The Emergence of Networking Abstractions and Techniques in TinyOS

Philip Levis†, Sam Madden*‡, David Gay‡, Joseph Polastre†, Robert Szewczyk†, Alec Woo†, Eric Brewer†, and David Culler†

†EECS Department	‡Intel Research Berkeley	*CSAIL
University of California, Berkeley	2150 Shattuck Ave.	MIT
Berkeley, CA 94720	Berkele y, CA 94704	Cambridge, MA 02139

Abstract

The constraints of sensor networks, an emerging area of network research, require new approaches in system design. We study the evolution of abstractions and techniques in TinyOS, a popular sensor network operating system. Examining CVS repositories of several research institutions that use TinyOS, we trace three areas of development: single-hop networking, multi-hop networking, and network services. We note common techniques and draw conclusions on the emerging abstractions as well as the novel constraints that have shaped them.

1. Introduction

Networked systems of small, often battery-powered embedded computers, referred to as EmNets in a recent NRC report [17], are touted as a revolution in Information Technology with "the potential to change radically the way people interact with their environment by linking together a range of devices and sensors that will allow information to be collected, shared, and processed in unprecedented ways." They are also seen as requiring a dramatically new approach to network system design:

`EmNets are more than simply the next step in the evolution of the personal

> computer or the Internet. Building on developments in both areas, EmNets will also be operating under a set of constraints that will demand more than merely incremental improvements to more traditional networking and information technology." [17]

Networking issues are at the core of the design of embedded sensor networks because radio communication - listening, receiving, and transmitting - dominates the energy budget, defining the lifetime of a deployed system.

Much of the research in this area has been based on the TinyOS operating system, created at UC Berkeley [23], but now a public Sourceforge project including many other groups. In the three years since TinyOS's introduction, it has been used by many research groups to address various aspects of EmNet design. Most of this work is available in the open, making it possible to examine the new system structures that have emerged and the common techniques that they exploit. Are they, in fact, substantially different from general purpose systems?

To begin to answer this question, this paper draws on our own experience building TinyOS, TinyDB and other applications as well as a study of development represented by the CVS trees of several other research groups (CMU, UCLA, USC, UIUC, UVA, and Vanderbilt). An appendix contains the list of the sources used to collect this data. We look in particular at code evolution through several generations of hardware platforms, OS releases, and applications. Our primary focus is on networking abstractions as they appear in applications with particular sensing and system management structures. We find that abstractions fall into four categories:

- Certain abstractions appear to be *general*: they are widely used, and TinyOS provides both the mechanism and policy to support them. The use of the active messages networking abstraction [6], multihop broadcast, and tree-based routing are prime examples.

- Others appear to be *specialized*: they are also widely used, but TinyOS provides only mechanisms. Each application includes code to dictate a specific policy. A typical example is power management, where the application decides when to power down particular subsystems.

- Still others are *in flux*, without any consensus on the abstraction. Instead, these abstractions are often implemented as part of an application. For example, epidemic protocols are often used, but the TinyOS community has not developed a common interface or implementation.

- Finally, a set of abstractions are conspicuous by their *absence*, in that they appear widely in the literature, but are scarcely used in the applications we find in our sample set.

Abstractions have moved among these classes, as developers refined them; our

classification is merely an observation of the current state of development, to which we look for insight. For example, initially tree-based routing was built into each application. It has since emerged as a general abstraction, with several implementations of a common interface used by multiple applications.

We also identify four system design techniques used in these abstractions that are comparatively uncommon or unimportant in more general systems. First, **cross-layer control**, where two system levels interact in a mutual relationship, is quite common. This reflects the hardware constrained and application specific nature of EmNets, but also the use of a 'design to suit' framework, rather than a strict set of predefined interfaces. Second, there are many instances of rate-matched data pumps, either of network or sensor data; the comparatively small storage on these devices often requires a **static resource allocation** discipline, where queue sizes are carefully considered and allocated at compile-time. Finally, **snooping** and **scheduled communication** pose a design tension. Snooping depends on frequently listening to the radio to collect as much data as possible, while scheduled communication avoids the energy cost of listening by coordinating the schedules of senders and receivers.

Section 2 provides a brief background on TinyOS, major hardware platforms, and three example applications. In Sections 3, 4, and 5, respectively, we examine single-hop (data link) communication, multi-hop (network) communication, and system services of particular importance to networking (time synchronization and power management). For each, we examine systems that have been built by the larger TinyOS community to identify networking abstractions and how they evolved. In Section 6 we survey the identified abstractions and discuss common system design techniques and trends.

2. TinyOS

As background, we review the basic design of TinyOS and summarize the variety of hardware platforms and applications that have driven TinyOS development, focusing on the requirements dictated by each.

2.1 TinyOS Design

TinyOS focuses on three high-level goals in sensor network system architectures [23]:

- Take account of current and likely future designs for sensor networks and sensor network nodes.
- Allow diverse implementations of both operating system services and applications, in varying mixes of hardware (in different mote generations) and software.
- Address the specific and unusual challenges of sensor networks: limited resources,

concurrency-intensive operation, a need for robustness, and application-specific requirements.

To achieve these goals, TinyOS provides an efficient framework for modularity and a resource-constrained, event-driven concurrency model. The modularity framework allows the OS to adapt to hardware diversity while still allowing applications to reuse common software services and abstractions.

The need to handle high concurrency comes from the observation that sensor nodes predominantly process multiple information flows on the fly, as opposed to performing heavy computation. Nodes must simultaneously execute several operations at once, but have limited storage. Combined with the need for sensor nodes to be robust, an event-driven concurrency model best suits this class of system. Therefore, all of the abstractions and techniques we explore are cast in an event-driven model.

```
interface StdControl { // booting and power management
  command result_t init();
  command result_t start();
  command result_t stop();
}
interface ADC { // data collection
  command result_t getData();
  command result_t getContinuousData();
  event result_t dataReady(uint16_t data);
}
interface SendMsg { // single-hop networking
  command result_t send(uint16_t addr, uint8_t len, TOS_MsgPtr msg);
  event result_t sendDone(TOS_MsgPtr msg, result_t success);
}
interface ReceiveMsg { // single-hop networking
  event TOS_MsgPtr receive(TOS_MsgPtr m);
}
```

Figure 1: Common TinyOS Interfaces. *ADC and SendMsg are split-phase operations, hence the combination of commands and events.*

Program execution in TinyOS is rooted in *hardware events* and *tasks*. Hardware events are interrupts, caused by a timer, sensor, or communication device. Tasks are a form of deferred procedure call that allows a hardware event or task to postpone processing. Tasks are *posted* to a queue. As tasks are processed, interrupts can trigger hardware events that preempt tasks. When the task queue is empty, the system goes into a sleep state until the next interrupt. If this interrupt queues a task, TinyOS pulls it off the queue and runs it. If not, it returns to sleep. Tasks are atomic with respect to each other.

System modularity is based on a component model. A named component encapsulates some fixed-size state and a number of tasks. Components interact with each other strictly via function call *interfaces*, which are related groups of *commands* and *events*. The set of interfaces a component uses and provides define its external namespace. Commands typically represent requests to initiate some action; events represent the completion of a request or something triggered by the environment, e.g., message reception. Both explicitly return error conditions, such as the inability to service the

request. A specific set of events are bound to hardware interrupts. A programmer assembles an application by specifying a set of components and "wiring" their interfaces together.

The TinyOS concurrency model intentionally does not support blocking or spin loops. As a result, many operations are *split-phase*: the request is a command that completes immediately, and an event signals completion of the requested action. This approach is natural for reactive processing and for interfacing with hardware, but complicates sequencing high-level operations, as a logically blocking sequence must be written in a state-machine style. It allows many concurrent activities to be serviced on a single stack, reflecting the very limited RAM on modern microcontrollers. It also makes handling of error conditions explicit.

TinyOS intentionally defines neither a particular system/user boundary nor a specific set of system services. Instead, it provides a framework for defining such boundaries and allows applications to select services and their implementations. Nonetheless, a common set of services has emerged and is present on most platforms and used by most applications. This include timers, data acquisition, power management, and networking. Some interfaces supporting these services are shown in Figure 1. This general decomposition is not unique to TinyOS; it can be found in other sensor network platforms, such as SensorSim [36], MANTIS [2], and EmStar [15]. It covers the basic requirements of a sensor network: using little power, periodically collect data, perform some simple processing on it, and, if needed, pass it to a neighboring node.

2.2 Hardware Platforms

Table 1: Hardware Platform Evolution

Mote	WeC	rene	dot	mica	mica2	mica2 dot	iMote [26]	btNode[25]
Released	1999	2000	2001	2002	2003	2003	2003	2003
Processor	4 MHz				7 MHz	4 MHz	12 Mhz	7 Mhz
Flash (code, kB)	8	8	16	128	128	128	512	128
RAM (kB)	0.5	0.5	1	4	4	4	64	4
Radio (kBaud)	10	10	10	40	40	40	460	460
Radio Type	RFM				ChipCon	ChipCon	Zeevo BT	Ericson BT
µcontroller	Atmel						ARM	Atmel
Expandable	no	yes	no	yes	yes	yes	yes	yes

There has been a dramatic evolution in hardware platforms since TinyOS was first designed. Table 1 summarizes the generations of the processing board of Berkeley

"motes" and similar designs from other groups. Although the Berkeley mote's microcontrollers are drawn from the same family (Atmel AVR), the radios, their interfaces, and the chip-to-chip interconnects differ substantially. Other hardware platforms that support TinyOS by Intel [26] and the ETH [28] use different processors (ARM) and/or different radios. Several companies have developed other variants. Highly integrated experimental devices with a processor and radio integrated on a single tiny die also exist [22]. We examine the impact these hardware variations have on networking in greater detail in Section 3.

While it is reasonable to expect that the capability of a device with a given size and power limit will improve with time, but the rate of improvement is unlikely to mirror that of desktop systems and there will be similar pressures to reduce the size and power consumption for a given capability. The balance of processing, memory, and communication for the designs in Table 1 is very far from the 1 MIPS:1 MB:1 mbps rule of thumb. Small, simple processors are fast, but memory accounts for most of the microcontroller chip and most of the standby power consumption. The radio consumes the majority of the active power.

2.3 Applications

Sensor network development has been very application driven. TinyOS applications have evolved from simple sense-and-route demos to a variety of complete applications, some of which have had long term deployments. We present three examples that illustrate a range of sensor network abstractions:

Habitat Monitoring (TinyDB): patches of nodes gather sensor data for several months from areas of interest to natural scientists or other environmental observers. Example deployments include James Reserve [32], Great Duck Island [31] and a vineyard in British Columbia [44]. We use TinyDB [30] as a representative habitat monitoring application. It allows a user to collect data from a sensor network using a flexible database-like query language. Nodes cooperatively process queries to collect and extract the requested data. TinyDB represents a class of stationary networks that monitor the encompassing space.

Shooter Localization: a small subset of a sensor network localizes the origin of a bullet in an urban setting [27]. Individual nodes detect projectile shockwaves (with latency in the tens of microseconds) and the sound of the weapon firing, and use the time between the two to estimate distance. Actual shooter location is computed centrally. Shooter localization represents a class of stationary networks that monitor objects or phenomena within a space.

Pursuer-Evader: a large network tracks the movement of one or more evader robots using magnetic or other field readings. The network routes this information to a pursuing robot using geographic or landmark routing [9]. Pursuer-evader represents a closed-loop network with stationary and mobile nodes, and was developed by several groups as part of a demo for the EmNet-related DARPA program, NEST [40].

These three applications drive three different areas of TinyOS development: Habitat Monitoring needs to keep energy consumption low, so that multi-month deployments are possible; Shooter Localization requires a high sample rate and fine-grain time synchronization; Pursuer-Evader requires mote localization and more advanced routing (to the mobile pursuer).

3. Single Hop Communication

Active messages [6] has remained the basic networking primitive of TinyOS. Active messages is a simple message-based networking abstraction where messages include an identifier that specifies an action to be executed upon message reception. Although the abstraction has changed little, its implementation and features have changed substantially, due to changing hardware platforms and emerging needs. There have been three major networking stacks from UC Berkeley, one for each of the primary platforms: rene, mica, and mica2. The primary differences between these stacks can be traced to shifts in the network hardware/software boundary.

The core challenge has been to meet the hard real time requirements of networking hardware. The radio timing requirements have influenced and restricted the use of tasks not only in the radio stack, but throughout TinyOS. Reliable radio reception requires high-frequency low-jitter channel sampling. This is simplified by raising the hardware/software boundary. However, raising this interface is not without cost: useful features, such as fine-grained power management, time-stamping, selective back-off, and link-level packet acknowledgment, become more expensive and complex to provide on more sophisticated radio interfaces. Leopold et al. [25] observe similar issues with the Bluetooth interface: "the arguments mentioned above [high energy utilization, lack of timestamps, no access to connectivity data] disqualify Bluetooth as a first choice for sensor nodes."

3.1 Communication Interfaces

In most cases, a macro-component called GenericComm encapsulates the TinyOS network stack. GenericComm provides the active message communication interfaces (SendMsg and ReceiveMsg, as shown in Figure 1), which support single hop unicast and broadcast communication. Message buffer structures have a fixed size. CSMA

(carrier-sense, multiple access) is the default media access. Applications and the network stack exchange message buffers through pointers: senders provide their own storage for messages (there is no send buffer pool). Successful calls to send implicitly yield the provided buffer to GenericComm, which returns it to the sender when signaling the sendDone event.

GenericComm provides limited receive buffering. When a component receives a message, it must return a buffer to the radio stack. The stack uses this buffer to hold the next message as it arrives. Typically, a component consumes and returns the buffer it was passed; however, if it wishes to store this buffer for later use, it may instead return a pointer to another free buffer. If it has exhausted all its buffering, the component must determine what should be dropped. The stated goal is to keep the system responsive and ensure that there is always a free buffer for the radio stack to use. Components that need to accumulate several packets before processing them must allocate buffers statically and locally [11], rather than relying on the network stack to provide arbitrary buffering.

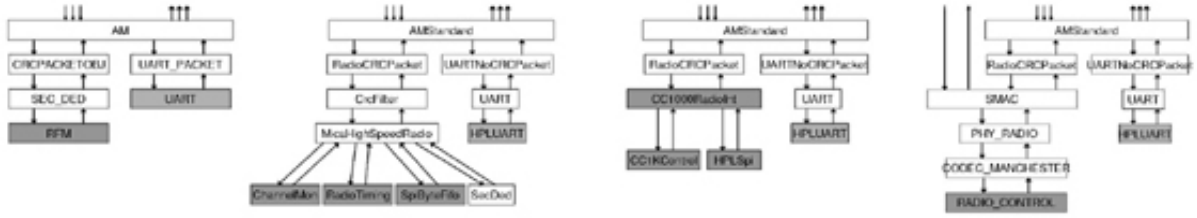


Figure 2: Typical single hop network stacks for three generations of motes. From left to right: rene, `mica`, `mica2`, and S-MAC on `mica` radio stacks. Grey components abstract hardware.

Alternative interfaces for single hop communication exist. For instance, S-MAC [45] is an RTS/CTS-based networking stack that supports fragmentation at the data link level. This allows S-MAC to support sending messages larger than the link MTU (maximum transmission unit). It schedules RTS/CTS exchanges and turns radios off to reduce listening costs transparently. S-MAC preserves the split-phase nature and zero-copy semantics similar to SendMsg in its byte stream interface and provides an Active Message interface.

Based on the code we reviewed in CVS, almost all applications developed under TinyOS use active messages. It appears that the availability of large message sizes is not yet a general need: in the few cases where the default message size has been insufficient, applications have increased it modestly (e.g., from 36 to 56 bytes in TinyDB). TinyDiffusion, a naming protocol discussed in Section 4.2, uses S-MAC, but does not send messages larger than the link MTU.

We now examine in detail each of the three implementations of active messages, beginning with the earliest implementation, for the rene mote. We emphasize the differences between successive implementations, noting ways in which the hardware/software boundary changed and which of these ways most influenced the capabilities of the system.

3.2 AM Stack Implementations

Figure 2 shows the structure of the various stack implementations. The rene RFM TR1000 radio is extremely simple. The system modulates the channel by writing a DC-balanced bit sequence to the transmit pin at precise times. Similarly, the data is recovered by reading the receive pin at precise times to sample the channel. The RFM component exports this bit-level interface. Bits are received and transmitted in a timer interrupt handler. CSMA media access control is intertwined with data encoding and transmission within the SEC_DED component. CRCPACKETOBJ performs the packet-to-byte translation, with CRC checking, while AM is responsible for application layer dispatch.

The mica hardware uses the same radio as the rene and can use the rene stack unchanged. However, the mica adds hardware edge detection and byte-to-bit conversion on top of the RFM chip. This allows two major advancements: more precise bit timing, and increased data rates (40KBaud over rene's 10KBaud). Instead of a strictly layered approach, the mica stack delegates separate parts of the communication process to different components. It separates out the media access control (ChannelMon), bit-level timing (RadioTiming) and data transfer (SpiByteFifo). MicaHighSpeedRadio signals packet reception events to the active message layer as well as low-jitter timing events at specific points in packet reception or transmission (e.g., when the channel is acquired). Interpositioning is used to provide CRC checks on the packet level (through CRCFilter).

A later revision of the stack introduced synchronous link-layer acknowledgments. The sender and receiver swap roles just as the last data byte of the packet is sent. The receiver transmits an acknowledgment code (a predefined sequence of bits), and the sender reports if it heard the code in its sendDone event. The sender then has a synchronous acknowledgment at the cost of a few byte times and can immediately decide whether to retransmit the buffer or use it for a new message.

Low power listening reduces the cost of idle listening by lowering the radio sample rate when looking for a packet start symbol and turning off the radio between samples. This requires a transmitter to send a longer start symbol. On the RFM TR1000, which can turn on in less than a bit time, low power listening can lead to significant idle listening energy savings at only a small overhead to a transmitter. Once a receiver detects a packet, it keeps its radio on and receives at the full data rate.

The mica2 is based on a Chipcon CC1000 radio, which performs bit synchronization and encoding and exports only a byte-level interface. A single, large component, CC1000RadioInt, replaces the modular architecture used in mica. Initially, the mica2 stack did not provide link-layer acknowledgments, but later revisions added them as well as low power listening. The application-level semantics of CC1000 acknowledgments are the same as the mica. However, the hardware interface requires

the implementation to send a tiny ACK data packet instead of an ACK code, and low power listening is not nearly as efficient.

New platforms are emerging that support the IEEE 802.15.4 standard for personal area wireless networks. 802.15.4 radios provide packet level interfaces for sending and receiving, link layer encryption and authentication, link-layer acknowledgments, CRC checking, and early packet rejection. These new standardized radios move most of the functionality of the CC1000 radio stack into hardware. simplifying the TinyOS code.

The S-MAC stack in Figure 2 shows an alternative approach to layering. The complex SMAC component handles backoff, retransmission, RTS/CTS/ACK handshakes, fragmentation, and radio duty cycle control. PHY_RADIO performs CRC checking and byte buffering while CODEC_MANCHESTER runs Manchester coding. RADIO_CONTROL handles the low-level carrier sensing and start symbol detection.

3.3 Analysis

Hardware differences between mote platforms affected software structure and networking capabilities in unforeseen and surprising ways. We discuss four examples: hard real time issues, low power listening, link layer acknowledgments, and low-level hooks.

The concurrency model was intended to allow a collection of components to be replaced by hardware and vice versa. Moving the hardware/software boundary changes the division of work between tasks and hardware events, leading to different maximum critical section and task lengths. The bit-level interface in rene requires the stack to handle interrupts at a high enough rate that the handler cannot encode or decode bytes; encoding/decoding is deferred to tasks. Because it has a small buffer, the encoding layer (SEC_DED) must run a task every byte time. This means no other tasks can run longer than a byte time (1.8ms). A larger encoding/decoding buffer would reduce sensitivity to long-running application tasks.

In mica, the addition of a byte-level hardware abstraction reduced the interrupt rate. This increased the number of cycles available for each event, allowing encoding/decoding to remain in the interrupt handler. Tasks are posted by the packet level component, making the stack more resilient, but not immune, to the effects of long-running application tasks: the maximum task length increases to a packet time, approximately 25 milliseconds.

Both the rene and `mica2`stacks support low-power listening. However, it is much more efficient in the former, because the TR1000 can turn on very quickly, while the CC1000 turns on slowly. This latency determines how quickly the stack can sample the channel, and therefore how much it can reduce the cost of idle listening.

Synchronous link layer acknowledgments present another timing issue. In the mica stack, they depend on being able to quickly transition (in 12 μs, where a raw bit time is

25 µs) the RFM radio between send and receive mode while maintaining synchronization to within a bit time. The Chipcon radio transitions much more slowly (250 µs, where a raw bit time is 26 µs) and self-synchronizes at byte boundaries, precluding this technique. For this reason, mica2 acknowledgments require a compete packet transmission. This is a well-known and fundamental tension: moving the hardware boundary further up the stack improves performance, but also disallows certain capabilities enabled by low-level hardware access. The important parameter in this case was transition time, as opposed to more traditional measures of radio performance such as throughput or sensitivity.

An additional feature that emerges is low-level hooks that, following the technique of cross-layer control, allow higher level components to monitor radio events with high precision. We examine one common use of these hooks in Section **5.2**, when we discuss time synchronization. They are also commonly used in localization systems based on time of flight measurements [**41**].

In summary, we observed the emergence of the AM abstraction as general and widely used throughout TinyOS. Lower levels of the radio stack, however, are implementation-specific and tied to a particular hardware platform. The development of the various mote platforms shows that several hardware characteristics enable important features and influence the ability of the radio stack to coexist with applications and services.

4. Multi-Hop Communication

The literature describes many ad-hoc multi-hop routing algorithms, where network routes are discovered through a self-organizing process [**9**,**13**,**37**,**38**,**43**]. Similarly, varieties of multi-hop routing components are among the most diverse and numerous implementations contributed to the TinyOS repository. We broadly divide these components into three classes: *tree-based collection*, where nodes route or aggregate data to an endpoint, *intra-network routing* where data is transfered between in-network end-points, and *dissemination*, where data is propagated to entire regions. Habitat monitoring primarily uses tree-based collection, whereas Pursuer-evader routes between mobile in-network regions. Essentially all applications use some form of broadcast or dissemination to convey commands, reconfigure, or control in-network processing. Examining several multi-hop implementations, we see the emergence of two common abstractions: (i) a neighborhood discovery and link quality estimation service and (ii) an augmented version of the AM interface that supports packet encapsulation and monitored forwarding. We summarize these classes and common abstractions, noting several key points in the evolution of routing as implemented in TinyOS.

4.1 Tree-Based Routing

Tree-based routing is primary based on two pieces of information: a parent node identifier, and a *hop-count* or depth from the tree root, i.e., the parent's hop-count plus

one. A routing tree is built via local broadcast from the root followed by selective retransmission from its descendents. A node routes a packet by transmitting it with the parent as the designated recipient. The parent does the same to its parent, until the packet reaches the root of the tree. The key design issues are how the routing tree is discovered and maintained, as well as how the forwarding is performed. We examine the historical improvements of tree formation over five successive protocols (AMROUTE, BLess, Surge, mh6, and MultiHopRouter).

The early AMROUTE builds a tree using periodic beacon floods from the root, keeping no route history; it only maintains a single, current parent, the first receipt of the recent beacon. BLess, in contrast, uses no flooding. Instead, each node listens to routed data packets and greedily selects a parent from overheard transmissions. The root node seeds the tree by periodically broadcasting BLess packets, but these broadcasts are not retransmitted. BLess maintains a table of candidate parents and uses the parent with the lowest hop-count.

Surge also uses beaconless tree formation and maintains a table of candidate parents, but selects its parent based on a combination of link quality (packet success rate) and hop count. mh6 [43] and its re-implementation in MultiHopRouter extend this approach by filtering the neighbor set (based on link quality and other factors) and choosing the parent based on an estimated cost to the root through each neighbor. Each node computes quality estimates on incoming links and periodically transmits these, along with its cost estimate to the root, allowing neighbors to determine outgoing link quality and total path estimates. Parent selection algorithms try to minimize either end-to-end packet loss or, with link-level acknowledgments, total expected transmissions, including retransmissions. Surge and MultiHopRouter provide support for retransmission and output queuing. Similar techniques for asymmetric link rejection and neighborhood management have recently been proposed for ad-hoc routing in 802.11 networks [13].

One reason why many implementations of tree-based protocols exist is that it is straightforward to construct a basic tree-based topology and forwarding mechanism.application. However, substantial care is required to construct and maintain a stable topology that can provide good connectivity in a diverse radio environment for weeks or months. The introduction of parameterized interfaces in NesC, combined with the development of more mature routing protocols made it possible to build a robust, encapsulated routing layer that could be easily reused in many applications.

4.2 Intra-network Routing

Several established ad hoc protocols, such as DSDV, AODV, and Directed Diffusion, have been implemented for TinyOS in various reduced forms. DSDV and AODV are designed for unicast routing to specific endpoints. In the TinyOS version implemented by Intel Oregon [44], the basic algorithms for route discovery and maintenance are similar to their IP counterparts, but the final results are not. Instead, they maintain a

single route, much as tree-based protocols do. TinyDiffusion builds, then prunes, a routing tree based at a particular requesting source. [19]

A number of protocols structure network names to assist routing. A fully featured implementation of GPSR [24], which uses a node's geographic location as its name, has been completed by University of Southern California. It supports both greedy geographic routing and perimeter routing to recover from local failures. Pseudo-geographic routing, in which each node is assigned a vector of hop-count distances from several beacons instead of coordinates in a Cartesian plane, has been explored as well. [38] Intra-network routing, the mainstay of Internet usage, is uncommon in TinyOS applications. One exception is Pursuer-Evader, which provides mobile-to-mobile routing within a single routing tree rooted at a landmark. The destination reinforces a path from the landmark for downward routing.

4.3 Broadcast and Epidemic Protocols

Many applications need to reliably disseminate a data item to every node in a network. For example, in TinyDB new queries must be installed, while in Pursuer-Evader the application can be reconfigured in-situ (e.g., to change filter settings or radio transmit power). Reliable network-wide dissemination can also be used to distribute new versions of TinyOS programs.

In practice, reliable data dissemination has two principal implementations. The first, a simple flooding protocol, is common, easily implemented, and often tightly integrated in an application; it appears in Pursuer-Evader as well as others. In this implementation, the source (usually a base station) generates a data packet, and each node that hears it forwards it once. Experience has shown that a single flood often reaches most nodes very quickly, but collisions and lossy links lead to several motes not hearing the data. Typically, the source repeats the flood several times, until every node receives it. Determining when to stop requires either visual inspection or routing data out of the network to the base station. Pursuer-Evader uses a delayed retransmission strategy, which greatly reduces collisions and improves coverage.

The second approach is to use an *epidemic* algorithm. Instead of a single flooding event, nodes periodically exchange information to know when to propagate data [14]. For example, the Maté virtual machine uses a purely epidemic algorithm to disseminate new code [29]. In contrast to a flood, an epidemic only transmits when needed. Local suppression mechanisms can reduce redundant transmissions, saving energy. By ensuring reliable delivery to every connected node, an epidemic approach is robust to transient disconnections and can propagate to new nodes added to a network.

TinyDB uses a hybrid approach to install and stop queries. It first floods new queries into the network; this reaches most nodes very quickly. The network then uses an epidemic approach to reach the few remaining nodes that missed the flood. As every data message contains a query ID, nodes can detect inconsistencies by snooping on

local data traffic.

4.4 Common Multi-Hop Developments

```
interface Send {
  command result_t send(TOS_MsgPtr msg, uint16_t length);
  command void* getBuffer(TOS_MsgPtr msg, uint16_t* length);
  event result_t sendDone(TOS_MsgPtr msg, result_t success);
}
interface Intercept {
  event result_t intercept(TOS_MsgPtr msg, void* payload, uint16_t payloadLen);
}
```

Figure 3: The Send and Intercept Interfaces for Routing.

We observe a number of common developments that have occurred in several multi-hop networking implementations for TinyOS. First, most of the current multi-hop routing implementations - MultiHopRouter, TinyDiffusion, GPSR, BVR - discover and manage a list of neighboring nodes for possible routes. They use this information when initially constructing routes and to adapt to connectivity changes, including node appearance and disappearance. Typical information appearing in neighborhood tables includes node addresses, link quality estimates, and routing metadata, such as hop-count in routing-tree protocols. Note that this table contains both link state (link quality) and routing layer (hop-count) information. With limited memory, the table is constrained to a limited number of entries. Routing components utilize link information in route selection, while link components utilize routing information in table management. Aspects of both are conveyed in route update messages.

Recent experiments have established that sensor networks experience lossy and asymmetric links whose quality changes over time [7,42]. Traditional protocols that assume bi-directional connectivity are likely to fail in such an environment; many early multi-hop routing protocols in TinyOS correspondingly suffer from poor end-to-end packet delivery [20]. To remedy this, recent routing layers in TinyOS, such as MultiHopRouter, BVR, TinyDiffusion and the TinyOS DSDV implementation include the notion of a *link quality estimator* that identifies a set of bi-directional, high-quality links that network, transport, and application layers can rely upon.

A second common development in multi-hop routing is that routing layer implementations (e.g., DSDV, MultiHopRouter) have begun to use the Send and Intercept interfaces, shown in Figure 3. The getBuffer command in the Send interface allows the routing layer to control the offset of the application payload in a message buffer, which is useful for packet encapsulation. A routing layer signals the Intercept event when it receives a packet to forward. An application can suppress forwarding by returning a certain result code. This allows application such as TinyDB to locally aggregate data. These interfaces are examples of cross-layer control, and enhance the AM abstraction with support for multi-hop communication by interposing new interfaces. Similar interfaces would support broadcast with processing at each hop.

A third development is augmenting low-level network abstractions to include interfaces

for promiscuous communication, where the network stack can pass non-locally addressed packets up to a higher level component. Some link estimation and neighbor table management modules rely on this to learn about nearby nodes. TinyDB uses it for application-specific network optimizations. This prevalent use of snooping indicates that it is a technique of general utility in sensor networks.

Finally, a fourth development that has recently emerged in multi-hop protocols is the addition of a send queue. Initial implementations, such as BLess, have a single packet buffer, and drop outgoing packets when the underlying communication components are busy transmitting. Surge, mh6, and MultiHopRouter all add outgoing queues, but have different queuing policies. For example, mh6 maintains separate forwarding and originating queues, giving priority to originating, while MultiHopRouter gives priority to forwarding. No implementations that we found include a receive queue beyond the simple buffering provided by AM.

4.5 Observations

Except for GPSR, all multi-hop protocol implementations in TinyOS we could find are built on top of the AM abstraction. The stability and wide use of the AM interface suggest that protocol developers are satisfied with the simple dispatch function it provides. It is possible that this stability reflects a desire not to modify the lower level portions of the network stack, which tend to be complex. However, the untyped packet interface below AM is similarly isolated from this complexity. It is worth noting that TinyDiffusion, which is built on top of S-MAC, currently uses S-MAC's AM interface rather than an interface which S-MAC provides that includes features like fragmentation.

When a message exceeds the length of an AM packet, application level framing is used [10]. For example, the TinyDB application partitions both queries and query results into logical units that have meaning on their own, (e.g., a field in a query result.) If a single fragment gets lost, the application can still use the other fragments.

Applications developed on TinyOS have predominantly used tree-based routing. Some applications, such as pursuer-evader, use intra-network routing, but implementations tend to be single destination and fairly simple. The literature, however, has proposed many intra-network routing algorithms. There are several potential explanations for this difference in usage. Complex routing algorithms may have difficulty scaling down to resource constrained nodes. The implementations may simply be too immature enough to have seen use in released applications. This was the case with initial implementations of tree-based routing which were not sufficiently robust to be widely adopted (despite their simplicity). However, it appears that general point-to-point routing is simply less common in applications of embedded networks, which tend to operate in aggregate on information that is distributed throughout the network. In such applications, information tends either to be broadcast out or to flow in a single direction towards a small number of sinks.

Applications are increasingly using reliable dissemination for programming or reconfiguration. For small networks, simple floods are sufficient. As sensor networks are deployed at larger scales, the need to respond to unforeseen system interactions increases, as does the benefit of an epidemic-based solution. However, unlike prior IP-based epidemics, sensor nets can take advantage of geographic proximity and spatial redundancy.

Based on the observations in this section, we note several progressions in the development of multi-hop networking in TinyOS. First, the evolution of a neighborhood management table with the ability to reject asymmetric links and select low-loss routes has finally resulted in a widely used tree-based routing component - MultiHopRouter. Second, the use of snooping (as opposed to broadcast floods) to gather neighborhood information and construct initial routes has become standard practice. Finally, the appearance of send queues suggests that applications may need to be tolerant of significant transmission delays and that some traditional networking techniques (e.g., differentiated services or load shedding) are applicable in this domain.

5 Network Services

A number of abstractions support efficient, low-power networking in TinyOS. In this section, we focus on two prominent examples, power management and time synchronization.

5.1 Power Management

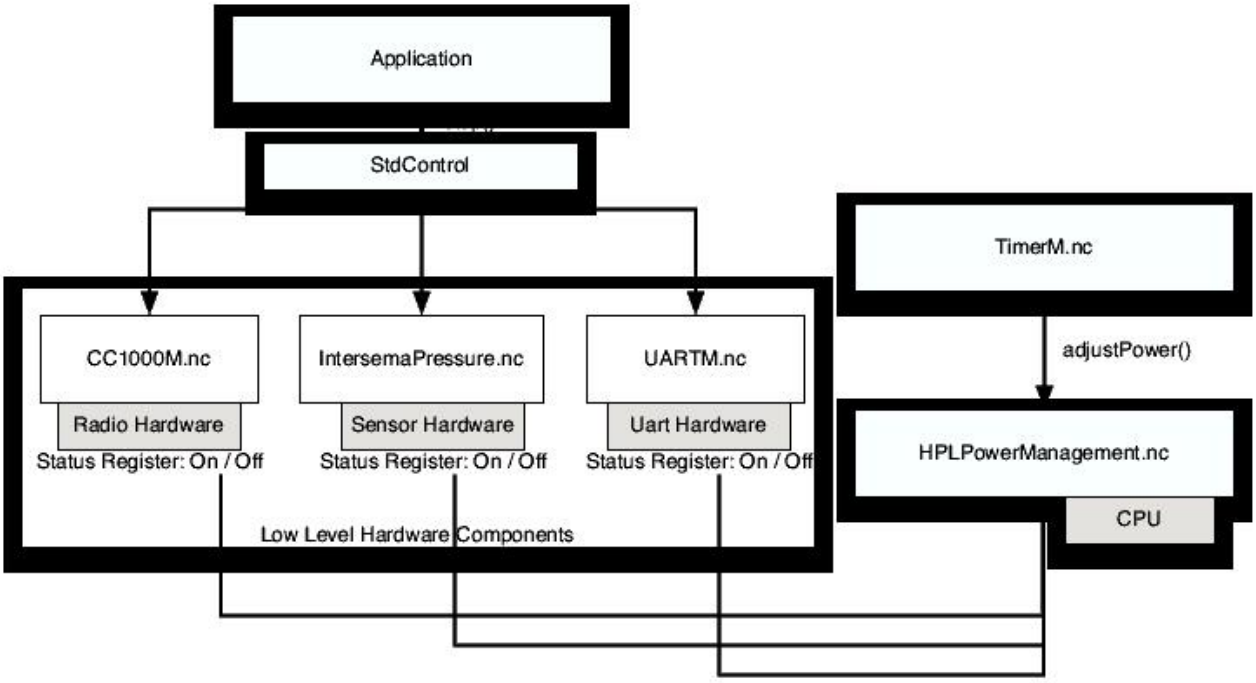
TinyOS manages power management through the interaction of three elements. First, each service can be stopped through a call to its StdControl.stop command (see Figure 1); components in charge of hardware peripherals can then switch them to a low-power state. Second, the HPLPowerManagement component puts the processor into the lowest-power mode compatible with the current hardware state, which it discovers by examining the the processor's I/O pins and control registers. Third, the TinyOS timer service can function with the processor mostly in the extremely-low-power *power-save* mode.

TinyDB uses these features to support sensor network deployments that last for months. In this context, idle listening dominates energy consumption. As discussed in Section 3.2, low-power listening reduces the cost of idle listening by increasing the cost of transmission. However, instead of low power listening, TinyDB uses communication scheduling. Using coarse-grained (millisecond) time synchronization, TinyDB motes coordinate to all turn on at the same time, sample data, forward it to the query root, and return to sleep.

Figure 4 illustrates TinyDB's power management approach. At the end of a round of

data collection, each mote calls StdControl.stop to stop both the on-board sensor hardware (IntersemaPressure) and the radio (CC1000M, UARTM). After the next timer event, HPLPowerManagement puts the processor into power-save mode (via adjustPower). At the start of the next data collection round, the timer wakes the mote up, and StdControl.start is called to restart the sensors and radio. This approach to communication scheduling requires time synchronization when used in conjunction with multi-hop routing, discussed below.

Figure 4: The TinyDB power management API. The application calls StdControl.stop to halt the low-level hardware. HPLPowerManagement.nc sees changes to the hardware status registers, which causes it to put the CPU into a low-power sleep state.



Power management illustrates cross-layer control at a very low-level: HPLPowerManagement goes directly to the hardware to determine when the processor can be switched into various low-power modes (e.g., idle, power-down, power-save, standby, extended standby). Correspondingly, power management is an abstraction that must inherently be specialized: effective power management without application input is not possible. For example, by supplying a small bit of application information, TinyDB allows TinyOS to spend most of its time in a very low power mode. Approaches such as S-MAC take a static approach to this communication scheduling, always waking at a certain fixed interval. TinyDB, however, allows this scheduling to dynamically change in a fine grained manner with regards to application needs (e.g. query sampling rates), conserving more energy.

5.2 Time Synchronization

Another service that many network-centric applications need is time-synchronization.

Such a service is useful in several scenarios. For example, sensor fusion applications that combine a set of coincident readings from different locations, such as shooter localization, need to establish the temporal consistency of data. TDMA-style media access protocols need fine-grained time synchronization for slot coordination, and power-aware approaches to communication scheduling (discussed above) require senders and receivers to agree when their radios will be on.

Several groups, including UCLA[1], Vanderbilt[2], and UC Berkeley[3], have implemented time synchronization. GenericComm provides a hook for modifying messages just as it transmits the first data bit, after media access. All of these implementations work similarly: they use the hook to place a time stamp in the packet. This allows very precise time-synchronization that might otherwise not be possible [16]. This is another example of cross-layer control in TinyOS. The Vanderbilt implementation models possible delays, which allows it to obtain slightly better accuracy than the other approaches; all three report sub-millisecond accuracy.

Initial efforts to develop a general purpose, low-level time synchronization component were unsuccessful. A number of subtle and bad interactions observed with some higher level applications, such that timer events are missed or software components hang in inconsistent states. Applications were fragile to time-critical intervals suddenly becoming slightly shorter or longer.

Instead, the current approach taken by TinyOS is to provide the mechanism to get and set the current system time (and time stamp messages at a low level), but to depend on applications to choose *when* to invoke synchronization. For example, in the TinyDB application, when a node hears a time-stamped message from a parent in the routing tree, it adjusts its clock so that it will start the next communication period at exactly the same time as its parent. It does this by changing the duration of the sleep period between communication intervals, rather than changing how long the sensor is awake, since cutting the waking period short could cause critical services, such as data acquisition, to fail.

As with power management, it appears that time synchronization is emerging in TinyOS as a specialized abstraction, with mechanism provided by the operating system and policy by the application. Applications have a varying set of time synchronization requirements, so incorporating their behavior in the service is beneficial. Gradual time shifting is suitable in some situations, while others require sudden shifts to the correct time.

The fact that applications fail when time-synchronization changes the underlying clock has been observed before; systems such as NTP [33] work around this problem by slowly adjusting the clock rate to synchronize it with neighbors. The NTP approach has the potential to introduce errors in an environment as time-sensitive as TinyOS, where a timer that fires even a few milliseconds earlier than expected can cause radio or sensor data to be lost.

Some applications may prefer to focus on robustness rather than maximizing the precision of time synchronization, as most algorithms proposed in the literature [16,18] and included in the TinyOS repository strive to do. For example, TinyDB only requires time synchronization to millisecond fidelity, but also requires a rapid settling time. A general abstraction that fit all possible needs would do much more work than TinyDB needs. Simple, specialized abstractions are a natural way to address these types of services.

6 Discussion

We focus now on what can be distilled from this broad study of TinyOS's networking software. We first revisit the four classes of abstractions that have emerged, discussing the members of each class. We also observe that there are design techniques commonly used in TinyOS, but which are not common to more general purpose systems. We then contrast the observed design goals of sensor networks and Internet-based systems.

6.1 Abstractions

We place the observed abstractions into one of four classes: general, specialized, in-flux, or absent. In this last class we note two abstractions that one might expect, based on literature and other networking systems, but were absent in the sources we examined.

6.1.1 General Abstractions

We noted several *general* networking abstractions, i.e, widely used and support by both TinyOS mechanisms and policies. The AM abstraction (Section 3.2) has remained stable since the earliest TinyOS work [23], and most network applications use it either directly or indirectly. This is not entirely surprising, as communication is the core service offered by TinyOS.

Another reason for the stability of AM is that the interface is very simple and lightweight. Other abstractions can easily provide it, in order to be compatible with pre-existing code. For instance, S-MAC provides an implementation of AM on top of its interface, allowing its use by existing programs without modification, despite that fact that it also provides another messaging interface.

A second set of general abstractions have emerged for tree-based routing, particularly the Send and Intercept interfaces shown in Figure 3. Implementations from Berkeley (Route) and Intel-Portland (HSN, AODV, DSDV) use this interface, and major applications, including TinyDB, have been reimplemented to make use of it. From a networking perspective, this is an important development, as it allows applications make use of a variety of different multi-hop implementations without source code

modification.

6.1.2 Specialized Abstractions

Specialized abstractions, i.e., those where TinyOS provides mechanisms and applications provide policies, have appeared for both power-management and time-synchronization. In many cases, it is possible to conceive of completely general versions of these abstractions, and general purpose operating systems often provide some version thereof. However, as with the effort to build a general version of time synchronization in TinyOS, general abstractions of some services are very hard to get right. This is because the requirements of applications vary dramatically: some applications and services need time synchronization that is accurate to within a few milliseconds with a small set of other nodes (e.g., TDMA), while others depend on a globally synchronized clock that is much less accurate (e.g., TinyDB). Many sensor network applications have long sleep periods, which provide natural, application-specific points for clock adjustment.

6.1.3 In-Flux Abstractions

A third class of abstractions we consider to be *in-flux*, commonly found but changing between applications and hardware versions, often in conflicting fashion.

One such abstraction is epidemic propagation. Both TinyDB and Maté use this technique to reliably disseminate code (capsules or queries) through a network; we believe that this abstraction is important for sensor networks and expect that it will see further use, e.g., for multi-hop network reprogramming. Many applications propagate commands that need to be received everywhere through naïve flooding. However, no established interface has emerged for this dissemination. This abstraction is a case where the literature [8,35] provides clear evidence of the shortcomings of flooding and its variants. Based on our observations, however, systems have suffered with these problems instead of adapting and adjusting solutions from prior work to the different constraints wireless sensor networks pose.

An abstraction that is surprisingly in flux is the radio MAC. We find considerable variation in how the channel activity is sensed, the use of control packets (RTS/CTS and ACK) per data packet, backoff, power management, link estimation, queuing, and assumptions about message size and traffic pattern. Additionally, each generation of hardware presented a distinct set of requirements for channel coding, start-symbol detection, alignment, and data transfer. We anticipate that the recent appearance of 802.15.4 radios will stabilize work in this regard; being an accepted IEEE standard gives it legitimacy and wide use. For media access protocols to stabilize they will need to have cross-layer control interfaces, discussed below, so they can be specialized to the particular needs of the application and network layer.

Routing from an arbitrary source to an arbitrary destination appears in only a few applications and involves a small subset of source-destination pairs within the context of substantial dissemination and collection traffic. Currently, it is realized out of the tree-based structures that are used for those other patterns.

6.1.4 Absent Abstractions

Finally, we note a few abstractions that we expected to find in TinyOS based on our reading of the sensor network literature, but that were absent in the code base.

One example is distributed cluster formation, about which there has been a large amount of publication in the ad-hoc and sensor network communities [12,3,5]. Instead we find dissemination using simple best-effort broadcasts and collection using continually monitored trees. The prime exception is the Intel DSDV implementation, which builds clusters using an available energy metric. The absence here may stem from the complexity of maintaining 2-hop neighbor lists with low-power radios that are heavily influenced by environmental factors.

Another abstraction missing from TinyOS is incoming (receive) queues. Applications generally handle message reception by accepting a message from the radio stack, processing it, and returning it back to the radio stack. Packets are typically dispatched to components that do a particular, simple operation on the contents. Compared to their communication bandwidth, motes have an abundance of CPU cycles; they can process messages at the rate they receive them.

6.2 Common Techniques

In addition to these abstractions, we observe certain design *techniques* that have been widely and successfully employed in TinyOS. We strive, in particular, to identify common approaches that facilitate program design and engineering or enhance performance. We consider those strategies that are prevalent in the TinyOS codebase, and which the preceding sections suggest will continue to be important in future sensor network software systems.

6.2.1 Communication Scheduling and Snooping

Two conflicting techniques that are widely used in TinyOS, and seem particularly important for sensor networks, are *communication scheduling* and *snooping*. Communication scheduling, as discussed in Section 5.1, refers to disabling the radio except during pre-arranged times when a pair of nodes expect to exchange data. It may also involve frequency or code division. In contrast, snooping refers to receiving packets that might not even be destined for a node, to acquire network neighborhood information or learn about new processing tasks (as with queries in TinyDB or program capsules in Maté); snooping is an essential part of the epidemic algorithms we noted throughout TinyOS.

Snooping tends to reduce the overall communication burden on the network, but requires the node to spend energy listening to its radio and receiving packets - exactly what scheduled communication avoids. While always-on and totally scheduled represent extreme points, we find that applications tend to strike a balance using partially scheduled techniques. For instance, TinyDB leaves the radio on for longer than a single transmission time so that some snooping can occur. Several TDMA schemes are in development [21], but some use the time slots as a heuristic to make collision improbable (using CSMA within slots), instead of assuming the time slot allocation is absolutely collision free. Applications may desire to open up additional `snooping slots'.

6.2.2 Cross-Layer Control

One approach that has emerged as particularly effective in the sensor network domain is the general technique of *cross-layer control*. An example of this is the way in which many TinyOS routing stacks share network neighborhood information between link state and network layers. Exposing state from a lower level (the link state layer) to a higher level (the network layer) avoids duplicating data in multiple layers, conserving RAM. Another example is the provision by the network stack of low-level information, such as received signal strength, to higher layers. Promiscuous communication works similarly: non-locally addressed packets overheard by the network layer are exposed to the application layer to allow it to avoid unnecessary communication.

Cross-layer techniques also work in the opposite direction: higher level components (frequently the application), can use logic in lower components to improve performance. For example, applications write timestamps in time synchronization messages just as the stack sends them using a hook the stack provides. In an extreme case, one radio stack exposes control of MAC layer functionality and parameters.

One reason that these techniques are so effective in sensor networks, and TinyOS in particular, is that each node is generally dedicated to a single task, allowing the application to choose which instance of a particular abstraction (e.g., S-MAC vs AM) it prefers without concern for the operation of other applications. This approach also supports the most important cross-layer technique that has emerged in TinyOS, application control of network services such as time-synchronization and power management. This control is necessary, as the application is the only software component capable of orchestrating the activities of all of the device's subsystems. This is an example of a case where TinyOS has been particularly successful at translating a stated goal of the project into reality; early work states [17]: "Without the traditional layers of abstraction dictating what capabilities are available, it is possible to foresee many novel relationships between the application and the underlying system."

Of course, cross-layer techniques are not new; they have been widely used in the networking and operating systems communities [4,34]. The resource constrained nature of sensor networks, however, makes these techniques particularly important. Without them, it is unlikely that there would be sufficient RAM or radio bandwidth available to

develop the complex systems that have begun to emerge. Moreover, we find that these techniques are used to enhance control, rather than simply to optimize frequent paths.

6.2.3 Static Resource Allocation

The third technique apparent in TinyOS is the notion of *static resource allocation*, or allocating buffers for the network, sensors, UART, and other OS services at compile-time. The prevalence of this static allocation in TinyOS is somewhat controversial, as RAM is valuable. To elucidate the importance of this technique, we consider a critical design parameter of any event-driven system: event-arrival rate.

Assuming that processing resources are sufficient, queue size (n) is determined by the maximum event-arrival rate (r), the number of events processed simultaneously (c), and the processing time per event (t). From Little's Law, we readily derive $n = rt + c$. Many TinyOS applications have processing times much smaller than the inverse of the arrival rate, and process one event at a time, which implies n=2 (one buffer for the current object and one for the next arrival). This is the root reasoning behind the buffer-swapping policy of the network stack; i.e. swap these two buffers on every arrival.

This reasoning argues against dynamic buffer allocation, which assumes that another buffer is always available. Since this is not true given the small amount of memory in today's sensor networks, it is much better to statically allocate the right number of buffers, which is the policy generally adopted in the TinyOS community.

This also leads to better composition, since components reserve the amount of memory they need, making total memory requirements checkable at compile-time. In contrast, two components that depend on dynamic buffer allocation may work fine alone, but not as well together as they unpredictably run out of memory. Static allocation can be wasteful when worst case requirements are much greater than the expected case. For example, TinyDB statically allocates 17 separate send buffers, but it is very unlikely that these would all used simultaneously; if the system were so overloaded, not having a buffer to allocate would be a minor issue.

Finally, it should be noted that although we believe the policy of statically allocating memory is the right approach for sensor networks. TinyOS provides limited tools for understanding static memory needs. Improved profilers or simulation tools that could predict such needs, such as recent work on computing stack usage [39], would be of great value to the community.

6.3 EmNets vs. the Internet

RFC 1958 ("Architectural Principles of the Internet") reads:

> "However, in very general terms, the community believes the goal is connectivity, the tool is the Internet Protocol, and the intelligence is end to

> end rather than hidden in the network ... connectivity is its own reward, and is more valuable than any individual application" [1]

Examined in this light, the networking abstractions that have emerged in sensor networks are the result of more than chance. They differ from traditional Internet abstractions not only because of resource constraints, which might change, but because of a very different set of goals and principles.

End-to-end connectivity is not the primary goal. Unlike the Internet, which is a collection of independent end points that share a common routing infrastructure, sensor networks are homogeneous systems deployed for an application-specific and collaborative purpose. Every node is both a sensor and a router. The relative costs of communication and computation push the architecture from end-to-end logic to in-network processing. Network neighbors are generally physically close; correspondingly, their sensor readings are often related. Additionally, a wireless medium, combined with geographic proximity, correlates network traffic between neighbors.

7. Conclusion

In this paper, we were able to classify the most prominent abstractions in TinyOS based on the degree of consensus shown in our community-wide study of the code base. Among the more mature, general abstractions are active messages and tree-based routing. However, most abstractions are still specialized and application specific, in flux, or largely missing despite being in the literature. We observe a trend where application specific abstractions become gradually become more general over time. This is illustrated, for example by the emergence of tree-based routing in TinyOS.

One major reason for the absence of consensus is the unusual degree to which application specialization matters. This appears to be due to a few factors: first, each mote runs only one application at a time, eliminating the need for shared abstractions; second, power management affects all levels of the system and is essentially always application specific; and third, limited resources lead to specialized implementations offering greater efficiency than their generalized counterparts. The last effect is evident in time synchronization, multi-hop routing, and buffer allocation. A fourth factor is that many applications have real-time requirements that mandate precise control over timing throughout the application, reducing the utility of off-the-shelf components.

We also found several techniques that work well in these systems. The two most obvious are cross-layer control and static resource allocation. The former allows better use of resources and more control over timing (e.g. time synchronization), and is well-supported by the "wiring" language in TinyOS, which exposes the layering and encourages interposition and cross-layer thinking. Part of the success of cross-layer control is the support it provides to application specialization. Static resource planning is more robust, more modular, and forces the author to think about the whole program,

including all of its resources.

In conclusion, we see that wireless sensor networking is driven by three differentiating factors: power management, limited resources and real-time constraints. These factors have driven the development of the abstractions and techniques seen above. It remains to be seen how lasting and wide-ranging these trends will be.

8. Appendix: Code Sources

The following source code repositories were used to collect the data for this paper.

CENS. From http://cvs.cens.ucla.edu/viewcvs/viewcvs.cgi/tos-contrib/. Includes RBS [16] time synchronization.

Rutgers. From http://www.cs.rutgers.edu/dataman/FourierNet/tos10/distro.html. Includes an ad-hoc positioning system and based on ranging code from Vanderbilt.

sf.net vert. From http://sourceforge.net/projects/vert/. Includes an activation tracking demo and "virtual real time" demo from UVA.

TinyOS Contrib. From http://sourceforge.net/projects/tinyos/ in the `tinyos-1.x/contrib` directory. Includes PRIME, S-MAC, TinyDiff, SensorIB, tinydb, and hsn.

TinyOS CVS and main UC Berkeley repository; includes all UC Berkeley files. From http://sourceforge.net/projects/tinyos/ in the `nest`, `tinyos`, `tinyos-1.x` and `tos` directories. The TinyDB application is available at `tinyos-1.x/tos/lib/TinyDB`.

TinyOS Documentation Project. From http://ttdp.org.

Acknowledgements

This work was supported, in part, by the Defense Department Advanced Research Projects Agency (grants F33615-01-C-1895 and N6601-99-2-8913), the National Science Foundation (grant NSF IIS-033017), California MICRO program, and Intel Corporation. Research infrastructure was provided by the National Science Foundation (grant EIA-9802069)

9. Bibliography

1

RFC 1958: Architectural Principles of the Internet, 1996.

2

H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han.
MANTIS: system support for MultimodAl NeTworks of In-situ Sensors.
In *Proceedings of the 2nd ACM International Conference on Wireless Sensor Networks and Applications*, pages 50-59. ACM Press, 2003.

3

A. D. Amis, R. Prakash, T. H. P. Vuong, and D. T. Huynh.
Max-min d-cluster formation in wireless ad hoc networks.
In *Proceedings of IEEE INFOCOM*, March 2000.

4

T. Anderson, B. Bershad, E. Lazowska, and H. Levy.
Scheduler activations: Effective kernel support for the user-level management of parallelism.
ACM Transactions on Computer Systems, 10(1):53-79, February 1992.

5

S. Bandyopadhyay and E. J. Coyle.
An energy efficient hierarchical clustering algorithm for wireless sensor networks.
In *Proceedings of IEEE INFOCOM*, 2003.

6

P. Buonadonna, J. Hill, and D. Culler.
Active message communication for tiny networked sensors.
Available from http://www.cs.berkeley.edu/~jhill/cs294-8/ammote.ps.

7

A. Cerpa, N. Busek, and D. Estrin.
SCALE: A tool for Simple Connectivity Assessment in Lossy Environments, September 2003.

8

B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris.
Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks.
In *Proceedings of ACM MOBICOM*, Rome, Italy, July 2001.

9

B. Chen and R. Morris.
L+: Scalable landmark routing and address lookup for multihop wireless networks.

Technical Report 837, MIT LCS, March 2002.

10

D. D. Clark and D. L. Tennenhouse.
Architectural considerations for a new generation of protocols.
In *Proceedings of SIGCOMM*, pages 200-208. ACM Press, 1990.

11

D. E. Culler, J. Hill, P. Buonadonna, R. Szewczyk, and A. Woo.
A network-centric approach to embedded software for tiny devices.
Lecture Notes in Computer Science, 2001.

12

B. Das and V. Bharghavan.
Routing in ad-hoc networks using minimum connected dominating sets.
In *Proceedings of the International Conference on Computing*, 1997.

13

D. De Couto, D. Aguayo, J. Bicket, and R. Morris.
A high-throughput path metric for multi-hop wireless routing.
In *Proceedings of ACM MOBICOM*, San Diego, California, Sept. 2003.

14

A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson.
Epidemic algorithms for replicated database maintenance.
In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1-12. ACM Press, 1987.

15

J. Elson, S. Bien, N. Busek, V. Bychkovskiy, A. Cerpa, D. Ganesan, L. Girod, B. Greenstein, T. Schoellhammer, T. Stathopoulos, and D. Estrin.
Emstar: An environment for developing wireless embedded systems software.
Technical Report 0009, CENS, Mar. 2003.

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. Estrin et al.
Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers.
National Acedemy Press, Washington, DC, USA, 2001.

18

S. Ganeriwal, R. Kumar, and M. B. Srivastava.
Timing-sync protocol for sensor networks.
In *Proceedings of SenSys*, 2003.
To Appear.

19

D. Ganesan.
TinyDiffusion Application Programmer's Interface API 0.1.
http://www.isi.edu/scadds/papers/tinydiffusion-v0.1.pdf.

20

D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker.
Complex behavior at scale: An experimental study of low-power wireless sensor networks.
Technical Report 02-0013, UCLA Computer Science Division, Mar. 2002.

21

L. Gu.
PRIME: A Collision-Free MAC Protocol.
Draft. Available from http://www.cs.virginia.edu/~lg6e/MAC.pdf.

22

J. Hill.
System Architecture for Wireless Sensor Networks.
PhD thesis, UC Berkeley, May 2003.

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 ASPLOS*, pages 93-104, Boston, MA, USA, Nov. 2000.

24

B. Karp and H. T. Kung.
GPSR: greedy perimeter stateless routing for wireless networks.
In *Proceedings of ACM MOBICOM*, pages 243-254, Boston, MA, USA, 2000.

25

O. Kasten and J. Beutel.
BTnode rev2.2.
http://www.inf.ethz.ch/vs/res/proj/smart-its/btnode.html.

26

R. Kling.
Intel research mote.
http://webs.cs.berkeley.edu/retreat-1-03/slides/imote-nest-q103-03-dist.pdf.

27

A. Ledeczi, K. Frampton, G. Simon, and M. Maroti.
Shooter localization problem in urban warfare.
http://www.isis.vanderbilt.edu/projects/nest/documentation/Vanderbilt_NEST_Shooter_SanDiego.ppt.

28

M. Leopold, M. B. Dydensborg, and P. Bonnet.
Bluetooth and sensor networks: A reality check.
In *SenSys '03*, Los Angeles, California, Nov. 2003.

29

P. Levis, N. Patel, D. Culler, and S. Shenker.
Trickle: A self-regulating algorithm for code maintenance and propagation in wireless sensor networks.
In *First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI 2004)*.

30

S. R. Madden.
The Design and Evaluation of a Query Processing Architecture for Sensor Networks.
PhD thesis, UC Berkeley, Decmeber 2003.
http://www.cs.berkeley.edu/~madden/thesis.pdf.

31

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.

32

Michael Hamilton et al.
Habitat sensing array, first year report.
http://cens.ucla.edu/Research/Applications/habitat_sensing.htm, 2003.

33

D. L. Mills.
Internet time synchronization: The network time protocol.
In Z. Yang and T. A. Marsland, editors, *Global States and Time in Distributed Systems*. IEEE Computer Society Press, 1994.

34

D. Mosberger and L. Peterson.
Making paths explicit in the Scout operating system.
In *Proceedings of the USENIX OSDI 1996*, October 1996.

35

S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu.
The broadcast storm problem in a mobile ad hoc network.
In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 151-162. ACM Press, 1999.

36

S. Park, A. Savvides, and M. B. Srivastava.
SensorSim: a simulation framework for sensor networks.
In *Proceedings of the 3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 104-111. ACM Press, 2000.

37

C. E. Perkins and E. M. Royer.
Ad hoc on-demand distance-vector (aodv) routing.
In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, 1999.

38

A. Rao, S. Ratnasamy, C. Papadimitriou, S. Shenker, and I. Stoica.
Geographic routing without location information.
In *Proceedings of ACM MOBICOM*, September 2003.

39

J. Regehr, A. Reid, and K. Webb.
Eliminating stack overflow by abstract interpretation.
In *Proceedings of the Third International Conference on Embedded Software (EMSOFT 2003)*, 2003.

40

C. Sharp and S. Shaffert.
Road to pursuit/evasion on a sensor network.
http://robotics.eecs.berkeley.edu/~cssharp/NEST-demos-overview-Aug2003.ppt, aug 2003.

41

K. Whitehouse and D. Culler.
Calibration as parameter estimation in sensor networks.
In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*.

42

A. Woo and D. Culler.
Evaluation of efficient link reliability estimators for low-power wireless networks.

Technical Report UCB//CSD-03-1270, U.C. Berkeley Computer Science Division, September 2003.

43

A. Woo, T. Tong, and D. Culler.
Taming the underlying challenges for reliable multihop routing in sensor networks.
In *SenSys '03*, Los Angeles, California, Nov. 2003.

44

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.

45

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.

... UCLA[1]

In `tinyos-1.x/contrib/Timesync-NESL-UCLA/`

... Vanderbilt[2]

In `tinyos-1.x/contrib/vu/`

... Berkeley[3]

In `tinyos-1.x/beta/TimeSync/`