

# Triage: Balancing Energy Consumption and Quality of Service in a Tiered Microserver

Nilanjan Banerjee Jacob Sorber Mark D. Corner Sami Rollins † Deepak Ganesan

Department of Computer Science  
University of Massachusetts, Amherst, MA  
{nilanb, sorber, mcorner, dganesan}@cs.umass.edu

†Department of Computer Science  
Mount Holyoke College, South Hadley, MA  
srollins@mtholyoke.edu

*Abstract—*

The ease of deployment of wireless and mobile systems is pushing the network edge far from powered infrastructures. A primary challenge in building untethered systems is offering powerful aggregation points and gateways between heterogeneous end-points—a role traditionally played by powered servers. *Microservers* are battery-powered in-network nodes that play a number of roles: processing data from clients, aggregating data, providing responses to queries, and acting as a network gateway. Providing QoS guarantees for these services can be extremely energy intensive; however, it is crucial that the microserver remain energy efficient, as increased energy consumption translates into a larger battery or shorter lifetime.

This paper presents *Triage*, a tiered hardware and software architecture for microservers. *Triage* extends the lifetime of a microserver by combining two independent, but connected platforms: a high-power platform that provides the capability to execute complex tasks and a low-power platform that provides high responsiveness at low energy cost. The low-power platform acts similar to a medical triage unit, examining requests to find critical ones, and scheduling tasks to optimize the use of the high-power platform. The scheduling decision is based on evaluating each task's resource requirements using hardware-assisted profiling of execution time and energy usage. Using three microserver services, storage, network routing, and query processing, we show that *Triage* can provide a 300% increase in microserver lifetime over existing systems. It provides probabilistic quality of service guarantees and meets lifetime goals with an error of less than 5%.

## I. INTRODUCTION

The ease of deployment of wireless and mobile systems is pushing the network edge far from powered infrastructures. Untethered, multi-hop networks support a wide range of applications from wildlife habitat monitoring [20] and security surveillance [16], to space exploration and disaster management [4]. Such applications require the development of highly efficient, long-lived, and low-cost mobile and wireless systems.

Hierarchical network architectures, such as the example shown in Figure 1, promise to balance functionality and energy efficiency in wireless systems. A hierarchical system combines both resource-constrained nodes, such as Motes [22] with resource-rich *microservers* [9]. Microservers form an intermediate battery-powered tier between tethered base-stations or access points, and highly constrained nodes. Microservers provide services including complex sensor data processing [17], providing high-capacity storage to augment storage-limited sensors [2], handling and responding to user queries in a low-latency manner [17], providing greater sensing coverage using higher power and longer range sensors [16], or acting as a gateway between a short range 802.15.4 radio network and a long-range 802.11 network [9]. In all these cases, the microserver waits for requests from either the smaller devices or from users, and responds to them in a timely manner.

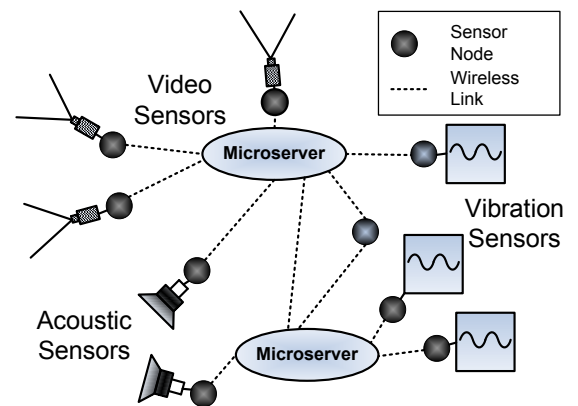


Fig. 1. Microserver Deployment

Designing long-lived microservers poses a unique challenge: *how can a microserver provide long lifetime as well as Quality of Service?* These goals are in direct conflict with one another: supporting QoS guarantees requires examining each request immediately, however providing such vigilance is energy intensive. To see why, consider

Platform	Sleep Power	Platform Startup
TelosB Mote (TI MSP430)	15 $\mu$ W (datasheet)	6 $\mu$ s/0.1 $\mu$ J (datasheet)
Stargate (Intel PXA255)	257mW (measurement)	3s/2.7J

Fig. 2. Energy Consumption in Platforms

the following: normally the microserver remains in a low-power sleep mode. When a request arrives, it transitions to a higher-power state, determines if it must process the request and returns to a low-power state. Unfortunately, platforms that provide sufficient resources to act as a microserver have high energy costs associated with this transition, and that cost must be paid for every request. For instance, the popular PDA-class Stargate requires several seconds to resume operation from a suspended state and more than ten seconds to transition from an off state, as shown in Figure 2. This cost is due to a variety of factors including operating system complexity, the use of large RAM banks, and processor startup. Shallow power saving modes, such as CPU DVFS [7, 10, 18], wireless PSM [1], or disk spin-down [6, 13], require less energy to transition to an active state. However, they consume high amounts of power when there are no requests being serviced and are therefore inefficient for handling infrequent or unpredictable requests. For instance, DVFS on the Stargate platform can trade a 2x slowdown in CPU speed but only obtains a one-third reduction in the energy consumption of the platform.

In contrast to the high transition costs of PDA-class devices, small nodes such as Motes provide very efficient power state transition. This is primarily the result of using a low-power set of components, including a small, simple microcontroller, a low-power radio, small amounts of RAM, and a minimal operating system. Fundamentally, low-power platforms are more efficient for waking and processing small events, exactly the reason they are used in sensor applications, but have insufficient resources to act as a microserver. Thus, reducing the transition time on battery-powered nodes improves energy-efficiency but requires sacrificing capability.

We propose to resolve this tension through a new architecture, *Triage*, that provides the responsiveness of an always-on server, together with an order-of-magnitude greater lifetime than existing microsensors. *Triage* is predicated on the fundamental property that rather than choosing a single hardware platform, the needs of untethered embedded applications are best met by combining platforms with complementary hardware characteristics. *Triage* provides QoS and energy-efficiency through a combination of a high-power, resource-rich platform and a low-power, resource-constrained platform. The low-power

tier, or *tier-0*, remains always-on ensuring responsiveness at minimal energy cost. The high-power tier, or *tier-1*, remains in a power saving mode until its resources are required for a given service. The low-power platform acts similar to a medical triage unit, examining requests to find the critical ones, and acting as a scheduler to minimize the number of times the high-power platform is woken up. Such a scheduling mechanism requires accurate *in-situ* profiling of the time and energy needs of each task, which is performed at the low-power platform. The scheduler optimizes for different criteria—in this paper we focus on two: minimizing energy consumption while meeting soft-realtime latency constraints, and meeting a lifetime goal while satisfying as many task deadlines as possible.

#### A. Research Contributions

*Triage* provides a general mechanism for building highly energy-efficient and QoS-aware microsensors suitable for many untethered applications, including sensor applications, mobile networking, and pervasive computing. Our design and implementation offers three novel contributions.

First, the core of *Triage* is a new system architecture designed for a combination of platforms with complementary characteristics. The *Triage* system targets common functionality required in such scenarios. We have implemented three optimized services: storage, routing and query processing.

Our second major contribution is a set of lightweight, online profiling and scheduling mechanisms. A key challenge that *Triage* addresses is how to place the complex profiling and scheduling logic at the constrained tier-0 platform in order to maximize energy-efficiency without sacrificing QoS. We build a highly optimized *in-situ* profiling service, and two intelligent scheduling algorithms: (a) a soft-realtime scheduler that uses an As-Late-As-Possible (ALAP) scheduling policy to provide deadline guarantees to tasks at minimal energy cost, and (b) a lifetime scheduler that uses a token-bucket energy rate control algorithm to achieve a target microserver lifetime while maximizing the number of tasks that can meet their QoS requirements.

Third, we have built a prototype of *Triage* on a platform combining the Intel/Crossbow Stargate with the TelosB Mote, augmented with a custom fabricated interface board for *in-situ* profiling. On this platform, we show the results of an extensive set of experiments using different workloads. We show that *Triage* provides a 300% increase in microserver lifetime over existing systems. It provides probabilistic Quality of Service guarantees and in addition meets lifetime goals with an error of less than 5%.

## II. TRIAGE ARCHITECTURE

Triage combines the use of multiple hardware platforms to provide efficiency and probabilistic, or soft, Quality of Service guarantees. In this section we describe the design of the Triage hardware and software architecture.

### A. Hardware Architecture

Triage employs a *tiered* hardware platform. The tier-0 platform is a very low-power platform, such as a Mote, and tier-1 is a more capable and higher-power platform, such as a Stargate. In Triage, the two tiers are tightly coupled and directly communicate over a wired link. This enables the lower tier to trigger the wake-up of the higher tier when necessary.

The prototype system that we use combines a Stargate [27] and the TelosB Mote [22]. The TelosB Mote is extremely resource-constrained, but consumes less than one-tenth the power of the Stargate. This platform works well for always-on operation, simple packet processing, and providing low-latency responses. The Stargate platform is significantly more capable but less responsive due to the high latency of sleep to active transitions. We augment this two-tier platform with a custom fabricated interface board that provides necessary voltage conversions for the two platforms, the wakeup interface, as well as a current-sense amplifier and fuel-gauge chip to measure energy-consumption in-situ.

### B. Software Architecture

Figure 3 illustrates the components of the software architecture. Tier-0 virtualizes resources available on tier-1 using a collection of *surrogates*. Surrogates receive requests from the network and can service them in three ways: by using locally cached information, by performing local execution, or by passing them to tier-1 for execution. A *profiler* measures the energy and processing requirements of tasks and a *scheduler* determines, based on the predicted energy cost of a task, when and where it should be executed to meet Quality of Service requirements. Requests that have been scheduled for execution at tier-1 are written into a *log* and delayed until their scheduled execution time. This log also serves as a *cache* of recent requests.

#### B.1 Surrogates

Surrogates are small software modules that run on tier-0 and provide a service such as storage or routing. Though tier-0 can process simple tasks, such as routing updates or time synchronization, most tasks require resources only available at tier-1. When a request arrives at the mi-

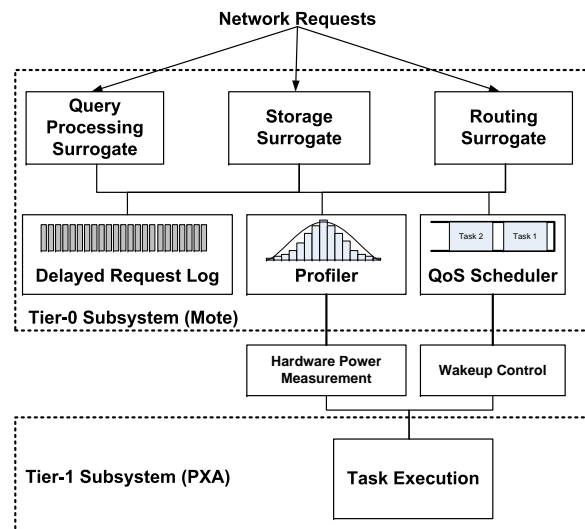


Fig. 3. Microserver Software Architecture

croserver the surrogate performs the following process: (1) immediately execute requests for information cached at tier-0; (2) if a request cannot be serviced from the cache, ask the scheduler to determine where and when to execute the task; (3) if the scheduler determines that the task should be executed at tier-0, execute it immediately; (4) otherwise write the request into the delayed request log.

Triage also uses the delayed request log as a cache for the surrogates. This functionality is particularly useful in storage applications; a read closely following a write to the same data can be serviced from the log. In order to maximize the amount of cached data, Triage does not erase the tier-0 log when a batch of requests is played at tier-1. Instead, the previously committed log entries and cached results are lazily overwritten by new requests using an LRU eviction policy.

To enable applications to compose the functionality of several surrogates, Triage also permits communication between surrogates using primitives provided by the operating system. For instance, a client may query the microserver for information, and request that the results of the query be sent to another node. This requires a combination of a storage surrogate as well as a routing surrogate.

#### B.2 Scheduler and Profiler

Triage uses a scheduler, running on tier-0, to provide QoS. The scheduler relies on a profiler to provide information regarding how long each type of task will take to process, and how much energy it will consume. The profiler measures the execution time of each task and builds a model of task execution time. Using this information, the scheduler determines *where* and *when* to execute each request.

The question of where to execute a task can be answered by comparing the amount of energy required to execute it at each platform. This decision may be even simpler if the task requires the resources of tier-1 and cannot be executed on tier-0. The question of when to execute requests is more complicated. There are two cases when the scheduler must wake tier-1 and dispatch outstanding tasks. The first case occurs when the log becomes full. In this case, the scheduler is automatically invoked by the log storage system; it wakes tier-1 and dispatches each outstanding task to the appropriate service. The second case requires the scheduler to consider Quality of Service constraints. Each request may arrive with a soft-realtime deadline. In order to meet the deadlines with maximum energy efficiency, the scheduler will delay execution of tasks as long as possible to increase the amount of time tier-1 remains in a low-power state. We describe the algorithms used by the scheduler and profiler in Section IV.

### III. EXAMPLE SURROGATES

The most common, and basic, functions found in servers for sensor networking, mobile networking, and pervasive computing are routing, storage, and query processing. To this end we present three example surrogates: a storage system surrogate, a network routing surrogate, and a query processing surrogate. As untethered networks proliferate, this library of surrogates will be expanded, enhanced, and further optimized.

#### A. Storage System Surrogate

The storage surrogate enables in-network storage applications. It accepts read, write, and delete requests for the tier-1 storage system. Upon receiving a request from the network, it first determines whether the request is a read request that can be satisfied by a recent write that is cached in the delayed request log. If so, it immediately provides the result. Otherwise, it asks the scheduler to schedule the task. The scheduler considers any latency constraint provided with the task and tells the surrogate when the task has been scheduled. The surrogate then inserts the task into the delayed request log.

#### B. Network Routing Surrogate

The network routing surrogate enables efficient routing by utilizing both the tier-0 and tier-1 network interfaces. When a packet arrives at the surrogate, it examines the destination address, consults a routing table, and determines over which radios the destination is reachable. It immediately passes this information, along with any latency constraint, to scheduler which determines which radio should be used to send the packet and when the packet should be

sent. If scheduler determines that the tier-0 radio should be used, the packet is sent immediately. Otherwise, the packet is inserted into the delayed request log.

#### C. Query Processing Surrogate

The query processing surrogate provides a simple query interface for data stored on the microserver. Clients may use simple queries, such as *retrieve all images from the last ten seconds*, or more complex queries, such as *retrieve all images that contain 2 or more objects and are from a particular geographic region*. When the surrogate receives a query, it first determines whether the query is a simple query that can be executed over data cached in the delayed request log. We assume that tasks are statically mapped into simple queries, which can be executed at either tier, and complex queries that require the resources of tier-1. If the query can be executed at tier-0, it is executed immediately. Otherwise, the surrogate passes the query and any latency constraint to the scheduler. Once the scheduler has scheduled the query, it is inserted into the delayed request log.

### IV. PROFILING AND SCHEDULING ALGORITHMS

At the heart of Triage is a profiling engine that is used to estimate the execution time and energy usage for different tasks, and a scheduling engine that determines how to meet QoS constraints. In this section, we describe the algorithms employed by the profiling engine and the scheduling engine to enable energy-efficient scheduling of tasks.

#### A. Task Profiling Algorithm

Triage employs a profiler to measure the execution time and energy usage for different tasks. In this discussion, we focus on determining the *typical energy usage* and *typical execution time* for each type of task. Such online profiling is necessary to deal with the variability in execution time and energy usage of tasks that involve a combination of processing, communication and storage.

Online profiling involves two steps, task grouping and parameter estimation. The online profiling engine first identifies a task as belonging to a certain group based on the nature of task. This grouping information is assumed to be provided a priori by the system designer. We believe that such a grouping is appropriate since many applications of microsensors involve a small and well-specified set of tasks. For instance, in a camera sensor network, a typical set of tasks might be {Motion Detection, Face Recognition, Store Image, Send Data}.

For each of these task groups, the profiler uses a separate history of execution times and energy consumption to build corresponding probability distributions. We focus on

the estimation of the typical execution time since a similar algorithm can be used for estimating typical energy usage.

Let  $f(t_i)$  be the probability distribution of time taken to execute task type  $i$ . Further, let  $\bar{X}_i$  and  $\sigma_i$  denote the average time and standard deviation for task type  $i$ . The profiler uses the Chebyshev's inequality (shown in Equation 1) to determine an interval of time such that the task executes within that interval with probability at least  $p$ .

$$Time(i, p) = [\bar{X}_i - \frac{\sigma_i}{\sqrt{1-p}}, \bar{X}_i + \frac{\sigma_i}{\sqrt{1-p}}] \quad (1)$$

The parameter  $p$  can be tuned depending on the guarantee required by a user. For instance, in the camera sensor network used for surveillance, a `Face Recognition` task might require a tight guarantee as the person might move out of the field of view of the camera sensors. In this case,  $p$  can be set to a high value, say 0.9. Other tasks such as `Send Data` might be more elastic, and `Store Image` might not have any deadline at all.

Using information collected after the task executes, the profiler builds two kinds of dynamic models: histograms and parametrized models. When prior models of execution time are unavailable, a simple histogram approximates the probability distribution,  $f(t_i)$ , and tracks bins of execution times and energy consumption for each task group. In contrast, when prior models are available, these can be used to more accurately model task execution time and energy consumption. For instance, the execution time and energy used in a communication task is a linear function of the number of bytes transmitted. In this case, the execution times are fit to a simple linear model to determine the costs of the two radios in Triage. Our prototype uses parametrized models for the storage and network surrogates tuned to the size of the file and the size of the packet respectively. More complicated models can be built for applications such as video coding, compression, and encryption. For the query processing surrogate we use the histogram-based profiling.

### B. Task Scheduling Algorithm

The scheduler that resides on tier-0 uses the profiling information about tasks to minimize the number of times tier-1 is woken up while still satisfying the task deadlines. The Triage scheduler uses different algorithms depending on the optimization criteria. In this work, we discuss two schedulers — the first is optimized to satisfy task deadlines, and the second is optimized to achieve a target lifetime for the microserver. While we limit our discussion to these two schedulers, we note that alternate schedulers that optimize, or balance, other QoS constraints can be plugged

into our system.

#### B.1 Scheduling for Deadline Constraints

The deadline scheduler tries to minimize the energy consumed by the microserver, such that the deadlines of the incoming tasks are met. Let the set of tasks which are already batched at tier-0 for delayed execution at tier-1 be denoted by  $S = \{T_1, \dots, T_k\}$  where task  $T_i$  has deadline  $D(T_i)$ , latest start time  $L(T_i)$ , and execution time  $E(T_i)$ . The latest start time is the latest time at which a task can begin executing on tier-1 such that the deadlines of all tasks after and including itself are met, and the execution time is the time it takes to execute the task on tier-1. Further, we assume that the list  $S$  is sorted by deadlines *i.e.*  $D(T_i) > D(T_j)$  if  $i > j$ . Let the wakeup time for tier-1 be  $W$ , and the current batch time,  $B$ , correspond to the latest time at which tier-1 needs to be woken up such that the deadlines of all tasks in the list  $S$  can be satisfied.

The scheduling framework that we propose is based on the well-known As Late As Possible (ALAP) scheduler. When a new task arrives, the scheduler first queries the profiler for the typical execution time for the task at tier-1 at the lowest power mode. Next, the algorithm recomputes the batch time,  $B$ , *i.e.* the latest time at which tier-1 can be woken up such that all the batched tasks and the new task meet their deadlines. Let the new task be inserted at index  $l$  into the sorted list  $S$  based on its deadline. The scheduler now needs to ensure that the insertion of the new task does not result in missed deadlines for any of the other tasks in the list. The scheduler only lowers the batch time and never increases it, hence only the tasks that are before  $T_l$  in the list need to be checked for deadline violation. Thus, for each task  $T_i : l \geq i \geq 1$ , the scheduler sets the latest start time such that it does not violate the deadline constraint of  $T_i$  or any task with deadline after  $T_i$ , *i.e.*  $L(T_i) = \min(L(T_i), L(T_{i+1}) - E(T_i))$ . The new batch time,  $B$ , is updated to reflect the latest start time for the first task in the list, *i.e.*  $B = L(T_1) - W$ . If  $B \leq 0$ , tier-1 is immediately woken up and the batch executed. If  $B > 0$ , a timer will fire at time  $B$  and tier-1 will be woken up. The time required to update the schedule is linear in the number of tasks currently in the queue.

We illustrate the deadline scheduling algorithm with a simple example, shown in Figure 4. Let there be two batched tasks,  $T_A$  with deadline 60 seconds and execution time 3 seconds, and  $T_B$  with deadline 64 seconds and execution time 3 seconds. The latest start times of the two tasks are  $L(T_A) = 57$  seconds and  $L(T_B) = 61$  seconds respectively, and the batch time,  $B$  is 50 seconds, assuming a tier-1 wakeup time,  $W$  of 7 seconds. Now, a new task  $T_C$  arrives with deadline 62 seconds and execution time 3

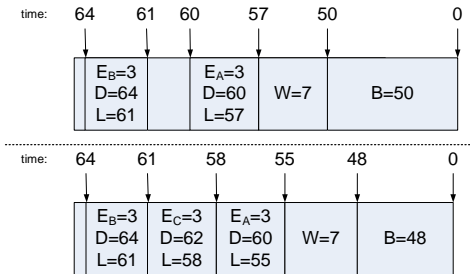


Fig. 4. ALAP Example: The figure shows the execution time and deadline for each task, the wake up latency for tier-1, and the resulting batching time. A new task  $T_C$  is inserted into the scheduling decreasing the batch time.

seconds. The task is inserted between  $T_A$  and  $T_B$ . The scheduler checks whether the current batch time satisfies  $T_C$ 's deadline, notices a violation, and pushes  $T_A$  forward in the schedule. Hence, the batch time is set to 48 seconds.

## B.2 Scheduling for a Lifetime Constraint

Though the goal of the deadline scheduler is to miss only a small percentage of deadlines while minimizing energy usage, the objective of the lifetime scheduler is to meet a target lifetime for the microserver while satisfying as many deadlines as possible. The scheduler should also be capable of handling periods of burstiness. To accomplish this we use a token-bucket algorithm for the lifetime scheduler. Given a target lifetime,  $L$ , and battery capacity,  $E$ , energy tokens are generated at a constant rate of  $\frac{E}{L}$ . The total number of accumulated tokens represents the amount of energy that is available for use by the system. The amount of energy used by the system is continually monitored by the energy profiler, and is queried periodically by the scheduler to determine the rate at which energy is depleted by the system. The difference between the accumulated energy tokens and depleted energy tokens at any time represents the surplus of energy that can be used by the system to execute the batched tasks.

The lifetime scheduler builds on the deadline scheduling algorithm that we described in Section IV-B.1. When a new task arrives, the deadline scheduling algorithm is used to queue the task and determine the batch time. When this batch time becomes zero, the lifetime scheduler checks to see whether there are sufficient energy tokens to wakeup tier-1, execute all the tasks in the batch, and shutdown tier-1. If so, tier-1 is woken up and the tasks are executed before shutting it down. The energy profiler is queried to determine how much energy was used during this batched processing, and the number of available energy tokens is updated accordingly. If the number of energy tokens is insufficient to execute the batch, the wakeup of the tier-1

platform is delayed until sufficient tokens have accumulated. During this period of waiting for tokens to accumulate, task deadlines could be missed, and tasks could be dropped if the size of the task queue exceeds the storage capacity of the tier-0 platform.

Neither of our scheduling algorithms take into account the availability of DVFS states at tier-1. Evaluating and profiling multiple DVFS states would possibly permit greater efficiency by allowing tier-1 to sleep longer, however this pushes the complexity of the scheduler to  $O(k^n)$  for  $n$  tasks and  $k$  power modes. We are currently investigating approximate heuristics that are computationally feasible for the tier-0 node.

## B.3 Idle State Management

One remaining issue is the state in which the scheduler leaves tier-1 while it batches new requests at tier-0—there are two options, suspension and shutdown. Suspension requires more power than shutting down the platform, but the transition cost is lower. For example, the Stargate platform with a CF 802.11 card draws 250mW in suspend mode and costs 2.7J and 20.1J of energy to wake up from suspend and shutdown respectively.

The Triage scheduler determines the appropriate idle state for tier-1 based on the expected arrival time of tasks that require tier-1. The expected idle time is computed using the inter-arrival times for the 50 most recent tasks. We estimate the cost of each state based on the expected idle time, and choose the state that minimizes that cost.

While this approach is sufficient for many applications, the choice of idle state does enforce a minimum latency which the microserver can support. For example, if tier-1 is shutdown due to infrequently arriving tasks, the next task that requires tier-1 will have to wait at least as much time as tier-1 requires to wake from shutdown. In the case of the Stargate, this minimum latency is 12 seconds. In order to deal with this problem, more sophisticated profiling techniques could be used to anticipate the latency requirements of upcoming tasks and choose the proper idle state accordingly.

## V. IMPLEMENTATION

In order to evaluate our approach we have implemented a working prototype of Triage, shown in Figure 5.

### A. Prototype Hardware

We constructed a prototype using a Crossbow Stargate (tier-1) [27] and a TelosB Mote (tier-0) [22]. The Stargate contains a 32-bit, 400MHz PXA255 XScale processor, 64 MB of RAM, 32 MB of internal flash, and a WiFi interface. The TelosB Mote contains an 8-bit, 8 MHz

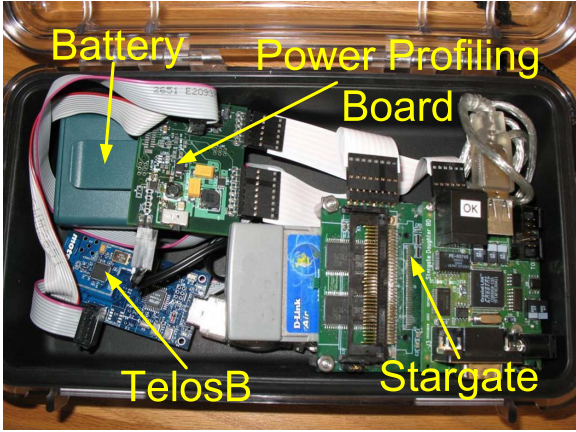


Fig. 5. Prototype Triage System

microcontroller, 10kB of RAM, 1 MB of external flash, and an 802.15.4 radio. These hardware platforms were chosen because they handle the range of workloads that we have targeted, are separated in power consumption by more than an order-of-magnitude (300mW-2500mW and 20mW-120mW), are easily programmable, and are well supported. The Stargate platform runs Linux, making available a broad range of software tools and services.

We fabricated an additional power supply board for attaching both boards to a single battery. This board provides two additional hardware elements necessary to Triage, a Maxim DS2770 fuel gauge chip and a Maxim MAX4072 current sense amplifier. The fuel gauge chip gives accurate readings of the energy left in the battery, while the current sense amplifier gives Triage the ability to measure the power consumption of each task that runs on the Stargate. As described, the scheduler uses this profiling information to control the platform energy policy.

One limitation of our current implementation is the transfer speed from the TelosB Mote to the Stargate. The two devices communicate over a USB line which is limited to 230 kbps. However, data needs to be read from the flash and then transferred over the USB which increases the total time required for the transfer. As a result, transferring 1024 KB of batched work along with protocol overhead takes more than 150 seconds and wastes a great deal of energy while blocked on serial I/O. We are currently investigating more efficient means of communication. Though, even with this limitation the current prototype shows extremely high gains in energy efficiency.

### B. Surrogates and Log

As part of this prototype, we implemented three surrogates: storage, network routing, and query processing. The delayed request log is managed on the TelosB’s flash storage using a custom designed file system. We imple-

TelosB Max. Power Consumption	120 mW
Stargate Bootup Energy	20.1 J
Stargate Bootup Time	12 s
Stargate Resume Energy	2.7 J
Stargate Resume Time	3 s
Stargate Suspension Power	257 mW

Fig. 6. Platform Measurements

mented these components as TinyOS modules written in nesC [8]. The routing, storage, and query processing surrogates comprise 430, 1200, and 500 lines of nesC code respectively, and the log storage system consists of roughly 1800 lines of code. The profiler and scheduler consist of 1500 lines of code. We also implemented an execution engine that runs on the Stargate and executes tasks when they are received from the Mote <sup>1</sup>.

## VI. EVALUATION

Our evaluation answers a number of key questions. First, we provide microbenchmarks that validate our use of a two-tier platform to achieve a combination of capability, energy-efficiency and responsiveness. Second, we evaluate the deadline and lifetime schedulers, and demonstrate their effectiveness in achieving their goals. Third, we evaluate how the networking surrogate can adapt to the inter-arrival times of requests and compare our results against a Wake-On-Wireless style system [25]. Fourth, we focus on the performance of the profiler, and its accuracy when used with the networking surrogate. Finally, we demonstrate how the storage surrogate exploits caching at tier-0 to improve energy-efficiency.

We designed several experiments in order to answer these questions. Except where noted, each experimental point is the result of a single experiment that runs for more than 10 minutes—this is long enough for the results to stabilize.

### A. Static Energy Costs

In order to provide better intuition into the behavior of our prototype, we measure the static energy costs that directly impact Triage’s performance. These values, shown in Figure 6, are the basis for the energy savings achieved by Triage. Specifically, the Stargate’s transition from suspend to active costs as much energy as 23 seconds of computation on tier-0. Transitioning from shutdown to active is equivalent to 170 seconds of active tier-0 computation. Replacing expensive state transitions at tier-1 with low-power tier-0 computation results in significant overall energy savings. Also, note that the Stargate’s suspend power

<sup>1</sup>The source code and hardware schematics will be available at time of publication.

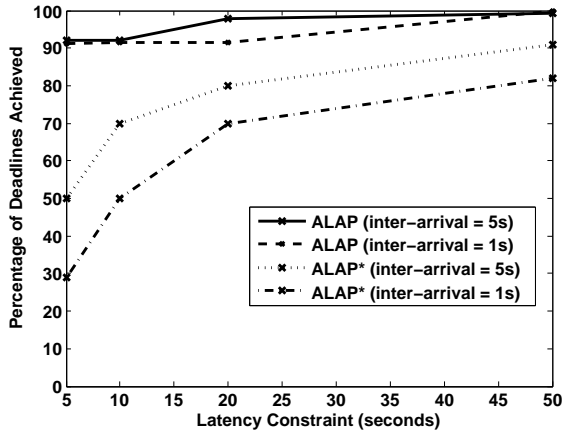


Fig. 7. Deadline scheduler. This figure shows the number of soft-realtime constraints met by the ALAP scheduler and the ALAP\* scheduler. The ALAP\* scheduler does not use profiling information and thus meets fewer deadlines than the 90% goal.

is higher than expected due to a well-known bug in how the platform manages the CF card slot. Therefore our prototype should be viewed as a conservative estimate of the potential energy savings that can be achieved.

### B. Soft-Realtime Scheduling

In order to evaluate the ALAP scheduling algorithm, we observe how a Triage microserver performs in the presence of different latency constraints. In this experiment, we use the Query Processing surrogate to answer periodic queries requesting any images that contain five objects, from a pool of 100 images stored on the Stargate’s flash. These images could correspond to events detected by a network of small cameras and stored remotely at the microserver. Objects within images are detected using a standard image processing algorithm. The time required to process each query varies between 600 and 1200ms with an average of 800ms. Each query also has a latency constraint, which specifies the amount of time within which the client expects a response. We show results from the Triage ALAP scheduler with task profiling and from ALAP without task profiling (labeled ALAP\*) for query inter-arrival times of 1 and 5 seconds. In both cases the scheduler wakes tier-1 and processes all delayed tasks in a single batch. The profiler uses a threshold  $p = 0.9$  in Equation 1 to determine the typical execution time of each task. We vary the latency constraint and measure the number of deadlines each system meets, as well as the power the microserver consumes, to produce the results shown in Figures 7 and 8.

The first thing to note in Figure 7 is that for each workload the Triage ALAP scheduler is always able to meet at least 90% of the latency constraints. This demonstrates

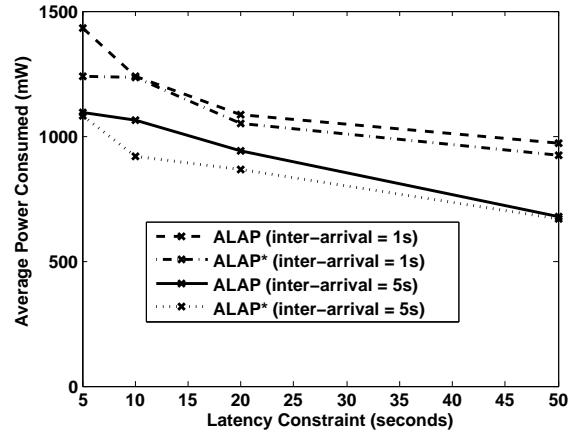


Fig. 8. Average Power Consumption for the deadline scheduler. The ALAP scheduler consumes slightly more power in order to meet more deadlines.

both the accuracy of the Triage profiler, which is able to precisely determine the computational needs required for these queries, and the accuracy of the ALAP scheduler, which correctly schedules the wakeup of tier-1 to meet the desired latency constraints. Without profiling the execution time of tasks, the ALAP\* scheduler is unable to determine when to wake tier-1 and regularly misses deadlines, especially when latency constraints are small. This is because scheduling errors are more likely occur at the beginning of each batch of tasks. Allowing longer latencies, results in larger batches of tasks and more tasks that are processed ahead of their deadlines.

The power consumption data in Figure 8 shows three results. First, as the latency constraint is relaxed, the microserver is able to batch more requests, amortizing the wakeup cost of tier-1 over more requests. Second, the ALAP scheduler consumes marginally more power than ALAP\*. This is a result of tier-1 waking up earlier order to meet deadlines. This results in slightly smaller task batches and therefore uses slightly more power.

### C. Lifetime Scheduling

To show that the lifetime scheduler is able to meet a lifetime goal, we subject the microserver to a similar experiment as before using Query Processing. In order to expedite this experiment, we use a small battery capacity of 100mAh—enough energy for the microserver to operate at maximum load for 14 minutes. We set the lifetime goal of the server to be 60 minutes. For the first 30 minutes, queries arrive every 180 seconds. For the remaining time, the server sees a more intense load with queries arriving every 15 seconds. Figure 9 shows the results of this experiment. The slope of the straight line demonstrates the lifetime goal divided by the energy capacity of the server—

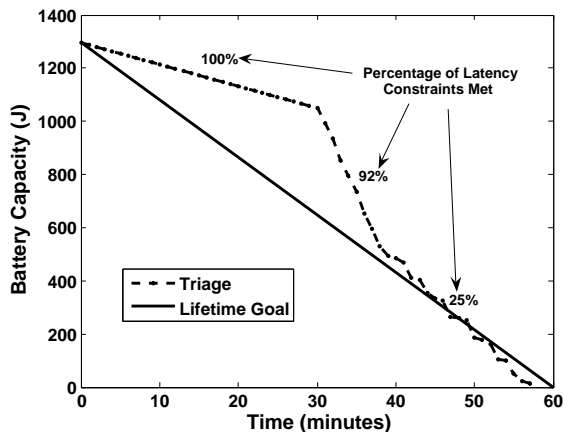


Fig. 9. Lifetime Scheduler. This figure shows Triage’s use of the lifetime scheduler for a goal of 60 minutes. For the first 30 minutes the load is light and the microserver accumulates an energy surplus. For the last thirty minutes it uses the surplus at the determinant of meeting deadlines.

this is the overall average power goal.

Recall that when using the lifetime scheduler, Triage prioritizes lifetime over latency constraints. It attempts to meet the latency constraint whenever it can, and the token-bucket algorithm allows for bursts of short, energy intensive workloads. For this algorithm to operate correctly Triage must accurately profile the energy use of tasks and track the overall energy consumption of the microserver to account for profiling error. For the first 30 minutes, the server consumes less energy than is required to meet its lifetime goal. During this time the server’s workload is insufficient to drain the bucket, so Triage is free to schedule all tasks and therefore meets its deadlines. After operating for 10 minutes under a more intense workload, and Triage continues to meet its deadlines, but it begins to consume the surplus energy that has accrued. At 38 minutes, Triage runs out of surplus energy and begins to sacrifice latency constraints for conserving energy. At 58 minutes Triage runs out of energy altogether, within 5% of meeting its lifetime goal of one hour.

#### D. Scaling to Inter-Arrival Time

The primary goal of the deadline scheduler is to meet latency constraints for each task. A secondary goal is to scale the energy consumption of the server according to the system load. Recall from Section IV-B.3 that Triage tracks the arrival rate of tasks and chooses one of the following operating states for tier-1 after a batch has been processed: suspend or shutdown. We examine the effect of this decision on the performance of the networking surrogate. In this experiment an application periodically sends 60kB of data to the networking surrogate to route to a neighbor.

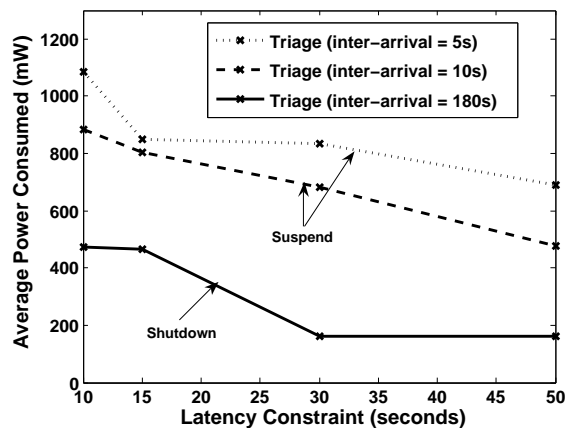


Fig. 10. Idle State Management. This shows how Triage chooses an idle mode based on the inter-arrival of tasks. Given a sufficiently long inter-arrival time it reduces the energy consumption of the microserver using suspension or shutdown.

The networking surrogate chooses between the 802.15.4 radio on the TelosB Mote, or the 802.11 radio on the Star-gate based on the latency requirements of the application. In our experiments we vary the frequency of these transfer events, as well as the latency constraint. Figure 10 shows the resulting power consumption.

Recall that the networking surrogate profiles the time and energy associated with sending data over each of the two radios, generating a parametrized model of both time and energy. As seen previously, longer latency constraints allow Triage to batch up more tasks before waking up tier-1. Figure 10 shows that power consumption drops more dramatically as the inter-arrival times increase. This is a result of Triage taking advantage of the suspension and shutdown modes to save power. For an inter-arrival time of ten seconds, the scheduler determines that it would be too inefficient to shut the tier-1 system down and it uses suspension. For a 180 second inter-arrival time the power savings of fully shutting down the tier-1 node doesn’t outweigh the increased transition costs of rebooting the device. Also, note that the far left point on the bottom line is unable to achieve the desired latency constraint because tier-1 is shutdown. In this case described in Section IV-B.3, each task will have to wait 12 seconds while tier-1 boots.

#### E. Task Profiling

The profiling function of the microserver is essential to providing soft-realtime guarantees. We demonstrate the benefits of the profiler using an evaluation of the network surrogate. We use the same experimental setup as the previous experiment with a fixed latency constraint of 17 seconds and fixed inter-arrival time of 180 seconds. We show

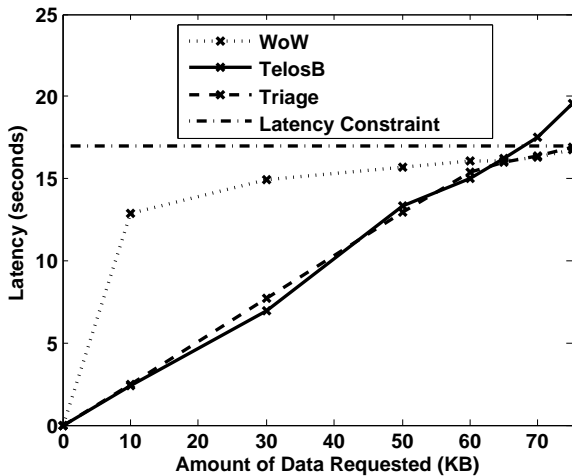


Fig. 11. Time Profile Accuracy. This shows the networking surrogate choosing between two radios based on a latency constraint. For 68KB of data it switches from the 802.15.4 radio to the 802.11 radio to meet the constraint.

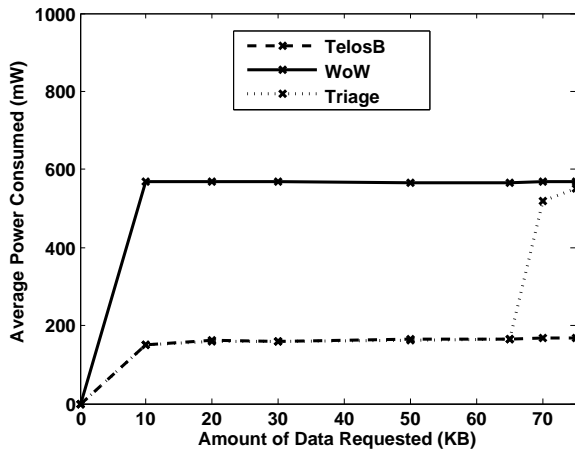


Fig. 12. Time Profile Power Consumption. This shows the networking surrogate power consumption based on the amount of data it sends. At 68KB of data it must switch to the 802.11 radio to meet the latency constraint, thus using more power. The WoW solution uses more power as it only uses the 802.11 radio to send data.

Triage in comparison to two other systems, one which solely uses the 802.15.4 radio on the tier-0 TelosB node, and one which wakes the tier-1 node on the receipt on every request (labeled WoW)—this is similar to a Wake-On-Wireless style system [25]. We vary the packet sizes and show the resulting latency and power in Figures 11 and 12.

Comparing these systems we see several effects. First, Triage is able to profile the time required for networking tasks correctly and send them by their required latency constraint. Triage uses the 802.15.4 radio at the lower data rates, and above 68KB of data it must wake the tier-1 system to still make the latency constraint. From Figure 11, this corresponds to the point where the Telos-only line and

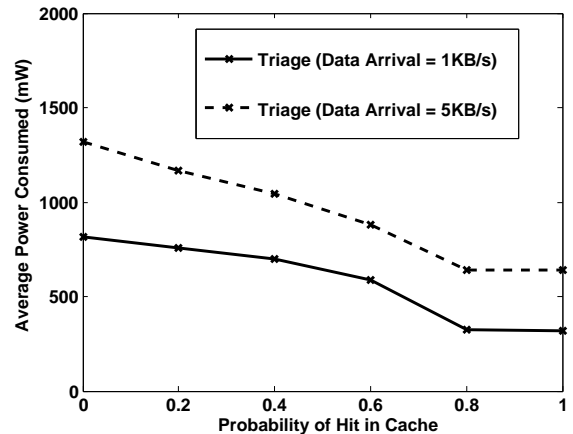


Fig. 13. Tier-0 Caching. This shows the benefit of caching in the storage surrogate. The greater the hit rate in the cache, the less the microserver must use the tier-1 system, thus reducing the power consumption of the microserver.

the latency constraint cross—this is the point where the 802.15.4 radio is unable to send the data by the latency constraint. Second, the WoW system can always send the required data by the latency constraint, however it consumes five times more average power than Triage for small transfers. As Triage can route data using the TelosB-node alone it can achieve a 300% increase in lifetime over a system that only uses one high power radio to send packets.

### F. Caching

In addition to Quality of Service constraints, cache performance also affects Triage’s efficiency. Recall that the storage surrogate can immediately provide read requests for data that is cached at tier-0, without waking tier-1. We examine how system power consumption varies with respect to cache hit rate and how caching at tier-0 lowers the power consumption of the microserver. We use a remote node to send write requests to the server at two different rates, 1 KB/s and 5 KB/s. A different node sends a read request every 10 seconds to the microserver for the same data, however we control the read requests in order to fix the hit rates in the tier-0 cache. The resulting power consumption of the server is shown in Figure 13.

The results show that if read requests exhibit high locality the Triage system achieves a 2X improvement in power consumption over using delayed execution alone. Note that as the hit rate approaches 1, the wake up cost incurred by incoming data begins to dominate the cost of cache misses, resulting in a nearly constant power consumption from 0.8 to 1.0. The left edge of the graph represents the system performance without the use of caching.

## VII. RELATED WORK

The design and implementation of Triage draws from several related research areas, which we survey here.

### A. *Microservers and Clustering*

Several sensor network systems utilize a subset of the participating nodes as aggregators, central processing nodes, or gateways [9]. This work can be classified into algorithms for networks of homogeneous devices and algorithms for networks of heterogeneous devices. In homogeneous systems such as Heed [28], LEACH [12], the leader, or clusterhead, rotates among nodes in the network. The goal is to distribute the extra energy drain incurred by the leader. In heterogeneous systems, larger, more powerful nodes called microservers *herd* other smaller nodes [11]. Our work focuses on the latter scenario and addresses the need for a power-aware software architecture to reduce the energy drain on the resource-rich nodes.

### B. *Energy Management*

Reducing the power consumption of mobile devices has been the subject of much research. Approaches include scaling the CPU voltage and frequency [10], managing wireless interface usage [1], turning off banks of RAM [14], or employing microsleep [3, 15]. In a larger device, such as a Stargate, these techniques still do not enable a power mode comparable to a mote device. While these efforts target optimization for laptops and PDAs, our architecture targets microservers for wireless networks, and is designed to exploit a hardware platform with complementary components.

Papathanasiou and Scott made an observation similar to ours: batching work, or increasing idle periods, leads to greater energy efficiency [21]. However, the goal of their work was to increase burstiness in laptop disk drives.

The Wake-on-Wireless project (WoW) [25] proposes a hierarchy of devices for PDAs, including a low-power receiver that can wake the PDA. Our goal is similar to WoW, to reduce power consumption in battery powered devices. Their focus was solely on exploiting low-power radios, whereas Triage tackles the significantly broader problem of building energy-efficient, QoS-aware microservers.

### C. *Sensor Platforms*

Recently, many embedded sensor platforms have emerged. These platforms span a broad spectrum of power requirements and functionality. A popular instance of sensor platforms is the family of Motes. These nodes are commercially available, widely used, and include the Crossbow MicaZ and Mica2Dot as well as the Telos node. All

of these nodes consume peak power between 10-120mW and are tuned to be highly power efficient.

There are also several more capable but still very power-efficient sensor nodes such as the Yale XYZ [19]. This node has dynamic frequency scaling capability and can operate between 2MHz and 56MHz with a power consumption of up to 3x greater than a Mote at comparable clock speeds. Such intermediate platforms can be used as clusterheads in applications that have moderate computation requirements.

Our architecture targets resource-rich but power efficient sensor platforms that combine two processing elements—one small and one large. Such architectures have been used in other research efforts. The Stargate platform [27] incorporates a connector to a Mica or Telos Mote, but the intention was to provide a gateway for Mote radios, rather than to optimize the energy efficiency of the platform. The PASTA node is an architecture that combines a trip-wire board with a DSP processor together with a PXA processor [24], and the LEAP platform [5] integrates a higher-end processor-radio module (Intel PXA255 XScale running Linux) with a lower-end processor-radio module (TI MSP430). The Turducken system [26] also employs multiple hardware tiers in the context of an always-on laptop system. While these efforts employ hierarchical structures, they do not provide software architectures or algorithms for intelligently controlling the use of the hardware infrastructure.

## VIII. CONCLUSIONS

This paper presents the design, implementation, and evaluation of Triage, an architecture for QoS-aware, energy-efficient microservers. Our work exploits hierarchical hardware with two connected platforms, one with high capability for executing a batch of tasks and one with high energy-efficiency while waiting for new tasks to arrive. A novel aspect of our work is our energy and QoS-optimized scheduler that uses extensive *in-situ* profiling capability to efficiently execute storage, communication and processing tasks. We demonstrate the use of two schedulers in Triage, an As-Late-As-Possible scheduler that optimizes for task deadlines, and a token-bucket based scheduler that optimizes for node lifetime. Our results show that Triage provides a 300% increase in microserver lifetime over existing systems. It provides probabilistic quality of service guarantees and in addition meets lifetime goals with an error of less than 5%.

While our Triage prototype demonstrates the potential for long-lived QoS-aware microservers, we believe that the ceiling is significantly higher than what we have demonstrated in this paper. First, we only considered an always-

on tier-0 platform since efficient duty-cycling support is not yet available for the CC2420 radio on the Telos Mote. We believe that Triage can provide significantly more benefits if tier-0 were also duty-cycled, especially when requests are infrequent. For instance, we can reduce the tier-0 energy by a factor of 10 while increasing average latency by 1 second [23]. Second, the hardware used in the current prototype is not ideal for a number of reasons including the large latency in waking the Stargate from suspension, its high suspension power, and the low bandwidth available to transfer data from the Telos to the Stargate. The large latency to wake the Stargate is the greatest limiting factor as it restricts the smallest latency requests that the microserver can support. As low-power platforms continue to evolve we expect that most of these bottlenecks will improve making Triage an even more useful and efficient system for untethered sensor deployments, mobile computing, and ubiquitous computing.

#### REFERENCES

- [1] M. Anand, E. B. Nightingale, and J. Flinn. Self-tuning wireless network power management. In *Proceedings of the 9th ACM International Conference on Mobile Computing and Networking (MobiCom'03)*, San Diego, CA, September 2003.
- [2] R. Balani, S. Han, V. Raghunathan, and M.B.Srivastava. Remote Storage for Sensor Networks. UCLA-NESL-200504-09, UCLA, 2005.
- [3] L. S. Brakmo, D. A. Wallach, and M. A. Viredaz. microSleep: A technique for reducing energy consumption in handheld devices. In *Proceedings of the Second International Conference on Mobile Systems, Applications, and Services (MobiSys'04)*, Boston, MA, June 2004.
- [4] B. Burns, O. Brock, and B. N. Levine. MV routing and capacity building in disruption tolerant networks. In *Proceedings of IEEE Infocom 2005*, March 2005.
- [5] K. Ho D. McIntire, B. Yip, A. Singh, W. Wu, and W. J. Kaiser. The low power energy aware processing (leap) embedded networked sensor system. Technical report, UCLA, Los Angeles, CA, 2005.
- [6] F. Douglass, P. Krishnan, and B. N. Bershad. Adaptive disk spin-down policies for mobile computers. In *Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing*, April 1995.
- [7] K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance-setting for dynamic voltage scaling. In *Proceedings of the Seventh ACM International Conference on Mobile Computing and Networking (MobiCom'01)*, Rome, Italy, July 2001.
- [8] D. Gay, P. Levis, R. V. Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation (PLDI)*, June 2003.
- [9] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems*, Baltimore, MD, 2004.
- [10] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proceedings of the First ACM International Conference on Mobile Computing and Networking (MobiCom'95)*, Berkeley, CA, November 1995.
- [11] Ramesh Govindan, Eddie Kohler, Deborah Estrin, Fang Bian, Krishna Chintalapudi, Om Gnawali, Sumit Rangwala, Ramakrishna Gummadi, and Thanos Stathopoulos. Tenet: An architecture for tiered embedded networks. Technical Report TR-56, CENS, November 2005.
- [12] Wendi Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient communication protocols for wireless microsensor networks. In *Proceedings of the Hawaiian International Conference on Systems Science*, January 2000.
- [13] D. P. Helmbold, D. D. E. Long, and B. Sherrod. A dynamic disk spin-down technique for mobile computing. In *Proceedings of the Second ACM International Conference on Mobile Computing and Networking (MobiCom'96)*, Rye, NY, November 1996.
- [14] H. Huang, P. Pillai, and K. G. Shin. Design and implementation of power-aware virtual memory. In *Proceedings of USENIX Technical Conference*, San Antonio, TX, June 2003.
- [15] N. Kamijoh, T. Inoue, C. M. Olsen, M. T. Raghunath, and C. Narayanaswami. Energy trade-offs in the IBM wristwatch computer. In *Proceedings Fifth International Symposium on Wearable Computers*, Zurich, Switzerland, October 2001.
- [16] P. Kulkarni, D. Ganesan, and P. Shenoy. Senseye: A multi-tier camera sensor network. In *ACM Multimedia*, 2005.
- [17] M. Li, D. Ganesan, and P. Shenoy. PRESTO: Feedback-driven data management in sensor networks. In *Proceedings of the 3rd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2006)*, May 2006.
- [18] J. R. Lorch and A. J. Smith. Reducing processor power consumption by improving processor time management in a single e-user operating system. In *Proceedings of the Second ACM International Conference on Mobile Computing and Networking (MobiCom'96)*, Rye, NY, November 1996.
- [19] D. Lymberopoulos and A. Savvides. XYZ: A motion-enabled, power aware sensor node platform for distributed sensor network applications. In *Proceedings of Information Processing in Sensor Networks (ISPN)*, Los Angeles, CA, April 2005.
- [20] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Workshop on Wireless Sensor Networks and Applications*, Atlanta, GA, September 2002.
- [21] A. E. Papatnasiou and M. L. Scott. Energy efficiency through burstiness. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, Monterey, CA, October 2003.
- [22] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors (IPSN/SPOTS)*, April 2005.
- [23] Joseph Polastre, Jason Hill, and David E. Culler. Versatile low power media access for wireless sensor networks. In *SenSys*, pages 95–107, 2004.
- [24] B. Schott, M. Bajura, J. Czarnaski, J. Flidr, T. Tho, and L. Wang. A modular power-aware microsensor with 1000x dynamic power range. In *Proceedings of Information Processing in Sensor Networks (ISPN)*, Los Angeles, CA, April 2005.
- [25] E. Shih, P. Bahl, and M. J. Sinclair. Wake on Wireless: An event driven energy saving strategy for battery operated devices. In *Proceedings of the Eighth ACM Conference on Mobile Computing and Networking*, Atlanta, GA, September 2002.
- [26] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins. Turducken: Hierarchical power management for mobile devices. In *Pro-*

*ceedings of The Third International Conference on Mobile Systems, Applications, and Services (MobiSys '05)*, Seattle, WA, June 2005.

- [27] R. Want, T. Pering, G. Danneels, M. Kumar, M. Sundar, and J. Light. The personal server - changing the way we think about ubiquitous computing. In *Proceedings of Ubicomp 2002: 4th International Conference on Ubiquitous Computing*, Goteborg, Sweden, September 2002.
- [28] O. Younis and S. Fahmy. HEED: A hybrid, energy-efficient, distributed clustering approach for ad-hoc sensor networks. *IEEE Transactions on Mobile Computing*, 4(4), October 2004.