Resource Aware Programming in the Pixie OS

Konrad Lorincz, Bor-rong Chen, Jason Waterman, Geoff Werner-Allen, and Matt Welsh
School of Engineering and Applied Sciences, Harvard University, Cambridge, MA 02138
{konrad,brchen,waterman,werner,mdw}@eecs.harvard.edu

ABSTRACT

This paper presents Pixie, a new sensor node operating system designed to support the needs of data-intensive applications. These applications, which include high-resolution monitoring of acoustic, seismic, acceleration, and other signals, involve high data rates and extensive in-network processing. Given the fundamentally resource-limited nature of sensor networks, a pressing concern for such applications is their ability to receive feedback on, and adapt their behavior to, fluctuations in both resource availability and load.

The Pixie OS is based on a dataflow programming model based on the concept of resource tickets, a core abstraction for representing resource availability and reservations. By giving the system visibility and fine-grained control over resource management, a broad range of policies can be implemented. To shield application programmers from the burden of managing these details, Pixie provides a suite of resource brokers, which mediate between low-level physical resources and higher-level application demands. Pixie is implemented in NesC and supports limited backwards compatibility with TinyOS.

We describe Pixie in the context of two applications: limb motion analysis for patients undergoing treatment for motion disorders, and acoustic target detection using a network of microphones. We present a range of experiments demonstrating Pixie's ability to accurately account for resource availability at runtime and enable a range of both generic and application-specific adaptations.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]; D.1 [**Programming Techniques**]; C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms

Design

Keywords

Resource-Aware Programming, Resource Reservations, Wireless Sensor Networks

SenSys'08, November 5–7, 2008, Raleigh, North Carolina, USA.
ACM 978-1-59593-990-6/08/11.

1. INTRODUCTION

Resources in sensor networks are precious. This is especially true in application domains where data rates and computational demands can outstrip the limited capabilities of low-power sensor nodes. Tuning a sensor network application to operate efficiently, especially given fluctuations in resource availability and load, is a difficult challenge, often involving cross-layer adjustments to duty cycle, sampling rates, computational fidelity, and communication patterns. Existing sensor network operating systems provide little help in this regard, exposing a wide range of low-level primitives for resource management, but little guidance in terms of how those primitives should be used in a holistic fashion.

We need a better approach to programming sensor networks that gives applications *awareness* of resource availability, as well as the ability to *adapt* their behavior in intelligent ways. We argue that to a large extent, adaptivity will be closely tied to the application logic, as it involves complex tradeoffs between lifetime, bandwidth usage, and data quality. In general, it is not possible to automate this process for all applications. Rather, our goal is to provide a programming model that gives developers a much more intuitive set of tools for managing resources within their applications.

In this paper, we describe *Pixie*, a new operating system for sensor nodes that enables *resource-aware programming*. In Pixie, a sensor node has direct knowledge of available resources, such as energy, radio bandwidth, and storage space, and can control resource consumption at a fine granularity. There are three key components of the Pixie system. The first is a *dataflow programming model* that structures applications as a graph of interconnected *stages*, similar to systems such as Click [30], Eon [48], and Flask [37]. The use of a dataflow model gives the operating system both visibility and control over the application's resource usage. Second, Pixie uses the concept of *resource tickets*, an abstraction representing a discrete allocation of physical resources. Tickets form the basis for resource awareness and control in Pixie. The final component is a set of *resource brokers*, reusable software modules that encode policies for resource management. An application can delegate its resource management decisions to an appropriate broker that handles the low-level details of resource ticket allocation.

In any adaptive system, one of the key questions that arises is whether an application should be responsible for its own resource management, or whether this process can be automated. On one hand, application-aware adaptation, such as that supported by Odyssey [41], offers the most flexibility and efficiency, but may involve a great deal of programmer burden. Systems such as Eon [48] and ECOsystem [56] shield applications from this burden, but tie applications to a specific set of policies. Pixie strikes a balance by *enabling* resource awareness at the application level without *requiring* it. Furthermore, Pixie decouples resource management mecha-

nisms, provided by resource tickets, from the policies, which may be implemented within the application or provided by resource brokers. In this way, Pixie applications can consist of a mixture of "resource-aware" and "naïve" components.

We explore Pixie's design in the context of two demanding applications. The first involves the use of wearable sensors for limb motion analysis, capturing high-resolution accelerometer and gyroscope data to aid in the rehabilitation of patients with Parkinson's Disease or recovering from a stroke [43]. The second application involves a network of sensors for acoustic target detection, such as that used in wildlife monitoring [1], shooter localization [47], and vehicle tracking [23]. These two applications are representative of a broad class of resource-intensive, adaptive systems that Pixie is intended to support.

Pixie makes the following contributions. First, Pixie enables resource awareness by making resources a first-class entity in the programming model. We present the design and architecture of Pixie as well as a prototype implementation running on the TMote Sky and iMote2 sensor platforms. Second, Pixie incorporates efficient runtime estimation of available resources, such as energy and radio bandwidth. Our approach to energy usage estimation is based on a simple software model and performs accurate energy estimation on standard mote platforms with no additional hardware support. Third, Pixie's resource broker abstraction enables a broad range of reusable adaptation strategies, including adaptive duty cycling, varying computational fidelity, and tuning radio bandwidth, while shielding application code from the details of low-level resource management. Finally, we present a detailed evaluation of Pixie in the context of two applications, demonstrating that Pixie achieves high efficiency, accurate resource estimation, and effective adaptivity across a range of energy and bandwidth constraints. We first introduced the Pixie design in an earlier position paper [34]. In this paper, we build upon our earlier work and provide a much more detailed description of the Pixie architecture and implementation, as well as a thorough evaluation in the context of two target applications.

The rest of the paper is organized as follows. In Section 2, we lay out the motivation and background for the Pixie approach. We describe Pixie's architecture in detail in Section 3, and present our prototype implementation in Section 4. Section 5 describes our two example applications and Section 6 presents results from our evaluation. Section 7 summarizes related work, and Section 8 presents future work and concludes.

2. BACKGROUND AND MOTIVATION

An increasing number of wireless sensor network applications involve high data rates, high data fidelity requirements, and computationally intensive processing. Examples of applications in this class include vibration monitoring of bridges and buildings [7, 8, 42]; seismic monitoring of fault zones and volcanoes [25, 53]; acoustic monitoring of animal habitats [1, 35]; and body sensor networks for monitoring activity and movement [9, 16, 43].

These applications present several unique challenges to application design. First, they involve potentially high load on computational and communication resources, and must therefore be extremely resource-efficient. Second, these applications exhibit load fluctuations in response to environmental stimuli: they cannot assume simple periodic operation. Third, resource availability may fluctuate as well, due to changes in RF interference and congestion, node mobility, and energy drain. These challenges point to the need for a programming abstractions to help application developers achieve the best use of limited sensor node capabilities under variable load and resource conditions.

To a large extent, the sensor networking community has not yet required general-purpose solutions bridging the gap between application demands and available resources: many applications have been designed to match sensor node resource constraints without requiring extensive runtime adaptation. Traditional sensor network applications are designed to operate at low data rates and as a result are "embarrassingly periodic" in nature, largely reducing resource management to a static problem.

We believe that in order to open up sensor networks to non-experts wishing to leverage the technology for their applications, we must do something about this problem and offer a programming model that treats resources as a first-class programming primitive. Currently, few tools exist to reason about resource usage in sensor network applications at a high level. Operating systems such as TinyOS [24], SOS [22], Contiki [11], and Mantis OS [6] provide low-level mechanisms for controlling hardware states, but do little to simplify resource management at the application level. Nano-RK [14] provides real-time guarantees through *static* resource reservations based on offline estimates of CPU time, packet rates, and sampling intervals used by an application. However, this approach fails to address dynamically-varying load or fluctuations in resource availability that arise at runtime.

The conventional structure used to build sensor network applications makes it difficult to automate many aspects of resource management. While component-oriented design, such as that used in TinyOS and NesC [17], facilitates composition and reuse, any sophisticated application becomes a tangled "web of components" that makes it difficult to inspect or tune resource usage. A similar lack of transparency pervades the conventional threading model (adopted by Mantis, for example), giving the OS little control over application behavior.

We argue that these problems demand a rethinking of the fundamental OS structure and programming interfaces used to manage resources. In this paper, we propose a new operating system, called *Pixie*, that combines the use of a dataflow-oriented programming model as well as a rich set of interfaces for observing and reacting to the state of resources on the node. Pixie is implemented in NesC and makes heavy use of TinyOS' drivers, and retains limited backwards-compatibility with legacy TinyOS code. We emphasize that Pixie is focused on resource management at the individual sensor node level, not the network as a whole. We believe that the Pixie OS exposes enough information and control to permit applications to coordinate resource-management decisions across the network.

Unlike systems such as Eon [48], which attempts to automate resource management decisions, in Pixie our philosophy is to give applications direct visibility and control over resource usage at runtime *should they need it*. This is provided in the form of a *resource ticket* abstraction that represents the right to use a given quantum of resources, and a set of resource allocators that manage each physical resource in the system. In cases where applications do not wish to deal with this level of detail, we introduce resource brokers that provide commonly-used, reusable policies that can be linked into the application.

There is an inherent tradeoff between programmer burden and efficiency in the face of changing resource conditions and load. Our challenge is to strike the right balance along this spectrum, permitting application developers to build reusable components that will operate well under a range of conditions, while limiting the amount of complexity and low-level details that application code must manage. We recognize that this approach may not be as resource-efficient as a carefully hand-coded application, but it should be significantly easier to program and far more intuitive to reason about.

3. PIXIE ARCHITECTURE

An overview of the Pixie architecture is shown in Figure 1. In Pixie, applications are structured as a dataflow graph of *stages*, where each stage represents some unit of computation or I/O. Decomposing the application into a stage graph gives the OS both visibility into and control over resource demands and data flow within the application. Application stages interact with *resource brokers*, which are specialized stages responsible for providing feedback on and arbitrating access to resources, such as energy, radio bandwidth, or flash storage. Pixie provides a toolkit of brokers implementing different resource-management policies. At the lowest level, Pixie's *resource allocators* manage access to physical resources and estimate resource availability at runtime.

Central to the Pixie design is the concept of a *resource ticket*, which represents a bounded-duration right to consume a given quantity of a given resource. Tickets are the basic "currency" exchanged by application code and brokers, enabling both resource awareness and compositional resource management policies.

Pixie is designed for resource-limited sensor nodes (such as the TMote Sky, MicaZ, and iMote2 platforms), in which a single application runs on the node at a time. Further, we assume that application code is cooperative and does not attempt to undermine the OS's resource management abstractions: Pixie's role is advisory rather than coercive. In the following sections, we detail each aspect of the Pixie design.

3.1 Dataflow programming model

Pixie applications are structured as a dataflow graph of *stages*, using a model similar to Click [30]. Stages are connected by *edges* that are linked to each stage's input and output *ports*. By default, each stage has a single input port and single output port, although multiple ports may be used. Edges themselves perform no queuing; a special *queue* stage can be introduced into the stage graph if needed.

We chose to adopt this model in Pixie for several reasons. First, dataflow maps well onto the structure of many sensor network applications, as demonstrated by systems such as Eon [48], Tenet [19], WaveScope [40], and Flask [37]. Second, compared to a conventional process model or the TinyOS component model, dataflow graphs represent data and resource dependencies of individual components explicitly, giving the OS greater visibility and control over the application's behavior. Third, dataflow graphs naturally support interpositioning, which (as described below) is a core mechanism used by Pixie's resource brokers to affect control over the application's resource usage.

Memory model: In Pixie, memory objects are represented as *memrefs* which are a dynamically-allocated, contiguous region of memory, along with a reference count. Memrefs are the basic data type carried by edges between stages. Memrefs are untyped and stages producing and consuming memrefs must agree on their format. When a stage is scheduled, it is passed a memref (from any one of its input ports) as input. Pushing a memref onto an output port increments the refcount for each stage receiving the memref. A stage must explicitly *release* a memref, decrementing its reference count, if it no longer wishes to access the memref (e.g., because it has passed it downstream or consumed it). Deallocation is performed implicitly when a memref's reference count drops to zero.

Scheduling: The Pixie dataflow model admits a range of scheduling algorithms and techniques. Our default implementation performs a simple depth-first traversal of the stage graph, directly invoking downstream stages using a procedure call, thereby minimizing cross-stage overhead. Traversals are initiated at *source stages*, such as sampling or timers, and terminate at *sink stages* which either do not push data to an output port, or have no output ports. A *queue stage* acts as both a sink stage and a source stage; the queue initiates graph traversals downstream when the stage is non-empty. Each source stage has an associated *priority*, which may be altered at runtime. The scheduler initiates a graph traversal at the source stage with the highest priority, and ties are broken using round-robin.

A stage may indicate to the scheduler that it is *blocked*, for example, due to lack of needed resources or pending an asynchronous I/O operation. The stage may later signal that it is *unblocked*. Blocking a stage causes all upstream stages in the stage graph to be implicitly blocked, until a source stage is reached. To prevent blocking from stalling upstream stages, applications generally introduce a queue upstream from the blocking point, allowing input data to accumulate at that point in the stage graph. For example, Pixie's radio transmission stage (which blocks when it is busy transmitting a packet and waiting for an acknowledgment) provides a queue that accumulates outgoing packets.

Example: Figure 1 shows an example Pixie application, closely related to the motion-analysis system described in Section 5. The sampling component serves as a source stage, producing memrefs containing six channels of accelerometer and gyroscope data. The data is passed to an activity filter stage, which drops data when no motion is detected. The raw data is logged by the flash I/O stage. The bandwidth broker, as described below, sends data to one or more feature extraction stages that process the signals, based on the current estimate of radio bandwidth availability. Each feature extraction stage transmits data to the base station through the radio transmission stage. The radio and flash stages may block while performing a transmission or I/O operation, so they require a queue for buffering incoming data. The use of resource allocators and tickets shown in the figure are described in the next section.

3.2 Resource tickets

A *resource ticket* is the basic abstraction for resource management in Pixie. A ticket $\langle R, c, t_e \rangle$ represents a right to consume up to c units of resource R until an expiry time t_e. Tickets are created by *resource allocators*, one for each physical resource managed by the operating system, and are typically handled by *resource brokers*, described in Section 3.4 below. Tickets can be thought of as the resource "currency" managed by Pixie, somewhat related to resource containers [5] and the ticket abstraction used in lottery scheduling [52]. By basing all aspects of resource management on this single abstraction, Pixie permits a great deal of flexibility and composition of resource-management policies.

Tickets can only be generated by the corresponding resource allocator for a given resource, as described below. Tickets support several basic operations: *redeem*, *forfeit*, *revoke*, *join*, and *split*. A ticket is redeemed by a stage when it requires resources to perform an operation; for example, sending a radio packet requires a corresponding bandwidth ticket. Forfeiting a ticket informs the system that the resources are no longer needed. The system may revoke a ticket on or before its expiry time t_e; note that the expiry time is therefore a hint and not a guarantee as to the ticket's validity. Multiple tickets of the same resource can be joined into a single ticket. Joining two tickets retains the earlier of the two expiry times. Splitting a ticket likewise creates two tickets with the corresponding amount of resource, with identical expiry times.

An example of an energy ticket is $\langle energy, 700mJ, 10sec \rangle$, while a radio bandwidth ticket might be $\langle bandwidth, 5packets, 5sec \rangle$. Likewise, a storage ticket gives the right to store a certain number of bytes in flash, while a memory ticket gives the right to allocate a memref of a given size.

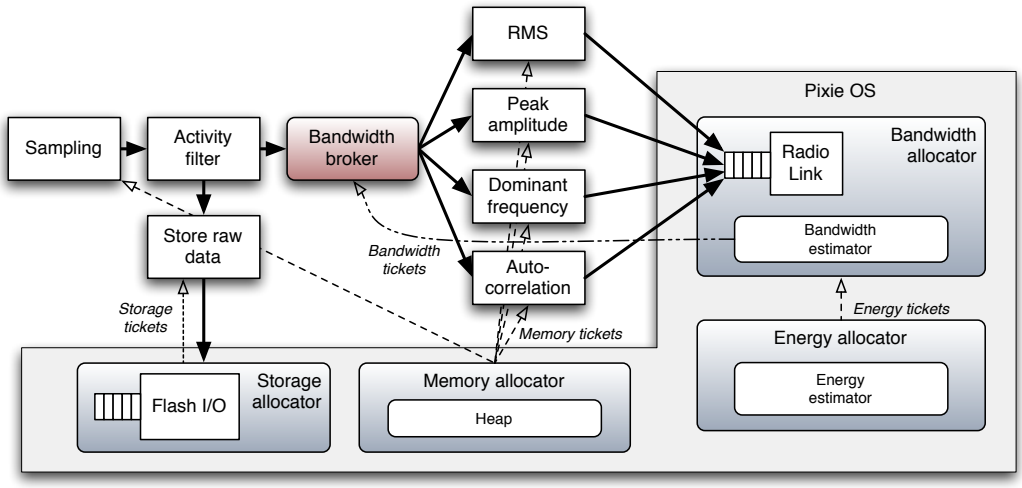

Figure 1: **An example Pixie application for limb monitoring using wearable sensors.** *Stages are represented as unshaded boxes and contain application-specific code. Resource allocators and brokers are components of the Pixie OS. Solid arrows represent data flow; dashed arrows represent resource ticket allocations. For clarity, not all components and arrows are shown in the figure.*

Tickets in Pixie decouple resource reservation from usage. This permits applications to make requests in advance of the need for a given resource, and take action based on the amount of resources that are available. For example, given a ticket for a certain amount of energy, the application can decide whether to perform a high-energy operation (such as sampling an expensive sensor) or a lower-energy alternative.

The granularity of the resources represented by a ticket, as well as the expiry time, depends on the physical resource involved. While radio bandwidth may fluctuate on short time scales (orders of seconds), energy and storage availability tend to be more stable. By limiting the amount of resources and expiry time in a ticket, the system can bound the amount of outstanding resource credit it is extending to applications. Ticket revocation permits the system to reclaim unused resources over time.

In our current design, resource tickets are not required for CPU usage; we rely instead on priority-driven scheduling to manage access to the CPU. In part, this is because requesting and managing tickets itself requires computation, and accounting for all CPU usage by a sensor node is extremely complex. We require applications to request energy tickets for performing computationally-intensive operations, such as those that would involve increasing the CPU clock frequency on platforms that support this operation.

In many cases, it is useful to couple a resource ticket with a data object that requires the ticket for its processing. Each memref can have an optional ticket attached to it that is passed transparently along with the object. The ticket may of course be split and redeemed as the data object flows through the stage graph. In this way, a Pixie stage graph represents flow of both data and resource allocations.

3.3 Resource allocators

Each physical resource managed by Pixie has a corresponding *resource allocator* that performs three functions: (1) Estimating the amount of available resource at runtime; (2) Allocating tickets for that resource on demand; and (3) Enforcing ticket redemptions. We currently employ resource allocators for radio, flash storage, energy, and memory, although the model can be readily extended to other physical resources. In our current model, sensor data acquisition is handled by the energy allocator.

Resource allocators simply mediate access to the physical resource that they control; they do not impose any policy on those allocations. Allocators perform first-come, first-served allocation of tickets as long as the requested resources are available. Allocators support two basic operations: *query* and *request*. The *query* operation allows a client to determine how much resource is currently available, such as the number of joules in the battery or free storage space in the flash. *request(c, t_e)* requests a ticket for c units of the resource with an expiry time of t_e. The allocator will reject the request if c exceeds the amount of available resource, or if t_e exceeds the allocator's *estimation horizon*, described below.

Allocators are also responsible for enforcing allocations, by requiring a valid ticket in order to consume the underlying resource. All direct access to a physical resource is performed by the corresponding allocator: for example, the Pixie storage allocator is responsible for performing all flash I/O.

Resource allocators may be *layered*. For example, both radio and flash I/O consume energy as well, and require a corresponding energy ticket. In this case, the radio and storage allocators are responsible for acquiring, and redeeming, energy tickets in order to perform an I/O operation. This implies that requesting, say, a storage ticket involves an (implicit) request for an energy ticket, and that the request may be denied due to lack of underlying energy availability, even if storage is plentiful.

An important function of allocators is *estimation* of the available resources at runtime. Due to the volatility of resource availability over time, each allocator has an *estimation horizon* that represents its ability to forecast future availability. For storage and memory, the horizon is effectively infinite. For energy, the horizon can be long (hours or days) depending on battery stability, estimation accuracy, and whether the sensor node can be recharged in the field (say by solar power). Radio bandwidth estimation has a short horizon, on the order of a few seconds.

3.3.1 Storage and memory allocators

The simplest resource allocators deal with flash storage and memory. The memory allocator maintains a fixed-size heap (allocated at boot time), while the storage allocator manages the node's flash, the size of which is also known at boot time. Requesting a storage or memory ticket reserves the corresponding amount of storage/mem-

ory until the ticket is redeemed or revoked. In the case of memory, allocating a memref is equivalent to atomically requesting and redeeming a corresponding memory ticket.

3.3.2 Energy allocator

Pixie's energy allocator performs runtime estimation of the sensor node's battery capacity and expected node lifetime. Energy metering can be performed using hardware support [13, 48, 50], with varying degrees of accuracy and overhead. In the absence of hardware support, Pixie employs a software-based metering technique that tracks the state of each hardware device on the sensor node and applies a model of the energy requirement for each hardware state. The energy consumption for each hardware component is combined to produce an estimate of the overall power consumption of the sensor node. Our approach is similar to that used in Contiki [12] and the PowerTOSSIM [46] simulator.

The Pixie energy metering component uses a set of hooks into low-level system components to track the energy state of each hardware device. For example, the energy meter receives upcalls from the CC2420 radio stack when the radio state switches between low-power idling, listening, and transmission modes. Likewise, upcalls are made from the flash I/O and CPU frequency scaling components to track their hardware state. Each state change triggers an update to the energy meter's estimate of the total energy usage. We use an empirical model of the current draw of each hardware device in different states, based on detailed measurements of the TMote Sky and iMote2 platforms. Our model is similar to that presented in [29].

Tracking CPU energy usage is somewhat more challenging, since we do not want to introduce expensive instrumentation into every task and interrupt handler execution. Instead, we assume a conservative, constant current draw for the CPU based on its current power setting. On MSP430-based platforms, CPU consumption averages about 1.9 mA. On XScale-based platforms (such as the iMote2) that support dynamic voltage scaling, we assume a constant current draw for each supported frequency level, and track frequency state changes, which are controlled by the energy allocator.

3.3.3 Bandwidth allocator

Pixie provides an abstract *radio link layer* that is intended to support a uniform interface for radio transmission and reception over different radio technologies and MAC layers, a goal shared by SP [45]. Our current prototype uses the default TinyOS 2.x CSMA MAC over 802.15.4, and supports reliable transmission using ARQ. Pixie uses low-power listening at the MAC layer [44]. Applications can control the listening duty cycle through the Pixie link layer interface, allowing for energy savings at the cost of additional packet delay or loss.

In sensor networks involving a nontrivial amount of radio communication, it is often beneficial to estimate the effective link capacity, allowing the application to adapt its behavior to available bandwidth. Given that link capacity can be affected by many factors, including RF interference, congestion, and node mobility, giving applications effective feedback on the state of the link for capacity estimation is very challenging. The problem of link estimation has been extensively studied [10, 49, 54], usually with the goal of picking the "best" link among several for the purpose of selecting an optimal routing path.

We observe that if we treat the link layer as a "black box," the problem reduces to estimating the maximum *packet injection rate*, defined as the rate at which packets can be pushed to the link layer without overflowing its bounded packet queue. This approach abstracts away the dynamics of the link layer's behavior and allows the application to only consider the rate at which it should generate outbound packets.

Pixie's bandwidth allocator estimates link capacity by measuring the *mean packet transmission delay* t_d, defined as the time for the link layer to transmit a packet (including MAC, transmission time, ACK reception, and any retransmissions), until an ACK is received or the ARQ threshold is reached. Note that this is a passive measurement that does not generate additional traffic, although it assumes that the application is periodically transmitting enough packets to keep the capacity estimate up to date. Although transmission delay is a function of the packet size, in a lossy environment we expect MAC and retransmission delays to dominate; we conservatively estimate t_d assuming large fixed-length packets.

To avoid overflowing the transmission packet queue, the application should avoid generating packets at a rate faster than $1/t_d$. For a given time window δ, the bandwidth allocator will issue no more than δ/t_d packet's worth of bandwidth tickets, each with an expiry time of δ sec. We have experimented with different time windows and find that δ = 1–5 sec works well in mobile settings with a high degree of link variation.

3.3.4 Estimating demand

To use resource tickets effectively, applications must estimate their demand for resources in order to request the appropriate ticket amounts. A full treatment of this problem is beyond the scope of this paper. Fortunately, resource demand estimation is a well-studied problem in the embedded, real-time, and mobile systems literature. In many cases, conservative estimates can be determined offline through simulation [31, 46] or code analysis [14, 38]. Demand can also be estimated by measuring the application's resource usage at runtime [15, 33, 48, 55]. It is worth noting that many common operations, such as transmitting a radio packet, reading or writing a flash block, or sampling a sensor typically consume a constant amount of energy; Pixie provides a set of macros to represent these amounts when requesting energy tickets. Estimating the CPU usage (and thereby energy requirements) of computationally-demanding stages can be performed using conventional offline profiling techniques. Also, in our experience it is not difficult for applications to estimate bandwidth and storage needs. While we recognize that demand estimation is not a trivial problem, our current approach assumes that one or more existing techniques are employed by application developers.

3.4 Resource brokers

Resource allocators and tickets provide a powerful, yet low-level interface to resource management. While this approach permits a great deal of flexibility, it is often cumbersome for applications to operate at this level. For this reason, Pixie introduces the concept of *resource brokers* that provide a high-level interface for mediating between application needs and underlying resource allocations. A resource broker is a specialized Pixie stage that requests and manages resource tickets on behalf of application code according to some policy. Brokers need not expose a uniform API, and indeed the interaction between the broker and application code is generally broker-specific. The use of brokers is entirely optional; applications are always able to circumvent them should they wish to perform allocations themselves.

Since brokers are stages, they can interpose directly on the application's dataflow path. As a result, the broker can inspect, redirect, discard, or refactor data flowing through it. As an example, consider a "switch" broker that directs data to one of a set of down-

stream stages based on energy or bandwidth availability. Another type of broker might decimate or degrade input data to match resource demands to current availability.

Pixie includes a small toolbox of standard brokers that we expect will find use in a range of applications. Developers are also free to implement their own brokers, thereby encapsulating resource-management decisions into a single component. Some of our standard brokers are described below.

Energy broker: The default energy broker doles out energy tickets according to a schedule that attempts to meet a given lifetime target for the sensor node. Each application stage registers its desire for energy *quanta* with the broker, and specifies its *priority* for energy allocation. Each quantum represents the amount of energy the stage requires to perform an (application-defined) unit of work. The broker delivers tickets to each stage according to its internal schedule and each stage's priority, ensuring that the ticket values match the requested quanta. This is similar to the token-bucket energy scheduler described by Banerjee *et al.* [3].

Given initial battery capacity E and lifetime target t_l, the broker computes the nominal energy depletion rate $\rho = E/t_l$. When allocating energy tickets, the broker determines the current energy availability $e(t)$ using Pixie's energy meter, and compares it to the scheduled energy availability $\hat{e}(t) = E - \rho \cdot t$. The *energy deficit* is computed as $\Delta = e(t) - \hat{e}(t)$.

We have implemented several policies for energy scheduling. The *conservative* policy will only allocate tickets as long as $\Delta > 0$, which adheres strictly to the energy schedule. In some cases, it may be desirable to permit energy consumption above the limit, as long as the application recoups the deficit at a later time. (As an example, consider transmitting a series of radio messages as soon as an interesting event is detected, which might necessitate sleeping for a longer interval to adhere to the lifetime target.) The *credit-based* policy allows the application to accumulate an energy debt, up to a configurable amount α. As long as $\Delta > -\alpha$, the broker will allocate new energy tickets. To ensure debt payback, once the deficit reaches this threshold the broker stops allocating tickets until $\Delta > 0$. This causes the application to adhere to the energy schedule over long periods, but allows short bursts of energy usage that would otherwise violate the schedule.

Energy-aware switch: The energy switch broker accepts data on its input port and directs it to one of several output ports, based on energy availability. Multiple application stages register with the switch, providing information on their energy quanta requirements and priority. The switch receives energy tickets from the default energy broker at some rate based on the lifetime target. The switch directs data to the stage with the highest priority that does not exceed energy availability. This is an example of *broker layering*, allowing the energy scheduling policy to be decoupled from that of the switch. This is similar to Eon's *energy state-based paths* [48], although Pixie's resource brokers permit a greater degree of flexibility.

Energy-aware filter: Similar to the energy-aware switch, this broker selectively drops input data to meet the lifetime target, again using the default energy broker to receive tickets according to the lifetime target. When dropping input data, downstream stages receive no input and go idle; in effect, the energy-aware filter controls the scheduling of downstream stages. By placing an energy-aware filter near the root of the dataflow graph, the entire application can be duty-cycled according to the energy schedule.

Bandwidth broker: Pixie's bandwidth broker uses information on current bandwidth availability to issue bandwidth tickets to applications according to a priority-driven policy. Application stages register their bandwidth quanta requirements (expressed as the number of packets to be transmitted over an allocation window) and priority with the broker. Each allocation window, the broker orders the requested bandwidth quanta by decreasing priority and allocates tickets to application stages until the ticket values exceed the current bandwidth estimate.

Given that bandwidth may fluctuate over short time intervals, the allocation window must be chosen carefully to avoid the need for ticket revocation. At the same time, using a short allocation window imposes high overhead and may not match the application's internal bandwidth requirements (which might vary over longer time scales). The broker matches its allocation epoch to the underlying bandwidth allocator's estimation window, set to 5 sec by default.

Combined bandwidth/energy broker: Pixie's broker model allows composition of resource management policies across distinct physical resources. As an example, the bandwidth/energy broker performs joint allocations of bandwidth and energy for downstream stages. Each client stage of the broker registers a request for a resource quantum of b units of bandwidth and e units of energy, again with a corresponding priority. Note that allocating bandwidth always implicitly allocates energy for the corresponding transmissions; the additional energy allocation e might represent computational work required to generate the data to be transmitted. The broker allocates resources using a priority-first bin-packing solution based on both current bandwidth and energy availability.

3.5 Discussion

Two issues arise when considering the potential space of resource brokers and corresponding policies. The first is whether tickets and brokers place an undue burden on application code to reason about resource management under varying conditions. While Pixie's ticket abstraction permits a wide range of policies to be implemented, it is also true that many applications do not need to be concerned with this level of detail. One approach for these applications is to treat allocations as a coarse-grained indication of the "fidelity level" (e.g., *low*, *medium*, or *high*) at which they should operate, similar to the interface provided by Levels [31]. Pixie is intended to support both fine-grained and coarse-grained adaptivity through a single programming model.

The second issue concerns the generality of the broker model, in terms of the set of policies that can be supported by interposing broker stages within an application graph. Brokers appear to be very general in this regard, and can affect cross-layer adaptations using control interfaces into stages in different portions of the stage graph. By the same token, multiple brokers might "compete" for the same underlying resource. Our assumption is that brokers explicitly coordinate (for example, by performing proportional-share allocations) in this event. Our experiments to date have not encountered a need for multiple brokers for a given resource. While a complete exploration of resource management policies is beyond the scope of this paper, we feel that Pixie's resource ticket model makes it possible to express a range of policies within a unifying architecture.

Finally, brokers must be aware of the resource demands of application stages. We assume that applications correctly (and perhaps conservatively) report their resource demand to brokers. This requires either online measurement or offline profiling, and Pixie is agnostic as to which method is used.

```
/* Simple stage implementation */
generic module TestStageP() {
  provides interface PixieStage;
  uses interface PixieSink as Output;
  uses interface PixieResource as EnergyAlloc;
  uses interface PixieMemAlloc;
} implementation {
  ticket_t curTicket; /* Energy ticket */

  command error_t PixieStage.init() { /* ... */ }

  command error_t PixieStage.run(memref_t ref) {

    /* Process incoming data if enough energy */
    if (tktAmount(curTicket) >= ENERGY_NEEDED) {
      /* Redeem energy ticket */
      call EnergyAlloc.redeem(curTicket);
      /* Process data */
      processData(ref);

    } else {
      /* Request more energy */
      call EnergyAlloc.request(ENERGY_REQUEST,
                               60*1000);
    }

    /* Send ref downstream */
    call Output.emit(ref);
    /* Ref no longer needed by this stage */
    call PixieMemAlloc.release(ref);
  }

  event void EnergyAlloc.ticketGranted(
      ticket_t ticket) { curTicket = ticket; }
}
```

Figure 2: **Implementing a Pixie stage.**

4. IMPLEMENTATION

Pixie is implemented in NesC [17], allowing us to directly link against the wide range of hardware drivers and libraries implemented for TinyOS [24]. Since Pixie provides its own abstractions for concurrency, memory management, and dataflow, "legacy" TinyOS code must be wrapped in a Pixie stage before use. Pixie does not guarantee backwards-compatibility with all TinyOS code, since such code might conflict with Pixie's scheduling and resource management mechanisms.

As shown in Figure 2, a Pixie stage is implemented as a NesC component that provides the `PixieStage` interface. For each input to the stage, the `PixieStage.run()` command is called with the corresponding memref as an argument. A stage may have zero, one or more output ports, provided by the `PixieSink` interface; a stage pushes data to an output port by calling `PixieSink.emit()`. Stages may provide additional, customized control interfaces to coordinate their operation with other stages or with resource brokers. Stages use the `PixieResource` interface for requesting and manipulating resource tickets.

Figure 3 shows the wiring for a complete Pixie application, consisting of stages for sampling, filtering sample values, storing data to flash, performing some processing on the data, and transmitting data to the base station. Pixie provides a standard set of stages for radio communication, timers, flash I/O, LEDs, and sampling. The `PixieEnergySwitch` stage implements logic for selecting which of the two processing stages should receive input data, based on available energy. The broker and the processing stages use separate *control wiring* to coordinate their operation. Likewise, the broker uses control wiring to `PixieCore`, which provides the low-level energy allocator. As shown in the figure, Pixie application wirings tend to be very succinct.

The core Pixie implementation (including interfaces, header files, and core system components) is 8755 lines of NesC code, including comments. Most components are very short, less than 200 lines of NesC code apiece. Section 6 presents benchmark results measuring Pixie's computational and memory overhead.

```
configuration MyApp {
} implementation {
  components
    PixieCore,
    new PixieSamplingStage(RATE, CHANNELS),
    new SampleFilterStage() as Filter;
    new PixieStorageStage() as Storage;
    new PixieEnergySwitch() as Broker,
    new ProcessingStageA() as ProcA,
    new ProcessingStageB() as ProcB,
    new PixieSendStage(BASE_STATION_ID);

  /* Dataflow graph wiring */
  PixieSamplingStage.Output -> Filter.Input;
  Filter.Output -> ProcA.Input;
  Filter.Output -> ProcB.Input;
  Filter.Output -> Storage.Input;
  ProcA.Output -> SendStage.Input;
  ProcB.Output -> SendStage.Input;

  /* Control wiring between broker and stages */
  Broker.EnergyControl -> ProcA;
  Broker.EnergyControl -> ProcB;
  /* Interface to energy allocator */
  Broker.EnergyAlloc -> PixieCore;
}
```

Figure 3: **Example Pixie application wiring.**

5. APPLICATION EXAMPLES

In this section, we detail two applications that we have developed using the Pixie OS: a system for limb motion analysis using a network of wearable sensors, and an acoustic target detection system. These applications represent a broad class of data-intensive sensor networks. Other applications that fall into this space include structural monitoring [7, 8, 42], seismic monitoring of earthquakes and volcanoes [25, 53], and acoustic animal habitat monitoring [1, 35].

5.1 Limb motion analysis

The first application we consider involves the use of wearable sensors for high-resolution monitoring of limb movements. This application is intended to improve the evaluation and care of patients being treated for Parkinson's Disease, stroke, epilepsy, and other diseases that affect an individual's motor ability. Our group is working closely with physicians in rehabilitation medicine to develop this application for various studies; an earlier version of the system is currently being used to measure the effectiveness of deep-brain stimulation in Parkinson's patients. The goal of the system is to capture detailed traces of limb movements to identify periods of dyskinesia and bradykinesia, which are tremors and sluggish movements associated with the disease, and are affected by the timing and dosage of the medication taken by the patient.

Current approaches to this problem include direct observation by a physician, (who scores the patient's motor function subjectively), as well as bulky dataloggers attached to wired sensors on the patient's limbs. These approaches are only viable in a laboratory setting, limiting the duration and realism of the collected data. Wearable, wireless sensors have the tremendous advantage that they can be worn continuously by a patient over the course of several weeks of a study, except while sleeping and bathing. Nodes can be recharged each night, allowing battery size to be minimized.

In our system, a patient wears a set of up to 10 SHIMMER [26]

nodes with triaxial accelerometer and gyroscope sensors. One node is worn on each of the limb segments (upper and lower arm and leg), one on the waist, and another on the torso. The SHIMMER platform is based on the TI MSP430 microcontroller and CC2420 802.15.4 radio, and includes 2 GB of MicroSD flash for data logging. The node uses a slim rechargeable battery and measures just $54 \times 36 \times 18$ mm and weighing 24 g, including the case. A laptop located in the patient's home acts as a base station, collecting data from the body sensor network and relaying it to the physician's office using a dialup or broadband Internet connection.

From a high level, the system collects six channels of data, sampled at 100 Hz per channel, from each body sensor node. Raw signals are then passed through a series of *feature extractors*, which derive features such as peak amplitude, RMS, dominant frequency, jerk (derivative of acceleration), periodicity, and approximate entropy over a series of overlapping time windows from each signal. Features are then passed to a *classifier* that clusters the feature values to determine if the patient is experiencing *normal movement*, *dyskinetic*, or *bradykinetic* motor fluctuations.

Given the limited radio bandwidth of low-power sensor nodes, especially with up to 10 nodes in close proximity, it is not practical to transmit complete raw signals from each node in real time. In addition, the radio link capacity may fluctuate greatly, due to the patient's movement within the home, and may experience periods of disconnection when the patient leaves the home or is otherwise out of range of the base station. Finally, the nodes lack the computational power to perform complete signal classification locally.

Our design strikes a balance between computation and communication load by performing feature extractions on the sensor node, and opportunistically using available radio bandwidth to send the *highest priority* data to the base station (either raw signal data or features computed from the raw signal), subject to bandwidth limitations. We assume that each feature type F_i has an associated size $S(F_i)$, in terms of the number of radio packets required to transmit one window of the feature, and a priority $P(F_i)$, assigned by the end user. For example, the priority for raw signals is higher than that for dominant frequency features, which is in turn higher than RMS and peak amplitude features. On each epoch, the system determines the set of features to transmit using a priority-first allocation, subject to the current estimate of the channel capacity to the base station.

This application maps naturally onto Pixie's dataflow model, as shown in Figure 1. Raw samples are passed through a *stillness filter*, which drops the data when the sensor node does not appear to be moving. Data is then passed to a series of feature extraction stages, which compute features across multiple overlapping windows of the input signal. Resource management is delegated to the bandwidth broker, described in Section 3.4, which assigns a fraction of the radio bandwidth to each of the feature generation stages according to the current channel capacity.

In our current prototype, sensor nodes do not buffer features for later transmission; our goal is to minimize the latency between signal collection and feature delivery at the base station. This implies that during periods of disconnection, features are not computed, thereby saving energy. A natural extension would involve buffering data in flash and transmitting previously-computed features when connectivity is reestablished. Also, all data must be sent directly to the base station, rather than to a body-mounted gateway that could collect and buffer data locally. Due to space limitations, we do not consider these alternatives in this paper.

The entire motion analysis application consists of five application-specific stages, with a total of 1486 lines of NesC code. The main application wiring is just 82 lines.

Detector	*Energy cost*	*Accuracy*	*False positive rate*
THRESH	2.71 mJ	70.5%	25.0%
HIGHPASS	13.95 mJ	84.0%	16.0%
FFT	49.12 mJ	100.0%	0.0%

Figure 4: **Summary of detection algorithms used by the acoustic target detection system.** *Accuracy is defined as the ratio of the sum of true positives and true negatives to the total number of sample windows.*

5.2 Acoustic target detection

The second application involves the use of a network of microphones to perform acoustic target detection, such as detecting animal movements, gunfire, or vehicles passing through the sensor field. Other systems that involve acoustic target detection include EnviroMic [35], ENSbox [1], and PinPtr [47]. Our goal is not to demonstrate new techniques for acoustic signal processing, but rather to highlight the features of Pixie that enable resource adaptivity in this domain.

Our focus here is on the *detection* of acoustic signals of interest, rather than on ranging and tracking of their source. Target detection requires sampling at high data rates (several kHz) and significant computation to filter out noise and detect the target's acoustic signature. In addition, the sensor network's operation is highly energy-constrained, given the need for a long lifetime. This application is designed to run on the iMote2, which features a Marvell PXA271 XScale processor, capable of running at frequencies between 13 and 416 MHz, with 256 KB of on-chip memory.[1]

The Pixie-based acoustic detection application strives to meet a target lifetime for each node while maximizing the accuracy of detection and minimizing false positives. Each sensor node samples acoustic data at 24 kHz and passes the raw samples grouped in 512 millisecond windows to a *quietness filter*, which drops windows that do not appear to contain a significant acoustic waveform.

Windows that pass the filter are passed to an *energy-aware switch*, as described in Section 3.4. The switch enables one of several *detector stages* based on current energy conditions. Each detector d consumes some amount of energy e_d to process an input signal, with a corresponding detection accuracy a_d and false positive rate f_d. When a detector "fires," it sends a radio message to the base station indicating the node ID and detection time. Note that both true and false detections cause radio messages to be sent.

By using a range of detectors, with varying energy cost and accuracy, the energy broker can tune the fidelity of the system subject to fluctuations in load (arising due to the quietness filter). The detection algorithms that we have developed are summarized in Figure 4. The simplest and cheapest detector just calculates the mean amplitude of the acoustic signal over each of a sequence of subwindows. If the mean amplitude of a subwindow exceeds a set threshold, the detector fires. Note that this detector cannot discriminate between noise and the target, so its false positive rate is expected to be high. A more sophisticated detector first applies an FIR highpass filter to remove some of the low-frequency noise, with the assumption that the target's acoustic signature falls in a higher frequency range. This may not be effective in the face of broadband noise sources. The most expensive detector performs a 512-point FFT, matching the spectral content of the signal against a known signature of the target (similar to the marmot call detector described by Girod *et al.* [18]).

As expected, the more sophisticated detectors require a greater amount of energy, due to their higher computational overhead. More-

[1]The iMote2 supports an additional 32 MB of external DRAM; however, in this application we disable the DRAM due to its prohibitively high power consumption.

over, the FIR detector and FFT detector requires that the CPU frequency be set to 104 MHz to keep up with the incoming sample rate. The FFT stage requests the frequency switch from Pixie when it begins processing a window of samples, and returns the CPU to the default frequency of 13 MHz when the computation is complete. For comparison, the threshold detector requires 2.71 mJ of energy for each window it processes, while the FFT detector requires over 49.12 mJ. This application consists of 5 application-specific stages, with a total of 1791 lines of NesC code. The application wiring is 62 lines of code.

6. EVALUATION

In this section, we present a detailed evaluation of Pixie along several axes. First, we evaluate Pixie's overhead, in terms of CPU time and memory footprint, through microbenchmarks. Second, we evaluate the accuracy of Pixie's runtime bandwidth and energy estimation techniques. Third, we explore the effectiveness of bandwidth adaptivity in the motion analysis system, and the use of energy adaptivity in the acoustic tracking application, while varying the target lifetime and energy scheduling policy. Finally, we demonstrate the ease of altering resource management policies through a broker that performs combined bandwidth and energy adaptivity.

6.1 Microbenchmarks

To measure the CPU overhead of Pixie's dataflow abstractions, we wrote a simple microbenchmark that pushes a memref through a linear chain of 10 stages that perform no computation. We measure the time to traverse the chain, comparing it to the time taken by a similar TinyOS application using a series of cross-module command invocations. Running on a iMote2 sensor node, Pixie requires 0.009 ms per stage traversal, compared to 0.003 ms per component traversal in TinyOS. The additional overhead is largely due to memref marshaling, requiring that the reference count be incremented and decremented at each stage boundary.

To estimate memory footprint, we wrote a benchmark consisting of a timer that causes the LEDs to blink and sends a radio message, analogous to the TinyOS *RadioCountToLeds* application. The Pixie application consumes 11752 bytes of program memory and 6256 bytes of RAM. Of this, 214 bytes of ROM and 125 bytes of RAM are consumed by the application stages. The Pixie scheduler and memory manager use 958 (5825) bytes of ROM (RAM); the bulk of this is the heap, statically allocated at compile time. In comparison, RadioCountToLeds uses 10570 bytes of ROM and 271 bytes of RAM. The additional memory overhead for Pixie is largely due to the heap.

6.2 Resource estimation accuracy

An important concern for enabling resource adaptivity is how well Pixie's low-level resource estimators can track varying conditions. To evaluate the bandwidth estimator, we introduced artificial packet delays into Pixie's link layer, allowing us to control the per-packet delay across a range of values. Pixie's bandwidth estimator (which is unaware of the artificial delays) must therefore track the change in the delay. A simple application sends packets at a rate just below that reported by the estimator. The results are shown in Figure 5. The top portion of the figure shows the artificial packet rate and the estimated rate, which match closely. The lower portion of the figure shows the estimation error, which is nearly zero except for transitions between rates; this is not surprising since the estimator computes the measured delay over a measurement window of 1 second. In all cases the bandwidth estimator rapidly converges to the new rate.

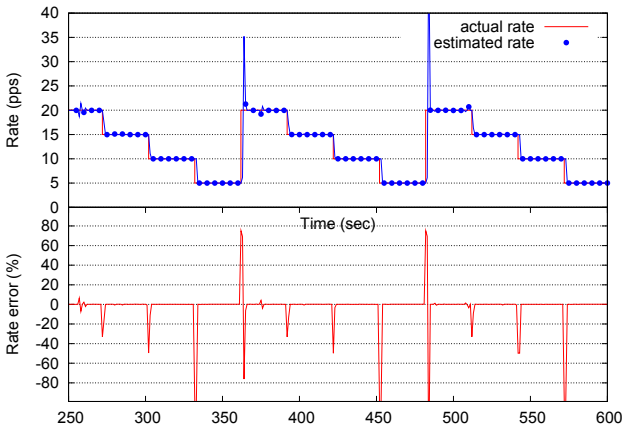

Figure 5: **Bandwidth estimation accuracy.** *In this experiment, the link layer is modified to artificially delay packets across a range of delay settings. Pixie's bandwidth estimator rapidly determines the new rate, with errors occurring on each rate transition.*

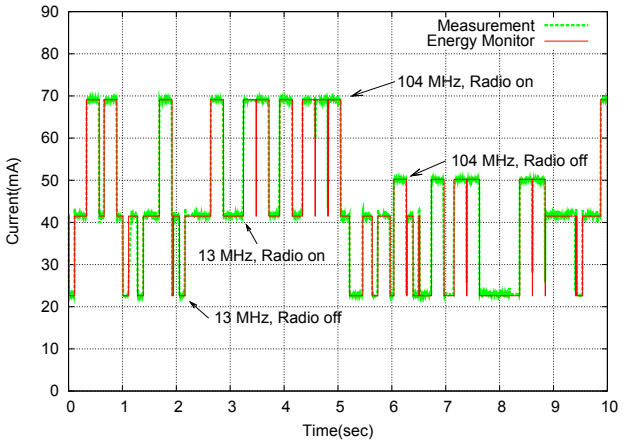

Figure 6: **Accuracy of Pixie's energy metering component.**

To evaluate Pixie's energy meter, we wrote an application for the iMote2 that performs a series of operations with varying energy drain, including radio listening, radio transmission, radio idle, and computing a 512-point FFT at 13 MHz and 104 MHz. The order and duration of each operation are randomized. We instrumented the node using a Keithley 2701 digital multimeter and compared its output to that of Pixie's energy metering component. A representative run is shown in Figure 6; the estimated energy is nearly identical to the actual energy usage. We ran the experiment for a total of 3 hours and measured the total energy consumption; Pixie's energy meter underestimated the true energy usage by 2.5%, and the error was nearly constant during the experiment. We believe our estimate can be improved with more detailed modeling.

6.3 Bandwidth adaptivity

Next, we look at the effectiveness of bandwidth adaptivity in the motion analysis system described in Section 5.1. For this experiment, we use a simplified version of the application that computes three types of features from the raw sensor data: *RMS*, *peak amplitude* (PA), and *autocorrelation* (AC). Each feature type, including raw samples, has an associated bandwidth requirement and application-specific value, shown in Figure 7. Time is divided into 1 second epochs. For each epoch, the bandwidth broker allocates

Feature type	Bandwidth	Value
Raw samples	20 pps	100
RMS	6 pps	20
Peak amplitude	4 pps	10
Autocorrelation	2 pps	5

Figure 7: **Feature types, bandwidth requirements, and application-assigned values used in the motion-analysis system.**

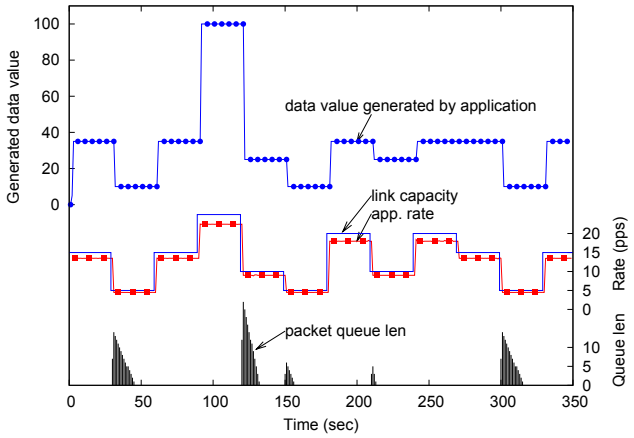

Figure 8: **Bandwidth adaptivity in the motion-analysis application.** *This graph shows the total value of data generated by the application (top) and the estimated link capacity, application packet rate, and packet queue length (bottom). Data is shown for one node in a four-node network. The radio link capacity is artificially varied using a random pattern.*

bandwidth tickets either for transmitting raw samples, or to the feature-extraction stages, according to the current link capacity estimate. Recall that multiple nodes are worn by the patient and are each competing for the radio channel in an uncoordinated fashion.

Varying link capacity: For the experiment, we use a network of 4 nodes (placed near each other) transmitting to a nearby base station, artificially varying the packet delay experienced by the link layer on each node; this ensures repeatability and avoids artifacts due to mobility and path loss. Figure 8 shows a trace from a single node as bandwidth is varied. The upper portion of the figure shows the total value in terms of features and/or raw samples generated by the application, for each 1 second epoch. The lower portion of the figure shows the corresponding bandwidth estimate (which is varied randomly), the achieved application data rate, and the transmission packet queue length. The bandwidth broker caps the generated packet rate just below the estimated link capacity, to avoid overflowing packet buffers. Although it is difficult to tell from the figure, there is a delay of 1 second between changes in the bandwidth and the corresponding response by the application. This causes the packet queue to grow when the link capacity drops, as expected.

Impact of RF interference: To evaluate adaptivity under more realistic link capacity variation, we set up a network of four nodes, with a separate node generating varying amounts of cross-traffic to induce radio interference. Figure 9 shows a trace of the behavior for a single node in the network as the cross-traffic rate increases. Pixie's bandwidth broker tunes the type of data generated by the application in response to the varying link capacity; the packet queue only begins to fill when the cross-traffic nears the channel capacity of approximately 100 packets/sec.

6.4 Energy adaptivity

Next, we look at the use of energy adaptivity in the context of the acoustic target detection application, running on an iMote2 node

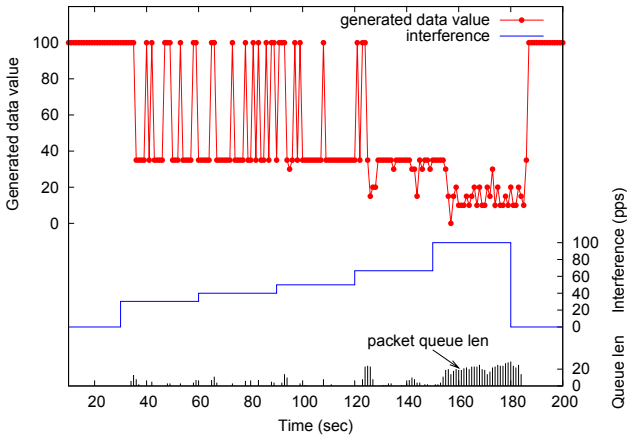

Figure 9: **Adaptivity under radio interference.** *This figure shows the behavior of one node (in a 4-node network) experiencing varying amounts of radio interference. The generated data value (top) and estimated link capacity (bottom) are shown. Pixie's bandwidth estimator tracks the estimated link capacity, while the bandwidth broker adapts the type of data sent in each epoch accordingly.*

described in Section 5.2. For these experiments, we generate a synthetic acoustic signal consisting of intermittent marmot calls, interspersed with varying background noise consisting of bird chirps and wind. These signals are based on an acoustic dataset of marmot chirps collected by the VoxNet system [2].

We observe the behavior of the node as it detects the marmot calls and adapts to energy availability by selecting different detection algorithms. While the FFT detector has high accuracy, it consumes considerably more energy than the simpler detection algorithms. Background noise tends to cause the less accurate detectors to report false positives, which wastes energy by transmitting spurious detection messages. We vary two parameters: the energy scheduling policy used by Pixie's energy broker and the node's lifetime target. We assume that nodes are equipped with high-capacity 35 Ah Tadiran D-cell batteries.

Different energy scheduling policies: Figure 10 shows the results with a target lifetime of 40 days, with three different energy scheduling policies provided by Pixie's energy broker. The *optimistic* strategy ignores energy limits and always runs the *FFT* detection algorithm, which has perfect marmot detection accuracy and no false positives. This strategy would achieve a lifetime of 29 days, well below the target. The *conservative* and *credit-based* strategies are described in Section 3.4. As the figure shows, the conservative strategy always adheres to the energy schedule, while the credit-based scheme occasionally uses more energy than the schedule would normally allow. However, it is forced to use the less-expensive *HIGHPASS* and *THRESH* detection algorithms, incurring a higher rate of false positives.

The credit-based scheme opts to incur "energy debt" when there has been a marmot detection in the previous time window (e.g., from 0-100 seconds in the figure). When no detections have occurred for at least 10 seconds, the policy enters "debt payback" by disabling detections altogether (e.g., from 100-150 seconds). This allows the node to recoup energy but may lead to missed marmot detections.

Figure 11 summarizes the accuracy and false positive rate for each policy over a one-hour run. The credit-based scheme is able to achieve higher accuracy and lower false positives than the conservative policy.

Different lifetime targets: Next, we explore the effect of vary-

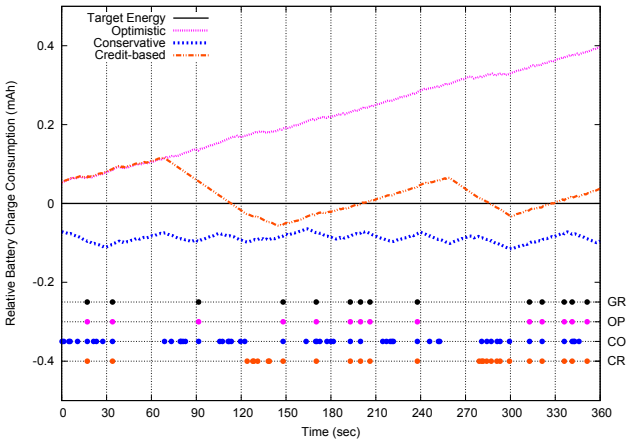

Figure 10: **Energy adaptivity with different energy scheduling policies.** *This figure shows energy consumption and marmot detections under three energy scheduling policies:* optimistic *(OP),* conservative *(CO), and* credit-based *(CR). The top portion of the figure shows the energy consumed by the node relative to a target node lifetime of 40 days. The bottom portion shows the number of marmot detections under each policy, compared to ground truth (GR) of true marmot calls in the acoustic trace.*

Policy	Detection accuracy	False positives
Optimistic	100%	0%
Credit-based	95.61%	3.99%
Conservative	92.19%	7.66%

Figure 11: **Detection accuracy and false positive rate for each energy scheduling policy.**

ing the lifetime target, using only the conservative energy-scheduling policy. Figure 12 shows results for lifetime targets of 30, 40, and 50 days. As expected, with a longer lifetime target, the application is forced to run lower-quality detection algorithms, which impacts both detection accuracy and the false positive rate. For a target of 30 days, the system achieves 99.9% accuracy and 0.1% false positives; for 50 days, the accuracy drops to 78.3% and false positives increase to 20.4%.

Impact of unexpected energy drain: The previous experiments assumed that the sensor node was capable of predicting its energy requirements accurately, based on knowledge of the energy cost for running the detection algorithms and transmitting detection messages. Next, we look at the impact of unexpected energy drain, in this case caused by the node forwarding radio packets along a multihop routing path. We emulate this condition using separate node that relays packets through the node under test, using a bursty traffic pattern generating up to 10 packets/sec.

Figure 13 shows the results for a lifetime of 40 days with conservative energy broker. As expected, packet forwarding causes the node to have less available energy for target detections, and its detection performance suffers accordingly. In this case, accuracy drops to 86.2% with a false positive rate of 13.6%. Note, however, that the node continues to meet its energy schedule, since Pixie's software energy meter accounts for the additional drain induced by packet forwarding and the acoustic detection logic adapts accordingly.

6.5 Combined energy and bandwidth adaptivity

As a final demonstration of Pixie's ability to facilitate adaptive applications, we consider augmenting the motion-analysis system with both energy *and* bandwidth awareness. Given the small (250

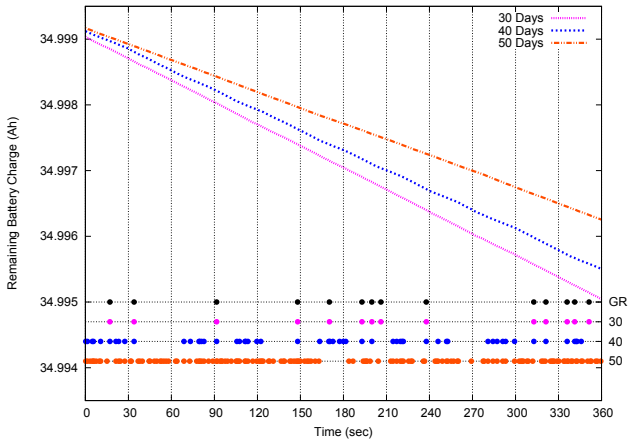

Figure 12: **Lifetime target.** *Conservative broker behavior under different lifetime targets.*

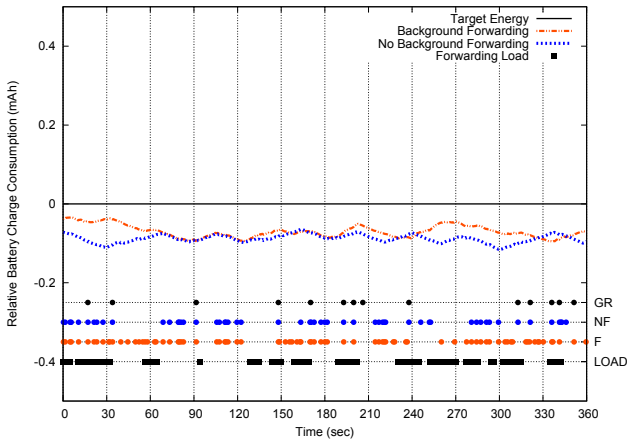

Figure 13: **Effect of multihop packet forwarding.** *This figure shows the behavior of the conservative energy broker under a bursty background packet forwarding load. The upper portion of the figure shows the energy availability using a target lifetime of 40 days. The lower portion of the figure shows the background traffic pattern (LOAD), ground-truth marmot calls (GR), and the marmot detections both with (F) and without (NF) the background forwarding load.*

mAh) battery size used by the wearable sensors, to achieve long target lifetimes, it may be necessary for nodes to limit their radio transmissions over time. Introducing this behavior into the application was easy: we created a new broker that bounds the maximum transmission rate by the link capacity or the energy availability, whichever is smaller. Figure 14 shows the behavior of a node adapting to bandwidth availability (artificially limited in a manner identical to Figure 8 with target lifetimes of 24, 36, and 48 hours. The longer the target lifetime, the lower the rate at which nodes transmit data, which leads to graceful degradation in the value of the data (raw samples or features) generated by the node.

7. RELATED WORK

Pixie is closely related to a range of primitives for resource management in sensor networks; programming models facilitating adaptive application design; and resource adaptivity for mobile and pervasive computing systems. We describe each in turn below.

Resource management primitives: Operating systems such as TinyOS [24], SOS [22], and Contiki [11] provide low-level inter-

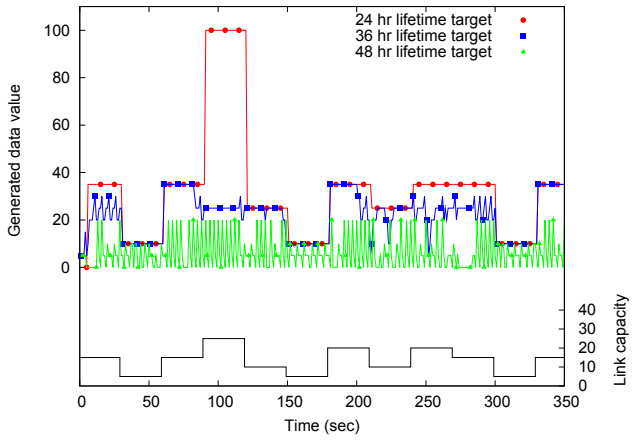

Figure 14: **Combined energy and bandwidth adaptivity in the motion analysis application.** *A combined energy/bandwidth broker is used to bound radio transmissions to meet a lifetime target of 24, 36, and 48 hours. As the figure shows, longer lifetimes cause nodes to gracefully degrade in terms of the type of data they produce.*

faces for managing hardware state but offer no guidance for using these interfaces to achieve a given resource-management goal. Low-power listening MACs, such as B-MAC [44], reduce energy usage for idle listening, while ICEM [29] and SNACK [20] automate some aspects of energy savings by coordinating access to hardware resources across multiple components and concurrent tasks. Adaptive duty-cycling algorithms, such as those proposed by Vigorito *et al.* [51] and Kansal *et al.* [28], tune the duty cycle of a sensor network application according to energy availability.

Pixie relies on estimation of available energy and radio bandwidth for driving resource allocations. Most approaches to energy metering have relied on hardware support, including iCount [13], Eon [48], Triage [4], and PowerScope [15]. Pixie opts for a model that tracks hardware states in software, similar to the approaches used in Contiki [12], ECOsystem [56], and the PowerTOSSIM simulator [46]. BodyQoS [57] proposes a communication layer for body sensor networks that provides quality-of-service guarantees using link estimation and admission control; this approach could be implemented as a broker atop Pixie's bandwidth allocation layer.

Programming models: Pixie is related to several systems that facilitate resource-aware application design. TinyDB's *lifetime-based queries* [36] target a given lifetime by (statically) setting the query duty cycle. Triage [4] allows application logic to be partitioned across coupled hardware platforms based on energy availability.

Eon [48] and Levels [31] provide programming models for adapting to energy availability. Eon provides a dataflow model similar to Pixie and automatically tunes timer rates and dataflow paths based on energy availability; application code is not involved in resource management decisions at runtime. In contrast, Levels allows application components to define multiple fidelity levels, which are configured in response to energy availability. Pixie and Levels are implemented in NesC, whereas Eon requires using a new configuration language. Both Levels and Eon are limited to energy management, and it is unclear whether these systems could be generalized to managing other resources. Both systems strongly couple the resource management policies to the underlying mechanisms. We note that the policies used by Eon and Levels could easily be implemented as Pixie resource brokers.

Nano-RK [14] provides real-time guarantees through *static* resource reservations based on offline estimates of CPU time, packet rates, and sampling intervals used by an application. However, this approach fails to address dynamically-varying load or fluctuations in resource availability that arise at runtime.

Pixie's resource tickets and broker models are strongly influenced by resource containers [5], as well as related concepts in Odyssey [41], ECOsystem [56], and Rialto [27]. Pixie's dataflow programming model is inspired by similar approaches in Vango [21], Click [30], WaveScope [40], and Flask [37], although none of these systems directly address resource awareness and adaptivity.

Mobile systems: Outside of the sensor network domain, much work has focused on resource awareness in mobile and pervasive computing systems. Odyssey [41] is a framework for adaptive mobile applications that permits applications to adapt to changing network bandwidth [41], energy [15, 33], and computational load [39]. ECOsystem [56] does not require application code changes, instead tuning OS scheduling parameters automatically based on energy availability. Puppeteer [32] takes a similar approach, adapting to bandwidth variation by interposing on the software component layer in Windows NT DCOM. These systems differ substantially from Pixie in terms of application demands, hardware platforms, and the need to support legacy operating systems. Indeed, we argue that sensor networks are inherently better suited to resource adaptivity, given that many applications can naturally tolerate variations in sampling, communication, or processing rates.

8. CONCLUSIONS AND FUTURE WORK

Pixie represents a new approach to resource management in sensor networks, providing applications with fine-grained visibility and control over resource allocation through tickets, while permitting high-level, reusable resource management policies via brokers. We have demonstrated Pixie's effectiveness at managing bandwidth and energy constraints in two data-intensive applications. Pixie incorporates runtime estimation of memory, storage, bandwidth, and energy availability, and a toolkit of brokers for mediating between application needs and low-level resource allocations.

To date, Pixie focuses on resource management of an individual sensor node. Our next steps involve *coordinated* resource management of sensor nodes across the network. A range of algorithms and protocols have been proposed for congestion control, network topology adaptation, coordinated duty-cycling, and other behaviors. Our goal is to allow such policies to be expressed through a unified programming model that can permit domain scientists and other non-expert users to build adaptive, efficient, and self-tuning sensor networks by reasoning about resource management as a core aspect of the programming abstraction.

Acknowledgments

The authors wish to thank Paolo Bonato and Shyamal Patel of the Spaulding Rehabilitation Hospital for their assistance with the motion-analysis application. We also thank Lewis Girod for providing acoustic dataset and offering advice on marmot detection DSP algorithms. Finally, we would like to thank our shepherd, Mark Corner, for his guidance in preparing the final version of the paper. This work was supported by the National Science Foundation (grant numbers CNS-0546338 and CNS-0519675), Microsoft, Sun Microsystems, Siemens, CIMIT, and ArsLogica SpA. We are extremely grateful to all of our sponsors for their support of this research.

9. REFERENCES

[1] A. M. Ali, K. Yao, T. C. Collier, C. E. Taylor, D. T. Blumstein, and L. Girod. An empirical study of collaborative

acoustic source localization. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, Cambridge, MA, 2007.

[2] M. Allen, L. Girod, R. Newton, S. Madden, D. T. Blumstein, and D. Estrin. Voxnet: An interactive, rapid-deployable acoustic monitoring platform. In *IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks*, St. Louis, Missouri, 2008.

[3] N. Banerjee, M. D. Corner, and B. N. Levine. An energy-efficient architecture for dtn throwboxes. In *Proc. IEEE INFOCOM*, May 2007.

[4] N. Banerjee, J. Sorber, M. Corner, S. Rollins, and D. Ganesan. Triage: Balancing Energy Consumption and Quality of Service in Tiered Microservers. June 2007.

[5] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. the Third OSDI (OSDI '99)*, February 1999.

[6] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. *ACM/Kluwer Mobile Networks and Applications (MONET)*, 10(4):563–579, August 2005.

[7] K. Chebrolu, B. Raman, N. Mishra, P. K. Valiveti, and R. Kumar. BriMon: A Sensor Network System for Railway Bridge Monitoring. In *Proc. Sixth International Conference on Mobile Systems, Applications, and Services (MobiSys)*, Breckenridge, CO, June 2008.

[8] K. Chintalapudi, J. Paek, O. Gnawali, T. Fu, K. Dantu, J. Caffrey, R. Govindan, and E. Johnson. Structural Damage Detection and Localization Using NetSHM. In *Proc. Fifth International Conference on Information Processing in Sensor Networks: Special track on Sensor Platform Tools and Design Methods for Networked Embedded Systems (IPSN/SPOTS'06)*, April 2006.

[9] T. Choudhury, G. Borriello, S. Consolvo, D. Haehnel, B. Harrison, B. Hemingway, J. Hightower, P. Klasnja, K. Koscher, A. LaMarca, J. A. Landay, L. LeGrand, J. Lester, A. Rahimi, A. Rea, and D. Wyatt. The mobile sensing platform: An embedded system for capturing and recognizing activities. *IEEE Pervasive Magazine*, April 2008.

[10] D. S. J. De Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. In *Proceedings of the 9th ACM International Conference on Mobile Computing and Networking (MobiCom '03)*, San Diego, California, September 2003.

[11] A. Dunkels, B. Gronvall, and T. Voigt. Contiki: A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proc. First IEEE Workshop on Embedded Networked Sensors (EmNetS)*, Tampa, FL, November 2004.

[12] A. Dunkels, F. Osterlind, N. Tsiftes, and Z. He. Software-based on-line energy estimation for sensor nodes. In *Proc. Fourth Workshop on Embedded Networked Sensors (EmNets 2007)*, June 2007.

[13] P. Dutta, M. Feldmeier, J. Paradiso, and D. Culler. Energy metering for free: Augmenting switching regulators for real-time monitoring. In *Proc. Seventh International Conference on Information Processing in Sensor Networks (IPSN'08)*, April 2008.

[14] A. Eswaran, A. Rowe, and R. Rajkumar. Nano-rk: An energy-aware resource-centric operating system for sensor networks. In *Proc. IEEE Real-Time Systems Symposium*, December 2005.

[15] J. Flinn and M. Satyanarayanan. Managing battery lifetime with energy-aware adaptation. *ACM Transactions on Computer Systems (TOCS)*, 22(2), May 2004.

[16] R. Ganti, P. Jayachandran, T. Abdelzaher, and J. Stankovic. SATIRE: A Software Architecture for Smart AtTIRE. In *Proc. ACM Mobisys*, Uppsala, Sweden, June 2006.

[17] 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 *Proc. Programming Language Design and Implementation (PLDI)*, June 2003.

[18] L. Girod, M. Lukac, V. Trifa, and D. Estrin. The design and implementation of a self-calibrating distributed acoustic sensing platform. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 71–84, Boulder, CO, 2006.

[19] O. Gnawali, B. Greenstein, K.-Y. Jang, A. Joki, J. Paek, M. Vieira, D. Estrin, R. Govindan, and E. Kohler. The TENET Architecture for Tiered Sensor Networks. In *Proc. ACM Conference on Embedded Networked Sensor Systems (Sensys)*, Boulder, CO, November 2006.

[20] B. Greenstein, E. Kohler, and D. Estrin. A sensor network application construction kit (snack). In *Proc. ACM SenSys*, November 2004.

[21] B. Greenstein, C. Mar, A. Pesterev, S. Farshchi, E. Kohler, J. Judy, and D. Estrin. Capturing high-frequency phenomena using a bandwidth-limited sensor network. In *Proc. Sensys 2006*, Boulder, CO, November 2006.

[22] C.-C. Han, R. K. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava. SOS: A dynamic operating system for sensor networks. In *Proc. Third International Conference on Mobile Systems, Applications, And Services (Mobisys)*, 2005.

[23] T. He, S. Krishnamurthy, J. A. Stankovic, T. Abdelzaher, L. Luo, R. Stoleru, T. Yan, L. Gu, G. Zhou, J. Hui, and B. Krogh. Vigilnet: An integrated sensor network system for energy-efficient surveillance. *ACM Transactions on Sensor Networks*, 2004.

[24] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Proc. the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Boston, MA, USA, Nov. 2000.

[25] A. Husker, I. Stubailo, M. Lukac, V. Naik, R. Guy, P. Davis, and D. Estrin. Wilson: The wirelessly linked seismological network and its application in the middle american subduction experiment (mase). *Seismological Research Letters*, May/June 2008.

[26] Intel Corporation. The SHIMMER Sensor Node Platform. 2006.

[27] M. Jones, P. Leach, R. Draves, and J. Barrera. Modular real-time resource management in the rialto operating system. In *Proc. Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, 1995.

[28] A. Kansal, J. Hsu, M. B. Srivastava, and V. Raghunathan. Harvesting aware power management for sensor networks. In *Proc. 43rd Design Automation Conference (DAC)*, San Fransisco, CA, July 2006.

[29] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis. Integrating concurrency control and energy

management in device drivers. In *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP 2007)*, October 2007.

[30] 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.

[31] A. Lachenmann, P. J. Marron, D. Minder, and K. Rothermer. Meeting lifetime goals with energy levels. In *Proc. ACM SenSys*, November 2007.

[32] E. D. Lara, D. S. Wallach, and W. Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *USITS'01: Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems*, pages 14–14, San Francisco, CA, 2001.

[33] X. Liu, P. Shenoy, and M. D. Corner. Chameleon: Application level power management. *IEEE Transactions on Mobile Computing*, 2008.

[34] K. Lorincz, B. rong Chen, J. Waterman, G. Werner-Allen, and M. Welsh. Pixie: An operating system for resource-aware programming of embedded sensors. In *Proc. Fifth Workshop on Embedded Networked Sensors (HotEmNets'08)*, June 2008.

[35] L. Luo, Q. Cao, C. Huang, T. Abdelzaher, J. A. Stankovic, and M. Ward. Enviromic: Towards cooperative storage and retrieval in audio sensor networks. In *Proc. 27th International Conference on Distributed Computing Systems (ICDCS '07)*, June 2007.

[36] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An Acqusitional Query Processing System for Sensor Networks. *ACM TODS*, 2005.

[37] G. Mainland, G. Morrisett, and M. Welsh. Flask: Staged functional programming for sensor networks. In *Proc. 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*, Victoria, British Columbia, Canada, September 2008.

[38] M. Mesarina and Y. Turner. Reduced energy decoding of mpeg streams. In *In MMCN*, pages 73–84, 2002.

[39] D. Narayanan and M. Satyanarayanan. Predictive resource management for wearable computing. In *Proc. ACM MobiSys 2003*, San Francisco, CA, May 2003.

[40] R. Newton, L. Girod, M. Craig, S. Madden, and G. Morrisett. Design and evaluation of a compiler for embedded stream programs. In *Proc. Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2008.

[41] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 276–287, Saint Malo, France, 1997.

[42] S. N. Pakzad, S. Kim, G. L. Fenves, S. D. Glaser, D. E. Culler, and J. W. Demmel. Multi-purpose wireless accelerometers for civil infrastructure monitoring. In *Proc. 5th International Workshop on Structural Health Monitoring (IWSHM 2005)*, Stanford, CA, September 2005.

[43] S. Patel, K. Lorincz, R. Hughes, N. Huggins, J. H. Growdon, M. Welsh, and P. Bonato. Analysis of feature space for monitoring persons with Parkinson's Disease with application to a wireless wearable sensor system. In *Proc. 29th IEEE EMBS Annual International Conference*, August 2007.

[44] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Proc. Second ACM Conference on Embedded Networked Sensor Systems (SenSys)*, November 2004.

[45] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, and I. Stoica. A unifying link abstraction for wireless sensor networks. In *Proc. Third ACM Conference on Embedded Networked Sensor Systems (SenSys)*, November 2005.

[46] V. Shnayder, M. Hempstead, B. rong Chen, G. Werner-Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proc. the Second ACM Conference on Embedded Networked Sensor Systems (SenSys 2004)*, November 2004.

[47] G. Simon et al. Sensor network-based countersniper system. In *Proc. ACM SenSys '04*, November 2004.

[48] J. Sorber, A. Kostadinov, M. Brennan, M. Garber, M. Corner, and E. D. Berger. Eon: A Language and Runtime System for Perpetual Systems. In *Proc. ACM SenSys*, November 2007.

[49] K. Srinivasan and P. Levis. RSSI Is Under-Appreciated. In *Proc. EmNets*, 2006.

[50] T. Stathopoulos, D. McIntire, and W. J. Kaiser. The Energy Endoscope: Real-time Detailed Energy Accounting for Wireless Sensor Nodes. In *Proc. Information Processing in Sensor Networks (IPSN)*, April 2008.

[51] C. Vigorito, D. Ganesan, , and A. Barto. Adaptive control of duty-cycling in energy-harvesting wireless sensor networks. In *Proc. IEEE SECON 2007*, San Diego, CA, 2007.

[52] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proc. Operating Systems Design and Implementation (OSDI 1994)*, November 1994.

[53] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proc. 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*, Seattle, WA, November 2006.

[54] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proc. the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, November 2003.

[55] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *Proc. the 20th SOSP (SOSP '03)*, 2003.

[56] H. Zeng, X. Fan, C. S. Ellis, A. Lebeck, and A. Vahdat. ECOSystem: Managing Energy as a First Class Operating System Resource. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, October 2002.

[57] G. Zhou, J. Lu, C.-Y. Wan, M. D. Yarvis, and J. A. Stankovic. BodyQoS: Adaptive and Radio-Agnostic QoS for Body Sensor Networks. In *Proc. IEEE INFOCOM 2008*, Phoenix, AZ, April 2008.