atomic` statement nor the analysis to detect potential race conditions; both TinyOS and its applications had many data races. Once race detection was implemented, we applied detection to every application in the TinyOS source tree, finding 156 variables that potentially had a race condition. Of these, 53 were false positives (discussed below) and 103 were genuine data races, a frequency of about six per thousand code statements. We fixed each of these bugs by moving code into tasks or by using `atomic` statements. We then tested each application and verified that the presence of atomic sections did not interfere with correct operation.

Figure 8 shows the locations of data races in the TinyOS tree. Half of the races existed in system-level components used by many applications, while the other half were application specific. `MultihopM`, `eepromM`, and `TinyAlloc` had a disproportionate number of races due to the amount of internal state they maintain through complex concurrent operations. `IdentC` tracks node interactions, records them in flash, and periodically sends them to the basestation; it has complex concurrency, lots of state, and was written before most of the concurrency issues were well understood. The NesC version is race free.

The finite-state-machine style of decomposition in TinyOS led to the most common form of bug, a non-atomic state transition. State transitions are typically im-

```
/* Contains a race: */   /* Fixed version: */
if (state == IDLE) {     uint8_t oldState;
  state = SENDING;       atomic {
  count++;                 oldState = state;
  // send a packet         if (state == IDLE) {
}                            state = SENDING;
                           }
                         }
                         if (oldState == IDLE) {
                           count++;
                           // send a packet
                         }
```

Fig. 9. Fixing a race condition in a state transition.

plemented using a read-modify-write of the state variable, which must be atomic. A canonical example of this race is shown in Figure 9, along with the fix.

The original versions of the communication, `TinyAlloc` and EEPROM components contained large numbers of variable accesses in asynchronous code. Rather than using large atomic sections, which might decrease overall responsiveness, we promoted many of the offending functions to synchronous code by posting a few additional tasks.

False positives fell into three major categories: state-based guards, buffer swaps, and causal relationships. The first class, state-based guards, occurred when access to a module variable is serialized at run time by a state variable. The above state transition example illustrates this; in this function, the variable `count` is safe due to the monitor created by `state`. Buffer swaps are a controlled kind of sharing in which ownership is passed between producer and consumer; it is merely by this convention that there are no races, so it is in fact useful that NesC requires the programmer to check them. The third class of false positives occurs when an event conflicts with the code that caused it to execute, but because the two never overlap in time there is no race. However, if there are other causes for the event, then there is a race, so these are also worth explicitly checking. In all cases, the `norace` type qualifier can be used to remove the warnings.

Context Switches: In TinyOS, context switch overhead corresponds to both the cost of task scheduling and interrupt handler overhead. These costs are shown in Figure 10 based on hand counts and empirical measurements. The interrupt overhead consists of both switching overhead and function overhead of the handler, which varies with the number of saved registers.

Overhead	**Time** (clock cycles)
Interrupt Switching	8
Interrupt Handler Cost	26-74
Task Switching	108

Fig. 10. TinyOS scheduling overhead.

Real-time Constraints: The real-time requirements in the sensor network domain are quite different from those traditionally addressed in multimedia and control applications. Rather than sophisticated scheduling to shed load when many tasks are ongoing, sensor nodes exhibit bursts of activity and then go idle for lengthy intervals. Rather than delivering a constant bit rate to each of many flows, we must meet hard deadlines in servicing the radio channel while processing sensor data and routing traffic. Our initial platforms required that we modulate the radio channel bit-by-bit in software. This required tight timing on the transmitter to generate a clean waveform and on the receiver to sample each bit properly. More recent platforms provide greater hardware support for spooling bits, but start-symbol detection requires precise timing and encoding, decoding, and error-checking must keep pace with the data rate. Our approach of allowing sophisticated handlers has proven sufficient for meeting these requirements; typically the handler performs the time-critical work and posts a task for any remaining work. With a very simple scheduler, allowing the handler to execute snippets of processing up the chain of components allows applications to schedule around a set of deadlines directly, rather than trying to coerce a priority scheme to produce the correct ordering. More critical is the need to manage the contention between the sequence of events associated with communication (the handler) and the sampling interval of the application (the tasks). Applying whole-system analysis to verify that all such jitter bounds are met is an area for future work.

3.3 Flexibility

To evaluate the goal of flexibility, we primarily refer to anecdotal evidence. In addition to the quantitative goal of fine-grain components, we look at the qualitative goals of supporting concurrent components, hardware/software transparency, and interposition.

Fine-grained Components: TinyOS allows applications to be constructed from a large number of very fine-grained components. This approach is facilitated by cross-module inlining, which avoids runtime overhead for component composition. The TinyOS codebase consists of 401 components, of which 235 are modules and 166 are configurations. The 42 applications in the tree use an average of 74 components (modules and configurations) each. Modules are typically small, ranging from between 7 and 1898 lines of code (with an average of 134, median of 81).

Figure 11 shows a per-component breakdown of the data and code space used by each of the components in the TinyOS radio stack, both with and without inlining applied. The figure shows the relatively small size of each of the components, as well as the large number of components involved in radio communication. Each of these components can be selectively replaced, or new components interposed within the stack, to implement new functionality.

Concurrent Components: As discussed in the previous section, any component can be the source of concurrency. Bidirectional interfaces and explicit support for events enable any component to generate events autonomously. In addition, the static

Component	**Code Size**		**Data Size**
(Sizes in bytes)	*inlined*	*noninlined*	
`AM` *Core Active Messages layer*	456	654	9
`MicaHighSpeedRadioM` *Radio hardware interface*	1162	1250	61
`NoCRCPacket` *Packet framing without CRC*	370	484	50
`CrcFilter` *CRC filtering*	–	34	0
`ChannelMonC` *Start symbol detection*	454	486	9
`RadioTimingC` *Timing for start symbol detection*	42	56	0
`PotM` *Transmit power control*	50	82	1
`SecDedEncoding` *Error correction/detection coding*	662	684	3
`SpiByteFifoC` *Low-level byte interface*	344	438	2
`HPLPotC` *Hardware potentiometer interface*	–	66	0

Fig. 11. Breakdown of code and data size by component in the TinyOS radio stack. *A '–' in the* inlined *column indicates that the corresponding component was entirely inlined. Dead code elimination has been applied in both cases.*

race detection provided by NesC removes the need to worry about concurrency bugs during composition. Out of our current set of 235 modules, 18 (7.6%) contain at least one interrupt handler and are thereby sources of concurrency.

Hardware/Software Transparency: The TinyOS component model makes shifting the hardware/software boundary easy; components can generate events, which may be software upcalls or hardware interrupts. This feature is used in several ways in the TinyOS codebase. Several hardware interfaces (such as analog-to-digital conversion) are implemented using software wrappers that abstract the complexity of initializing and collecting data from a given sensor hardware component. In other cases, software components (such as radio start-symbol detection) have been supplanted with specialized hardware modules. For example, each of the radios we support has a different hardware/software boundary, but the *same* component structure.

Interposition: One aspect of flexibility is the ability to *interpose* components between other components. Whenever a component provides and uses the same interface type, it can be inserted or removed transparently.

One example of this is seen in work at UVA [26], which interposes a component in the network stack at a fairly low level. Unknown to the applications, this component buffers the payload of each message and aggregates messages to the same destination into a single packet. On the receive side, the same component decomposes such packets and passes them up to the recipients individually. Although remaining completely transparent to the application, this scheme can actually *decrease* network latency by increasing overall bandwidth.

A similar type of interpositioning can be seen in the object tracking application described in Section 5.2. The routing stack allows the interpositioning of components that enable, for example, reliable transmission or duplicate message filtering. Similarly, the sensor stacks allow the interpositioning of components that implement weighted-time averaging or threshold detection.

3.4 Low Power

The application-specific nature of TinyOS ensures that no unnecessary functions consume energy, which is the most precious resource on the node. However, this aspect alone does not ensure low power operation. We examine three aspects of TinyOS low power operation support: application-transparent CPU power management, power management interfaces, and efficiency gains arising from hardware/software transparency.

CPU power usage: The use of split-phase operations and an event-driven execution model reduces power usage by avoiding spinlocks and heavyweight concurrency (e.g., threads). To minimize CPU usage, the TinyOS scheduler puts the processor into a low-power sleep mode whenever the task queue is empty. This decision can be made very quickly, thanks to run-to-completion semantics of tasks, which maximizes the time spent in the sleep mode. For example, when listening for incoming packets, the CPU handles 20000 interrupts per second. On the current sensor hardware, the CPU consumes 4.6 mA when active and 2.4 mA when idle, and the radio uses 3.9 mA when receiving. System measurements show the power consumption during both listening and receiving to be 7.5 mA. The scheduler, which needs to examine the task queue after every event, still manages to operate in idle mode 44% of the time.

Power-Management Interfaces: The scheduler alone cannot achieve the power levels required for long-term applications; the application needs to convey its runtime requirements to the system. TinyOS address this requirement through a programming convention which allows subsystems to be put in a low power idle state. Components expose a `StdControl` interface, which includes commands for initializing, starting, and stopping a component and the subcomponents it depends upon. Calling the `stop` command causes a component to attempt to minimize its power consumption, for example, by powering down hardware or disabling periodic tasks. The component saves its state in RAM or in nonvolatile memory for later resumption using the `start` command. It also informs the CPU about the change in the resources it uses; the system then uses this information to decide whether deep power saving modes should be used. This strategy works well: with all components stopped, the base system without the sensor board consumes less than 15 μA, which is comparable to self discharge rate of AA alkaline batteries. The node lifetime depends primarily on the duty cycle and the application requirements; a pair of AA batteries can power a constantly active node for up to 15 days or a permanently idle node for up to 5 years (battery shelf life). By exposing the start/stop interface at many levels, we enable a range of power management schemes to be implemented, for example, using power scheduling to disable the radio stack when no communication is expected, or powering down sensors when not in use.

Hardware/Software Transparency: The ability to replace software components with efficient hardware implementations has been exploited to yield significant improvements in energy consumption in our platform. Recent work [36] has demonstrated a single-chip mote that integrates the microcontroller, memory, radio transceiver, and radio acceleration logic into a 5 mm^2 silicon die. The standard software radio stack consumes 3.6 mA (involving about 2 million CPU instructions per second); The hardware implementation of these software components consumes less than 100 μA and allows for much more efficient use of microcontroller sleep modes while providing a 25-fold improvement in communication bit rate.

4 Enabled Innovations

A primary goal for TinyOS is to enable innovative solutions to the systems challenges presented by networks of resource constrained devices that interact with a changing physical world. The evaluation against this goal is inherently qualitative. We describe three subsystems where novel approaches have been adopted that can be directly related to the features of TinyOS. In particular, TinyOS makes several kinds of innovations simpler that appear in these examples: 1) cross-layer optimization and integrated-layer processing (ILP), 2) duty-cycle management for low power, and 3) a wide-range of implementation via fine-grain modularity.

4.1 Radio Stack

A mote's network device is often a simple, low-power radio transceiver that has little or no data buffering and exposes primitive control and raw bit interfaces. This requires handling many aspects of the radio in software, such as controlling the radio state, coding, modulating the channel, framing, input sampling, media access control, and checksum processing. Various kinds of hardware acceleration may be provided for each of the elements, depending on the specific platform. In addition, received signal strength can be obtained by sampling the baseband energy level at particular times. The ability to access these various aspects of the radio creates opportunities for unusual cross-layer optimization.

Integrated-Layer Processing: TinyOS enables ILP through its combination of fine-grain modularity, whole-program optimization, and application-specific handlers. One example is the support for link-layer acknowledgments (acks), which can only be generated after the checksum has been computed. TinyOS allows the radio stack to be augmented with addition error checking by simply interposing the checksum component between the component providing byte-by-byte radio spooling and the packet processing component. It is also important to be able to provide link-level acknowledgments so that higher levels can estimate loss rates or implement retransmission, however, these acks should be very efficient. The event protocol within the stack that was developed to avoid buffering at each level allows the checksum computation to interleave with the byte-level spooling. Thus, the ack can be generated immediately after receiving the last byte thus the underlying radio component can send the ack *synchronously*, i.e. reversing the channel direction without re-arbitration or reacquisition. Note that holding the channel is a real-time operation that is enabled by the use of sophisticated handlers that traverse multiple layers and components without data races. This collection of optimizations greatly reduce both latency and power, and in turn allows shorter timeouts at the sender. Clean modularity is preserved in the code since these time-critical paths span multiple components.

ILP and flexible modularity have been used in a similar manner to provide flexible security for confidentiality and authentication [2]. Although link-level security is important, it can degrade both power and latency. The ability to overlap computation via ILP helps with the latency, while interposition makes it easy add security transparently as needed. This work also showed that the mechanisms for avoiding copying or gather/scatter within the stack could be used to substantially modify packet headers and trailers without changing other components in the stack.

A TinyOS radio stack from Ye *et al.* [83, 84] is an example that demonstrates ILP by combining 802.11-style media access with transmission scheduling. This allows a low-duty cycle (similar to TDMA) with flexible channel sharing.

Power Management: Listening on the radio is costly even when not receiving anything, so minimizing duty cycle is important. Traditional solutions utilize some form of TDMA to turn off the radio for long periods until a reception is likely. TinyOS allows a novel alternative by supporting fast fine-grain power management. By integrating fast power management with precise timing, we were able to periodically sample the radio for very short intervals at the physical layer, looking for a preamble. This yields the illusion of an always-on radio at a 10% duty cycle while listening, while avoiding a priori partitioning of the channel bandwidth. Coarse-grain duty cycling can still be implemented at higher levels, if needed.

TinyOS has also enabled an efficient solution to the epidemic wakeup problem. Since functionality can be placed at different levels within the radio stack, TinyOS can detect that a wakeup is likely by sampling the energy on the channel, rather than bring up the ability to actually receive packets. This low-level wake-up only requires 0.00125% duty cycle [29], a 400-fold improvement over a typical packet-level protocol. A similar approach has been used to derive network neighborhood and proximity information [73].

Hardware/Software Transparency: The existence of a variety of radio architectures poses a challenge for system designers due to the wide variation in hardware/software boundaries. There are at least three radio platforms that are supported in the TinyOS distribution: the 10kbps first-generation RFM, the 40kbps hardware-accelerated RFM, and the recent 40kbps Chipcon. In addition, UART and I2C stacks are supported. The hardware-accelerated RFM platform exemplifies how a direct replacement of bit level processing with hardware achieves higher communication bandwidth [29]. In the extreme cases, the entire radio stack has been built in pure hardware in Spec (mote-on-a-chip) [36], as well as in pure software in TOSSIM [44]. We have also transparently used hardware acceleration for encryption. Stack elements using a component remain unchanged, whether the component is a thin abstraction of a hardware element or a software implementation.

4.2 Time Synchronization and Ranging

Time and location are both critical in sensor networks due to the embodied nature of sensor nodes; each node has a real, physical relationship with the outside world. One challenge of network time synchronization is to eliminate sources of jitter such as media access delay introduced by the radio stack. Traditional layering often hides the details at the physical layer. Timing protocols often perform round-trip time estimation to account for these errors. TinyOS allows a component to be interposed deep within the radio stack to signal an event precisely when the first bit of data is transmitted; this eliminates media access delay from calculations. Similarly, receivers can take a timestamp when they hear the first data bit; comparing these fine-grain timestamps can reduce time synchronization error to less than a bit time ($<25\mu$s). Although reference broadcast synchronization (RBS) [16] achieves synchronization accurate to within 4μs without interposition by comparing time stamps of receivers, it does so at the cost of many packet transmissions and sophisticated analysis.

The ability to interact with the network stack at this low level also enabled precise time of flight (TOF) measurements for ranging in an ad-hoc localization system built on TinyOS [76]. A transmitter sends an acoustic pulse with a radio message. TinyOS's low context switching overhead enables receivers to check for the acoustic pulse and the radio message concurrently. Taking the difference between the timestamps of the two signals produces an acoustic TOF measurement. TinyOS can accurately measure both arrival times directly in their event handlers, since the handlers execute immediately; a solution based on queuing the work for later would forfeit precise timing, which is also true for the time-syncrhonization example above.

The newest version of the ranging application uses a co-processor to control the acoustic transducer and perform costly localization calculation. Controlling the acoustic transducer requires real time interactions between the two processors which is enabled by TinyOS's low overhead event handling. To exploit parallelism between the two processors, computation and communication must be overlapped; the split-phased nature of TinyOS's AM model makes this trivial.

4.3 Routing

The rigid, non-application specific communication stack found in industrial standards such as IEEE 802.11 [1] or Bluetooth [7] often limit the design space for routing protocols. TinyOS's component model and ease of interposition yield a very flexible communication stack. This opens up a platform for implementing many different routing protocols such as broadcast based routing [23], probabilistic routing, multipath routing [37], geographical routing, reliability based routing [80, 82], TDMA based routing [14], and directed diffusion [34].

The large number of routing protocols suggests that sensor network applications may need to use a diverse set within one communication stack. TinyOS's parameterized interfaces and extensible component model enable a coherent routing framework where an application can route by network address, geographic location, flooding, or along some application specific gradients [69].

4.4 Dynamic Composition and Virtual Machines

In our experience, most sensor network applications utilize a common set of services, combined in different ways. A system that allows these compositions to be concisely described could provide much of the flexibility of full reprogramming at a tremendous decrease in communication costs. Maté, a tiny bytecode interpreter that runs on TinyOS [43], meets this need. It is a single NesC module that sits on top of several system components, including sensors, the network stack, and non-volatile storage.

Maté presents a virtual stack architecture to the programmer. Instructions include sensing and radio communication, as well as arithmetic and stack manipulation. Maté has a set of user-definable instructions. These allow developers to use the VM as a framework for writing new VM variants, extending the set of TinyOS services that can be dynamically composed. The virtual architecture hides the split-phased operations of TinyOS behind synchronous instructions, simplifying the programming interface. This requires the VM to maintain a virtual execution context as a continuation across split-phase operations. The stack-based architecture makes virtual context switches trivial, and as contexts are only 78 bytes (statically allocated in a component), they consume few system resources. Contexts run in response to system events, such as timers or packet reception.

Programs virally propagate through a network; once a user introduces a single mote running a new program, the network rapidly and autonomously reprograms itself. Maté programs are extremely concise (orders of magnitude shorter than their binary equivalents), conserving communication energy. TinyOS' event-driven execution provides a clear set of program-triggering events, and the NesC's interfaces allow users to easily change subsystems (such as ad-hoc routing). Maté extends TinyOS by providing an inexpensive mechanism to dynamically compose programs. NesC's static nature allows it to produce highly optimized and efficient codes; Maté demonstrates that run-time flexibility can be re-introduced quite easily with low overhead. By eschewing aside the traditional user/kernel boundary, TinyOS allowed other possibilities to emerge. Maté suggests that the run-time/compile-time boundary in sensor networks might better be served by a lean bytecode interpreter that sits on top of a TinyOS substrate.

5 Applications

In this section, we describe three applications that have been built using the TinyOS platform: an environmental monitoring system, a declarative query processor, and magnetometer-based object tracking. Each of these applications represents a distinct set of design goals and exhibits different aspects of the TinyOS design.

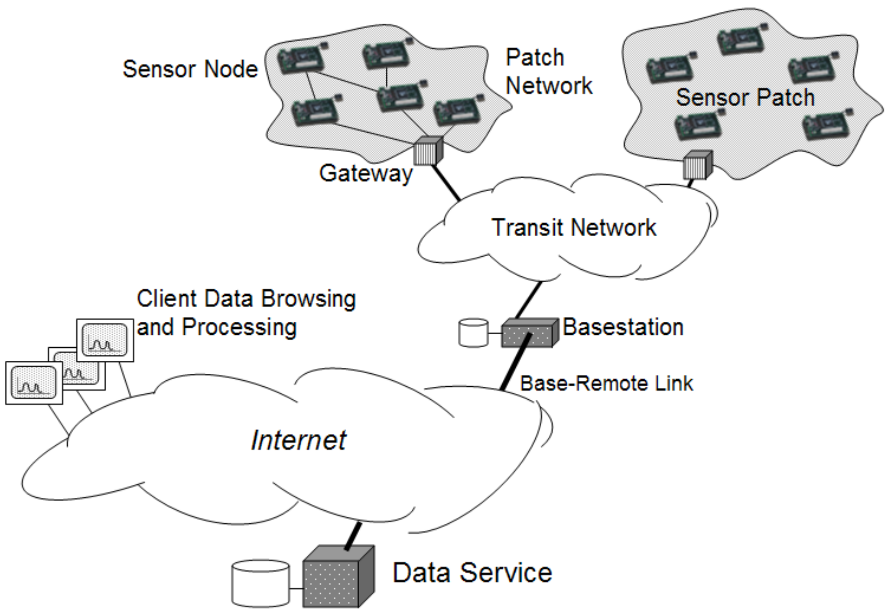


Fig. 12. System architecture for habitat monitoring.

5.1 Habitat Monitoring

Sensor networks enable data collection at a scale and resolution that was previously unattainable, opening up many new areas of study for scientists. These applications pose many challenges, including low-power operation and robustness, due to remote placement and extended operation.

One such application is a habitat monitoring system on Great Duck Island, off the coast of Maine. Researchers deployed a 35-node network on the island to monitor the presence of Leach's Storm Petrels in their underground burrows [51]. The network was designed to run unattended for at least one field season (7–9 months). Nodes, placed in burrows, monitored light, temperature, relative humidity, pressure, and passive infrared; the network relayed readings back to a base station with an Internet connection via satellite, to be uploaded to a database. Figure 12 illustrates the tiered system architecture for this application.

A simple TinyOS program ran on the motes. It periodically (every 68 s) sampled sensors and relayed data to the base-station. To achieve long network lifetimes, nodes used the power management facilities of TinyOS aggressively, consuming only 35 μA in low power state, compared to 18–20 mA when active. Nodes sampled sensors concurrently (using a split-phase data acquisition operation), rather than serially, resulting in further power reduction. During the 4 months of deployment, the network collected over 1.2 million sensor readings.

A specialized gateway node, built using a mote connected to a high-gain antenna, relayed data from the network to a wired base station. The gateway application was very small (3090 bytes) and extraordinarily robust: it ran continuously, without failing, for the entire 4 months of deployment. The gateway required just 2 Watt-hours of energy per day and was recharged with a 36 in^2 solar panel [63]. In comparison, an early prototype version of the gateway, an embedded Linux system, required over 60 Watt-hours of energy per day from a 924 in^2 solar panel. The Linux system failed every 2 to 4 days, while the gateway mote was still operating two months after researchers lost access to the island for the winter.

5.2 Object Tracking

The TinyOS object-tracking application (OTA) uses a sensor network to detect, localize and track an object moving through a sensor field; in the prototype, the object is a remote-controlled car. The object's movement through the field determines the actions and communication of the motes. Each mote periodically samples its magnetometer; if the reading has changed significantly since the last sample, it broadcasts the reading to its neighbors. The node with the largest reading change estimates the position of the target by computing the centroid of its neighbors' readings. Using geographic routing [38], the network routes the estimated position to the base-station, which controls a camera to point at the target. The operation of the tracking application is shown in Figure 13.

OTA consists of several distributed services, such as routing, data sharing, time synchronization, localization, power management, and sensor filtering. Twelve different research groups are collaborating on both the architecture and individual subsystem implementation. TinyOS execution model enables running these services concurrently on limited hardware resources. The component model allows for easy replacement and comparative analysis of individual services. Currently, the reference implementation consists of 54 components. General purpose services, such as time synchronization or localization, have many competing implementations, enabled by different features of TinyOS. Replacement of low-level components used for sensing allowed OTA to be adapted to track using light values instead of magnetic fields.

Several research groups have successfully implemented application specific services within this framework. Hui *et al.* developed a sentry-based approach [31] that addresses power management within an object tracking network. Their algorithm chooses a connected subset of sentry motes, which allows for degraded sensing; the non-sentry units are placed in a low power state. This service makes extensive use of the TinyOS power management interfaces, and is shown to reduce energy consumption by 30% with minimal degradation of tracking accuracy.

5.3 TinyDB

Many sensor network users prefer to interact with a network through a high-level, declarative interface rather than by low-level programming of individual nodes. TinyDB [50], a declarative query processor built on TinyOS, supports this view, and

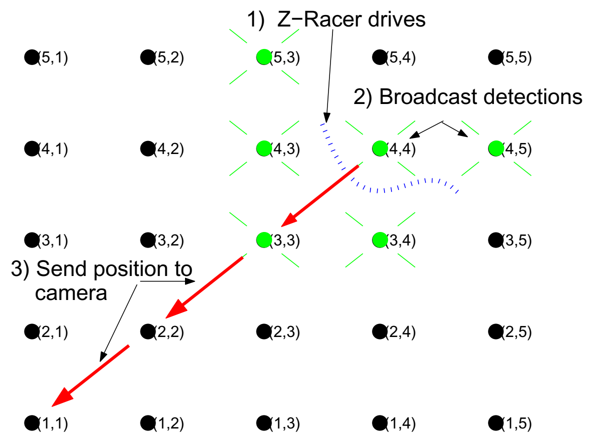


Fig. 13. Event-triggered activity in the object tracking application. *(1) The vehicle being tracked drives around position (4,4) (dashed-line); (2) Six nodes broadcast readings (lightened nodes); (3) Node (4,4) declares itself the leader, aggregates the readings, and routes them to the base station (dark arrows).*

is our largest and most complex application to date. It poses significant challenges for concurrency control and limited resources.

In TinyDB, queries (expressed in an SQL-like syntax) propagate through the network and perform local data collection and in-network aggregation. Queries specify only what data the user is interested in and the data collection rate; the user does not specify any details of query propagation, data collection, or message routing. For example, the query:

```
SELECT AVG(light)
  FROM sensors
  WHERE temp > 100° F
  SAMPLE PERIOD 10s
```

tells the network to provide the average light value over all the nodes with temperature greater than 100^o F once every 10 seconds. TinyDB uses in-network aggregation [42, 49] to greatly reduce network bandwidth requirements; this requires that nodes coordinate to produce the results.

TinyDB relies heavily on TinyOS'component-oriented design, concurrency primitives, and ability to perform cross-layer optimizations. TinyDB consists of components that perform query flooding, local data collection, formation of routing trees, aggregation of query data, and a catalog of available sensor devices and attributes (such as location) at each node. It uses the routing, data collection, and power management interfaces of TinyOS, and inter-operates with a variety of implementations of these services.

TinyOS's task model meshes well with the concurrency requirements of TinyDB, which supports multiple simultaneous queries by scheduling a timer for each query which fires when the next set of results for that query are due. Each timer event posts a task to collect and deliver results for the corresponding query. The non-preemptive nature of tasks and the support for safe concurrent handlers avoid data races despite extensive information sharing.

One example benefit of cross-layer optimization in TinyDB is message snooping, which is important for determining the state of neighboring nodes in the network. Snooping is used to enable query propagation: new nodes joining the network learn of ongoing queries by snooping for results broadcast by neighbors. This technique also enables message suppression; a node can avoid sending its local reading if it is superseded by a message from another node, as in the case of a query requesting the maximum sensor value in the network.

6 Related Work

Sensor networks have been the basis for work on *ad hoc* networking [34, 37, 38, 47], data aggregation [33, 49], distributed algorithms [25, 46, 59], and primitives such as localization [8, 76, 77], and time synchronization [16, 62]. In addition to our mote platform, a number of low-power sensor systems have been proposed and developed [3, 4, 12, 39, 55, 56, 64], though few of these systems have addressed flexible operating systems design. Several projects use more traditional embedded systems (such as PDAs [16]) or customized hardware [64].

A wide range of operating systems have been developed for embedded systems. These range from relatively large, general-purpose systems to more compact real-time executives. In [30] we discuss range of these embedded and real-time systems in detail. These systems are generally not suitable for extremely resource-constrained sensor nodes, which mandate very compact, specialized OS designs. Here, we focus our attention on a number of emerging systems that more closely match the resource budget and execution model of sensor networks.

Traditional embedded operating systems are typically large (requiring hundreds of KB or more of memory), general-purpose systems consisting of a binary kernel with a rich set of programming interfaces. Examples include WinCE [52], QNX [28], PalmOS [60], pSOSystem [79], Neutrino [65], OS-9 [54], LynxOS [48], Symbian [71], and uClinux [72]. Such OSes target systems with greater CPU and memory resources than sensor network nodes, and generally support features such as full multitasking, memory protection, TCP/IP networking, and POSIX-standard APIs that are undesirable (both in terms of overhead and generality) for sensor network nodes.

There is also a family of smaller real-time executives, such as CREEM [40], OSEKWorks [78], and Ariel [53], that are closer in size to TinyOS. These systems support a very restrictive programming model which is tailored for specialized application domains such as consumer devices and automotive control.

Several other small kernels have been developed that share some features in common with TinyOS. These systems do not support the degree of modularity or flexibility in TinyOS's design, nor have they been used for as wide a range of applications. EMERALDS [85] is a real-time microkernel, requiring about 13KB of code, that supports multitasking using a hybrid EDF and rate-monotonic scheduler. Much of this work is concerned with reducing overheads for semaphores and IPC. AvrX [5] is a small kernel for the AVR processor, written in assembly, that provides multitasking, semaphores, and message queues in around 1.5 KB of memory. Nut/OS [15] and NESOS [58] are small kernels that provide non-preemptive multitasking, similar in vein to the TinyOS task model, but use somewhat more expensive mechanisms for interprocess communication than TinyOS's lean cross-module calls. The BTNode OS [39] consists mainly of library routines to interface to hardware and a Bluetooth communication stack, but supports an event-driven programming model akin to TinyOS. Modules can post a single-byte event to a dispatcher, which fires the (single) handler registered for that event type.

A number of operating systems have explored the use of component architectures. Click [41], Scout [57], and the x-kernel [32] are classic examples of modular systems, but do not address the specific needs of low-power, low-resource embedded systems. The units [19] component model, supported by the Knit [67] language in OSKit [20], is similar to that in NesC. In Knit, components provide and use interfaces, and new components can be assembled out of existing ones. Unlike NesC, however, Knit lacks bidirectional interfaces and static analyses such as data race detection.

Several embedded systems have taken a component-oriented approach for application-specific configurability [21]. Many of these systems use heavyweight composition mechanisms, such as COM or CORBA, and several support runtime component instantiation or interpositioning. PURE [6], eCos [66], and icWORKSHOP [35] more closely match TinyOS's goal of lightweight, static composition. These systems consist of a set of components that are wired together (either manually or using a composition tool) to form an application. Components vary in size from fine-grained, specialized objects (as in icWORKSHOP) to larger classes and packages (PURE and eCos). VEST [70] is a proposed toolkit for building component-based embedded systems that performs extensive static analyses of the system, such as schedulability, resource dependencies, and interface type-checking.

7 Discussion, Future Work, and Conclusion

Sensor networks present a novel set of systems challenges, due to their need to react to the physical environment, to let nodes asynchronously communicate within austere resource constraints, and to operate under a very tight energy budget. Moreover, the hardware architectures in this new area are changing rapidly. When we began designing an operating system for sensor nets we believed that the layers and boundaries that have solidified over the years from mainframes to laptops were unlikely to be ideal. Thus, we focused on building a framework for experimenting with a variety of system designs so that the proper boundaries could emerge with time. The key elements being a rich component approach with bidirectional interfaces and encapsulated tasks, pervasive use of event-based concurrency, and whole-system analysis and optimization. It has been surprising just how varied those innovations are.

Reflecting on the experience to date, the TinyOS' component approach has worked well. Components see a great deal of re-use and are generally defined with narrow yet powerful interfaces. NesC's optimizations allow developers to use many fine-grained components with little penalty. This has facilitated experimentation, even with core subsystems, such as the networking stack. Some developers experience initial frustration with the overhead of building components with a closed namespace, rather than just calling library routines, but this is compensated by the ease of interpositioning, which allows them to introduce simple extensions with minimal overhead.

The resource-constrained event-driven concurrency model has been remarkably expressive and remains almost unchanged from the first version of the OS. We chose the task/event distinction because of its simplicity and modest storage demands, fully expecting that something more sophisticated might be needed in the future. Instead, it has been able to express the degree of concurrency required for a wide range of applications. However, the mechanics of the approach have evolved considerably. Earlier versions of TinyOS made no distinction between asynchronous and synchronous code and provided inadequate support for eliminating race conditions, many of which were exceedingly difficult to find experimentally. At one point, we tried introducing a hard boundary to AC, so all "user" processing would be in tasks. This made it impossible to meet the real-time requirements of the network stack, and the ability to perform a carefully designed bit of processing within the handler was sorely missed. The framework for innovation concept led us to better support for building (via atomic sections) the low-level concurrent data structures that cleanly integrate information from the asynchronous external world up into local processing. This particularly true for low-level real-time operations that cannot be achieved without sophisticated handlers.

TinyOS differs strongly from most event-driven embedded systems in that concurrency is structured into modular components, instead of a monolithic dispatch constructed with global understanding of the application. Not only has this eased the conceptual burden of managing the concurrency, it has led to important software protocols between components, such as split-phase data acquisition, data-pumps found between components in the network stack, and a power-management idiom that allows hardware elements to be powered-down quickly and easily. In a number of cases, attention to these protocols provided the benefits of integrated-layer processing while preserving clean modularity.

TinyOS is by no means a finished system; it continues to evolve and grow. The use of language tools for whole-system optimization is very promising and should be taken further. Currently, components follow implicit software protocols; making these protocols explicit entities would allow the compiler to verify that components are being properly used. Examples of these protocols include the buffer-swapping semantics of the networking stack and the state sequencing in the control protocols. Parallels exist between our needs and work such as Vault [13] and MC [17].

Richer means of expressing composition are desirable. For instance, while developing a routing architecture, we found that layers in the stack required significant self-consistency and redundancy in their specifications. A simple example is the definition of header fields when multiple layers of encapsulation are provided in the network stack. We have explored *template wiring*, which defines a skeleton structure, behaviors of composition, and naming conventions into which stackable components can be inserted. A template wiring produces a set of modules and configurations that meet the specification; it merges component composition and creation into a single step. We expect to incorporate these higher-level models of composition into NesC and TinyOS as they become more clear and well defined.

We continue to actively develop and deploy sensor network applications; many of our design decisions have been based on our and other users' experiences with these systems in the field. Sensor networks are still a new domain, filled with unknowns and uncertainties. TinyOS provides an efficient, flexible platform for developing sensor network algorithms, systems, and full applications. It has enabled innovation and experimentation on a wide range of scale.

References

1. ANSI/IEEE Std 802.11 1999 Edition.
2. TinySec: Link Layer Security for Tiny Devices. `http://www.cs.berkeley.edu/~nks/tinysec/`.
3. G. Asada, M. Dong, T. Lin, F. Newberg, G. Pottie, W. Kaiser, and H. Marcy. Wireless integrated network sensors: Low power systems on a chip. 1998.
4. B. Atwood, B. Warneke, and K. S. Pister. Preliminary circuits for smart dust. In *Proceedings of the 2000 Southwest Symposium on Mixed-Signal Design*, San Diego, California, February 27-29 2000.
5. L. Barello. Avrx real time kernel. `http://www.barello.net/avrx/`.
6. D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. The PURE family of object-oriented operating systems for deeply embedded systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 1999.
7. Bluetooth SIG, Inc. `http://www.bluetooth.org`.
8. N. Bulusu, V. Bychkovskiy, D. Estrin, and J. Heidemann. Scalable, ad hoc deployable, rf-based localization. In *Proceedings of the Grace Hopper Conference on Celebration of Women in Computing*, Vancouver, Canada, October 2002.
9. D. W. Carman, P. S. Kruus, and B. J. Matt. Constraints and approaches for distributed sensor network security. *NAI Labs Technical Report #00-010*, September 2000.
10. Center for Information Technology Research in the Interest of Society. Smart buildings admit their faults. `http://www.citris.berkeley.edu/applications/disaster_response/smartbuil%dings.html`, 2002.
11. A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao. Habitat monitoring: Application driver for wireless communications technology. In *Proceedings of the Workshop on Data Communications in Latin America and the Caribbean*, Apr. 2001.
12. L. P. Clare, G. Pottie, and J. R. Agre. Self-organizing distributed microsensor networks. In *SPIE 13th Annual International Symposium on Aerospace/Defense Sensing, Simulation, and Controls (AeroSense), Unattended Ground Sensor Technologies and Applications Conference*, Apr. 1999.

13. R. Deline and M. Fahndrich. Enforcing High-level Protocols in Low-Level Software. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, June 2001.
14. L. Doherty, B. Hohlt, E. Brewer, and K. Pister. SLACKER. `http://www-bsac.eecs.berkeley.edu/projects/ivy/`.
15. egnite Software GmbH. Nut/OS. `http://www.ethernut.de/en/software.html`.
16. J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, USA., dec 2002.
17. D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation.*, Oct. 2000.
18. D. Estrin et al. *Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers*. National Acedemy Press, Washington, DC, USA, 2001.
19. M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 236–248, 1998.
20. B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The flux OSKit: A substrate for kernel and language research. In *Symposium on Operating Systems Principles*, pages 38–51, 1997.
21. L. F. Friedrich, J. Stankovic, M. Humphrey, M. Marley, and J. J.W. Haskins. A survey of configurable component-based operating systems for embedded applications. *IEEE Micro*, May 2001.
22. D. Ganesan. TinyDiffusion Application Programmer's Interface API 0.1. `http://www.isi.edu/scadds/papers/tinydiffusion-v0.1.pdf`.
23. 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. `citeseer.nj.nec.com/ganesan02empirical.html`, 2002. Submitted for publication, February 2002.
24. 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 *Proceedings of Programming Language Design and Implementation (PLDI)*, June 2003.
25. I. Gupta and K. Birman. Holistic operations in large-scale sensor network systems: A probabilistic peer-to-peer approach. In *Proceedings of International Workshop on Future Directions in Distributed Computing (FuDiCo)*, June 2002.
26. T. Ha, B. Blum, J. Stankovic, and T. Abdelzaher. AIDA: Application Independant Data Aggregation in Wireless Sensor Networks. Submitted to *Special Issue of* ACM TECS, January 2003.
27. J. S. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.
28. D. Hildebrand. An Architectural Overview of QNX. `http://www.qnx.com/literature/whitepapers/archoverview.html`.
29. J. Hill and D. E. Culler. Mica: a wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, nov/dec 2002.
30. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Boston, MA, USA, Nov. 2000.

31. J. Hui, Z. Ren, and B. H. Krogh. Sentry-based power management in wireless sensor networks. In *Proceedings of Second International Workshop on Information Processing in Sensor Networks (IPSN '03)*, Palo Alto, CA, USA, Apr. 2003.
32. N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.
33. C. Intanagonwiwat, D. Estrin, R. Govindan, and J. Heidemann. Impact of network density on data aggregation in wireless sensor networks. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, July 2002.
34. C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the International Conference on Mobile Computing and Networking*, Aug. 2000.
35. Integrated Chipware, Inc. Integrated Chipware icWORKSHOP. `http://www.chipware.com/`.
36. Jason Hill. Integrated μ-wireless communication platform. `http://webs.cs.berkeley.edu/retreat-1-03/slides/Mote_Chip_Jhill_Nest_jan2003.pdf`.
37. C. Karlof, Y. Li, and J. Polastre. ARRIVE: Algorithm for Robust Routing in Volatile Environments. Technical Report UCB//CSD-03-1233, University of California at Berkeley, Berkeley, CA, Mar. 2003.
38. B. Karp and H. T. Kung. GPSR: greedy perimeter stateless routing for wireless networks. In *International Conference on Mobile Computing and Networking (MobiCom 2000)*, pages 243–254, Boston, MA, USA, 2000.
39. O. Kasten and J. Beutel. BTnode rev2.2. `http://www.inf.ethz.ch/vs/res/proj/smart-its/btnode.html`.
40. B. Kauler. CREEM Concurrent Realitme Embedded Executive for Microcontrollers. `http://www.goofee.com/creem.htm`.
41. 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.
42. B. Krishanamachari, D. Estrin, and S. Wicker. The impact of data aggregation in wireless sensor networks. In *International Workshop of Distributed Event Based Systems (DEBS)*, Vienna, Austria, Dec. 2002.
43. P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002.
44. P. Levis, N. Lee, A. Woo, S. Madden, and D. Culler. Tossim: Simulating large wireless sensor networks of tinyos motes. Technical Report UCB/CSD-TBD, U.C. Berkeley Computer Science Division, March 2003.
45. D. Liu and P. Ning. Distribution of key chain commitments for broadcast authentication in distributed sensor networks. In *10th Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, Feb 2003.
46. J. Liu, P. Cheung, L. Guibas, and F. Zhao. A dual-space approach to tracking and sensor management in wireless sensor networks. In *Proceedings of First ACM International Workshop on Wireless Sensor Networks and Applications*, September 2002.
47. C. Lu, B. M. Blum, T. F. Abdelzaher, J. A. Stankovic, and T. He. RAP: A real-time communication architecture for large-scale wireless sensor networks. In *Proceedings of IEEE RTAS 2002*, San Jose, CA, September 2002.
48. LynuxWorks. LynxOS 4.0 Real-Time Operating System. `http://www.lynuxworks.com/`.
49. S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *OSDI*, 2002.

50. S. Madden, W. Hong, J. Hellerstein, and M. Franklin. TinyDB web page. http://telegraph.cs.berkeley.edu/tinydb.
51. A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, USA, Sept. 2002.
52. Microsoft Corporation. Microsoft Windows CE. `http://www.microsoft.com/windowsce/embedded/`.
53. Microware. Microware Ariel Technical Overview. `http://www.microware.com/ProductsServices/Technologies/ariel_technology_bri%ef.html`.
54. Microware. Microware OS-9. `http://www.microware.com/ProductsServices/Technologies/os-91.html`.
55. Millenial Net. `http://www.millennial.net/`.
56. R. Min, M. Bhardwaj, S.-H. Cho, N. Ickes, E. Shih, A. Sinha, A. Wang, and A. Chandrakasan. Energy-centric enabling technologies for wireless sensor networks. 9(4), August 2002.
57. D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation 1996*, October 1996.
58. Nilsen Elektronikk AS. Nilsen Elektronikk Finite State Machine Operating System. `http://www.ethernut.de/en/software.html`.
59. R. Nowak and U. Mitra. Boundary estimation in sensor networks: Theory and methods. In *Proceedings of 2nd International Workshop on Information Processing in Sensor Networks*, Palo Alto, CA, April 2003.
60. Palm, Inc. PalmOS Software 3.5 Overview. `http://www.palm.com/devzone/docs/palmos35.html`.
61. A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J. D. Tygar. Spins: Security protocols for sensor networks. *Wireless Networks*, 8(5):521–534, sep 2002. Previous version of this paper appeared as PSWCT2001.
62. S. Ping. Something about time syncronization. XXX Lets get this written up as an Intel tech report.
63. J. Polastre. Design and implementation of wireless sensor networks for habitat monitoring. Master's thesis, University of California at Berkeley, 2003.
64. N. B. Priyantha, A. Miu, H. Balakrishnan, and S. Teller. The Cricket Compass for context-aware mobile applications. In *Proceedings of the 7th ACM MOBICOM*, Rome, Italy, July 2001.
65. QNX Software Systems Ltd. QNX Neutrino Realtime OS . `http://www.qnx.com/products/os/neutrino.html`.
66. Red Hat, Inc. eCos v2.0 Embedded Operating System. `http://sources.redhat.com/ecos`.
67. A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proc. of the 4th Operating Systems Design and Implementation (OSDI)*, pages 347–360, 2000.
68. C. Sharp. Something about the mag tracking demo. XXX Lets get this written up as an Intel tech report.
69. C. Sharp et al. NEST Challenge Architecture. `http://www.ai.mit.edu/people/sombrero/nestwiki/index/`.
70. J. A. Stankovic, H. Wang, M. Humphrey, R. Zhu, R. Poornalingam, and C. Lu. VEST: Virginia Embedded Systems Toolkit. In *IEEE/IEE Real-Time Embedded Systems Workshop*, London, December 2001.

71. Symbian. Symbian OS - the mobile operating system. `http://www.symbian.com/`.
72. uClinux Development Team. uClinux, The Linux/Microcontroller Project. `http://www.uclinux.org/`.
73. University of California at Berkeley. 800-node self-organized wireless sensor network. `http://today.cs.berkeley.edu/800demo/`, Aug. 2001.
74. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrating communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.
75. B. Warneke, M. Last, B. Leibowitz, and K. S. J. Pister. Smart dust: Communicating with a cubic-millimeter computer. *IEEE Computer*, 32(1):43–51, January 2001.
76. K. Whitehouse. The design of calamari: an ad-hoc localization system for sensor networks. Master's thesis, University of California at Berkeley, 2002.
77. K. Whitehouse and D. Culler. Calibration as parameter estimation in sensor networks. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, USA, Sept. 2002.
78. Wind River Systems, Inc. OSEKWorks 4.0. `http://www.windriver.com/products/osekworks/osekworks.pdf`.
79. Wind River Systems, Inc. pSOSystem Datasheet. `http://www.windriver.com/products/html/psosystem_ds.html`.
80. A. Woo and D. Culler. Evaluation of Efficient Link Reliability Estimators for Low-Power Wireless Networks. Technical report, UC Berkeley, 2002.
81. A. D. Wood and J. A. Stankovic. Denial of service in sensor networks. *IEEE Computer*, 35(10):54–62, Oct. 2002.
82. M. D. Yarvis, W. S. Conner, L. Krishnamurthy, A. Mainwaring, J. Chhabra, and B. Elliott. Real-World Experiences with an Interactive Ad Hoc Sensor Network. In *International Conference on Parallel Processing Workshops*, 2002.
83. W. Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks. In *Proceedings of IEEE Infocom 2002*, New York, NY, USA., June 2002.
84. W. Ye, J. Heidemann, and D. Estrin. A flexible and reliable radio communication stack on motes. Technical Report ISI-TR-565, USC/ISI, Aug. 2002.
85. K. M. Zuberi, P. Pillai, and K. G. Shin. EMERALDS: a small-memory real-time microkernel. In *Symposium on Operating Systems Principles*, pages 277–299, 1999.