Mercury: A Wearable Sensor Network Platform for High-fidelity Motion Analysis

Konrad Lorincz, Bor-rong Chen, Geoffrey Werner Challen,
Atanu Roy Chowdhury, Shyamal Patel*, Paolo Bonato*, and Matt Welsh
School of Engineering and Applied Sciences, Harvard University
*Spaulding Rehabilitation Hospital

Abstract

This paper describes *Mercury*, a wearable, wireless sensor platform for motion analysis of patients being treated for neuromotor disorders, such as Parkinson's Disease, epilepsy, and stroke. In contrast to previous systems intended for short-term use in a laboratory, Mercury is designed to support long-term, longitudinal data collection on patients in hospital and home settings. Patients wear up to 8 wireless nodes equipped with sensors for monitoring movement and physiological conditions. Individual nodes compute high-level features from the raw signals, and a base station performs data collection and tunes sensor node parameters based on energy availability, radio link quality, and application specific policies.

Mercury is designed to overcome the core challenges of long battery lifetime and high data fidelity for long-term studies where patients wear sensors continuously 12 to 18 hours a day. This requires tuning sensor operation and data transfers based on energy consumption of each node and processing data under severe computational constraints. Mercury provides a high-level programming interface that allows a clinical researcher to rapidly build up different policies for driving data collection and tuning sensor lifetime. We present the Mercury architecture and a detailed evaluation of two applications of the system for monitoring patients with Parkinson's Disease and epilepsy.

Categories and Subject Descriptors

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

General Terms

Design

Keywords

Resource-Aware Programming, Wireless Sensor Networks, Mercury

1 Introduction

Wireless sensor networks have the potential to greatly improve the study of diseases that affect motor ability. Small, wearable sensors that measure limb movements, posture, and physiological conditions can yield high-resolution, quantitative data that can be used to better understand the disease and develop more effective treatments. In a typical scenario, a patient wears up to 8 sensors (one on each limb segment) equipped with MEMS accelerometers and gyroscopes. A base station, such as a laptop in the patient's home, collects data from the network. Sophisticated analysis can be performed on the data in order to rate the patient's motor coordination and activity level, which is in turn used to measure the effect of rehabilitation exercise and medications.

The key challenge is tuning the network's operation to achieve high data quality as well as long battery lifetimes. Wearable sensors operate under varying radio link conditions and extremely limited energy budgets. Existing systems have focused on laboratory settings and assume that a clinician is assisting with the data capture and that the patient is always in radio range of the base station. Moreover, previous work has not focused on obtaining long battery lifetimes, since a single clinic visit may only last a matter of hours. It is highly desirable to capture longitudinal data from patients during daily life or during extended stays at the hospital, which pushes the limitations of current sensor platforms.

To support the clinical requirements we must address a number of challenges. Battery lifetime is a paramount concern, in order for the network to operate continuously for up to several days between recharge cycles. This requires careful management of radio communications, flash storage, and data processing on the sensor nodes. Second, the network must tune its operation in the face of variations in radio bandwidth (e.g., as the patient moves around or leaves the home) and energy availability (based on the activity level of each sensor). Third, and most importantly, the system must yield high-quality, clinically-relevant data. The high data rates of the sensors involved (100s of Hz per channel or more) demand on-board processing and storage. It is infeasible to transmit the complete sensor data because it would rapidly deplete the nodes' batteries.

In this paper, we present *Mercury*, a wearable sensor network platform for clinical neuromotor disease studies. The high-level goal of Mercury is to enable a broad range of clinical applications to be deployed on a wearable sensor

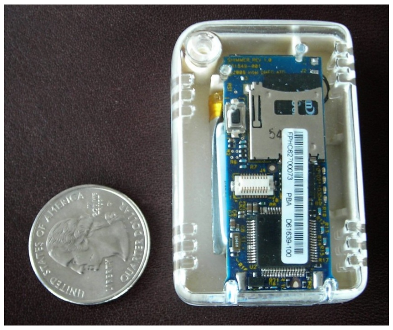

Figure 1: **The SHIMMER wearable sensor platform.**

network, with a focus on acquiring and processing high-resolution signals. Since applications differ in data requirements, Mercury provides a flexible programming interface allowing a range of policies to be implemented on top of the core functionality of the sensor network.

This paper makes the following contributions. First, we describe the Mercury architecture and its techniques for managing sensor energy and optimizing data collection. Second, we provide a detailed energy profile of the SHIMMER [20] wearable sensor platform that motivates the design tradeoffs made in our system. Third, we describe two applications of Mercury, involving monitoring patients with Parkinson's Disease and epileptic seizure detection. Fourth, we present a thorough experimental evaluation of Mercury along several axes including data coverage, battery lifetime, and latency. We present results from a testbed, extensive simulations, and a deployment of the system worn by one of the coauthors. We demonstrate that Mercury achieves our goals of high data quality and long lifetime for high-resolution motion analysis.

The rest of this paper is organized as follows. In the next section, we describe the motivation and background for the Mercury system. In Section 3, we describe the Mercury architecture and application programming interface. We describe two representative applications in detail in Section 4, and present our prototype implementation in Section 5. Section 6 evaluates Mercury across a range of lifetime targets, radio link conditions, and sensor activity levels. Related work is presented in Section 7, and Section 8 describes future work and concludes.

2 Motivation and Background

In recent years there has been increased interest in *body sensor networks* for wearable applications as diverse as elder care [12, 46], emergency response [28, 9], studying athletic performance [1, 32], gait analysis [38, 40, 44], and activity classification [18, 33]. A great deal of work has focused on sensor and hardware design [8, 10, 20, 41], MAC and routing protocols [39, 48], and algorithms for processing wearable sensor data [5, 18]. Our focus in this paper is on the systems challenge of designing a sensor network that can be used for high-fidelity motion analysis studies in patients being treated for neuromotor disabilities including Parkinson's disease, epilepsy, stroke, and Huntington's disease.

As an example, consider a study of patients being treated for Parkinson's disease. The basic approach is to capture triaxial accelerometer and gyroscope data from each limb segment (upper and lower arms and legs) using wearable sensors. The patient wears up to 8 sensors each day and recharges the sensors at night. A laptop in the home serves as a base station to collect and store sensor data. The data is then delivered via an Internet connection to the clinic where it is visualized and further processed by physicians. The signals are subject to extensive processing and data mining to tie the sensor data to clinical measures that evaluate the patient's motor function and progression of the disease [37].

Each sensor samples 6 channels of data at 100 Hz with 16-bit resolution, yielding an aggregate per-node data rate of 1200 bytes/sec. Nodes log raw sensor data to flash. The SHIMMER sensor platform, described below, supports up to 2 GBytes of MicroSD flash, which is enough to record more than 19 days worth of uncompressed data.

A core challenge is how to achieve efficient collection of the sensor data. Although the complete signal can be downloaded from the node's flash (via USB) while the sensors are recharged at night, wireless data collection is highly desirable as clinicians wish to obtain real-time or near real-time observations of the patient's motor activities over the course of each day. A physician might engage with the patient and monitor sensor data during a telephone or video conference in which the patient performs routine exercises that assist with clinical assessment. In the case of epilepsy monitoring, real-time detection of seizures is critical as these events can be life-threatening.

However, wireless data collection is challenged by energy constraints and fluctuations in radio bandwidth as the patient moves about (or leaves) the home. Given the low power radios used on wearable sensor nodes, it is highly unlikely that the *complete* sensor stream can be collected in this way. Even assuming a perfect radio link to the base station, continuous transmission of sensor data would rapidly deplete the sensor nodes' batteries. Therefore, some means of data reduction is needed. Fortunately, in many applications, the first step of clinical data analysis involves *feature extraction* from the raw sensor data [19, 37], which can be performed (at least partially) on the sensor nodes themselves. Generally, these features are much smaller than the original signal, so energy and bandwidth can be conserved by computing and transmitting features instead of the raw data. The stream of features represents a high-value, compressed form of the raw signal.

2.1 Hardware Platform and Energy Profile

Mercury is designed to support an emerging class of low-power, wearable sensor platforms for medical monitoring, such as the SHIMMER [20] platform shown in Figure 1. In this paper we give a detailed description and energy profile of SHIMMER in order to provide a concrete context in which to evaluate our platform. However, the Mercury architecture is not specific to the SHIMMER platform.

SHIMMER includes a TI MSP430 microcontroller; a Chipcon CC2420 radio supporting the 802.15.4 standard; a MicroSD slot supporting up to 2 GB of flash storage; and a 250 mAh Li-polymer rechargeable battery. SHIMMER incorporates triaxial MEMS accelerometer, and add-on daughterboard interface to gyroscope, ECG, EMG, and other sen-

Component	*Energy (µJ)*
Sampling accel	2805
CPU (activity filter)	946
Radio listen (LPL, 4% duty cycle)	2680
Time sync protocol (FTSP)	125
Sampling gyro	53163
Log raw samples to flash	2590
Read raw samples from flash	3413
Transmit raw samples	19958
Compute features	718
Log features to flash	34
Read features to flash	44
Transmit features	249
512-point FFT	12920

Figure 2: **Energy profile of the SHIMMER sensor platform.** *Energy consumption is shown for each major operation. Results are shown for each second of data sampled, processed, or transmitted. Radio energy assumes a perfect radio link to the receiver. Sampling both accelerometer and gyroscope produces 1200 bytes of raw samples per second.*

sor types. The SHIMMER measures $44.5 \times 20 \times 13$ mm and weighs just 10 g, making it well-suited for long-term wearable use.

The need for small size and low weight mandate the use of a small battery, and we anticipate future wearable sensor designs will be further miniaturized. We do not expect energy limitations to go away anytime soon, despite advances in device miniaturization and power. Future reductions in the energy consumption of hardware components would permit even smaller batteries to be used. As an extreme example, implantable wireless sensors will have even more significant power constraints. Therefore, it is vital that we understand the energy characteristics of the hardware platform, which define the operating regime for the Mercury system. Figure 2 summarizes the energy consumption of various hardware components and operations on SHIMMER.

These results motivate a number of design tradeoffs that we must consider. First, the gyroscope consumes a large amount of energy, dominating both flash I/O and radio transmissions. This suggests that we should duty cycle the sensor, for example, by disabling gyroscope sampling when the node is not moving (which can be determined using only the accelerometer). Second, computing certain features on sensor nodes is relatively inexpensive, and the energy savings in terms of reduced flash logging and packet transmissions more than compensate for the increased CPU energy cost. Third, it does not make sense to compute more computationally-demanding features, such as an FFT, on the sensor node itself, due to the high energy cost. The FFT cannot be computed on the fly as sensor data is being sampled.[1] Although it would be possible to compute the FFT as an asynchronous background task, this would require reading raw sample data back from flash, further increasing the energy cost. This cost outstrips the potential savings in transmission overhead.

Using the data from Figure 2, we can estimate the lifetime of the SHIMMER sensor in different modes of operation. With the node continuously sampling and logging accelerometer and gyroscope data, maintaining time sync, but performing no data transfers to the base station, the achievable lifetime with a 250 mAh battery is 12.5 h. Data downloads by the base station impinge further on this energy budget. Figure 3 shows the achievable lifetime for several download scenarios, including the worst-case lifetime of 9.2 h assuming continuous raw sample downloads.

[1]A 512-point FFT across 6 channels of data requires approximately 2.7 sec to compute on the MSP430, which could not keep up with a 100 Hz sampling rate.

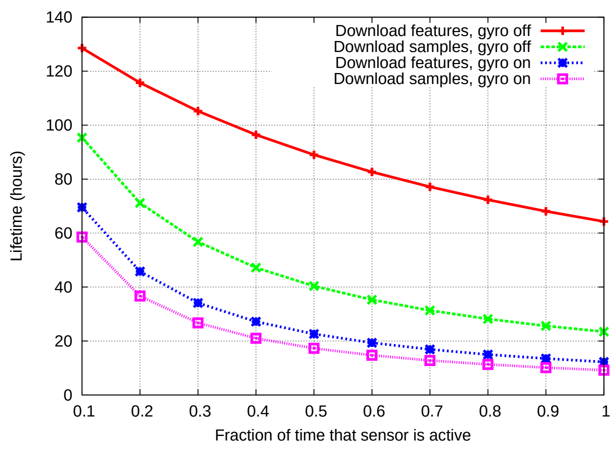

Figure 3: **Estimated lifetime for a SHIMMER under varying activity levels with a 250 mAh battery.** *Separate lines are shown for continuously downloading either samples or features and with gyroscope on or off.*

Adaptive duty cycling of the gyroscope and flash logging significantly reduces baseline energy consumption. A simple *activity filter* algorithm applied to the accelerometer data can determine when the sensor is moving. With the sensor moving 50% of the time, the worst-case lifetime (assuming continuous downloads) jumps to 17 h. Further, if we opt to download features, rather than raw samples, the lifetime can exceed 22 h. Alternately, we could disable the gyroscope altogether, which leads to a lifetime of up to 40 h when downloading raw samples and 89 h when downloading features. However, this severely impacts the fidelity of the data produced by the network, which may or may not be acceptable depending on the application requirements.

3 Mercury Architecture

Mercury provides a platform for the development of wearable sensor applications that must balance battery lifetime and data quality requirements. A typical Mercury network consists of multiple sensor nodes worn by a patient, and a base station, which is typically a laptop with an 802.15.4 transceiver. The Mercury software architecture, shown in Figure 4, is divided into components that run on the sensor nodes and the base station.

Sensor nodes capture and store sensor data, compute features, and respond to requests from the base station to download data and change sampling and storage modes. The core of the application-specific logic resides in the *driver* running on the base station, which can be customized by an end user to support a diverse range of clinical applications. Mercury provides a simple API to the driver to control sensor node operation and retrieve data. In this way, a clinician need not

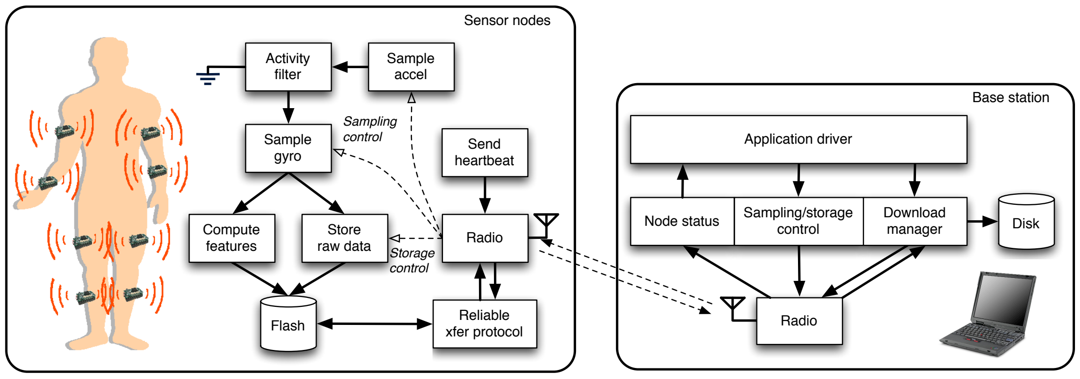

Figure 4: **The Mercury system architecture.**

program the sensor nodes to customize the Mercury network.

The Mercury driver can employ a wide range of policies for tuning data sampling, storage, and downloads to trade off energy consumption and data fidelity. We provide a standard suite of policies for achieving a given battery lifetime target, acquiring high-value data from sensor nodes, and adapting the sensor operation based on activity profile. In addition, Mercury provides an accurate system simulator that permits an end-user to rapidly assess the impact of changes to the driver code on data quality and battery lifetime. This is an essential tool for tuning the application code to meet clinical requirements. In the following sections we detail each aspect of the Mercury architecture.

3.1 Sensor Node Software

In Mercury, sensor nodes are programmed with a single application that performs data sampling, storage, feature computation, and reliable transfers. Sensor nodes provide a narrow interface allowing the base station driver to tune sampling and storage modes, as well as to download previously-stored data. This avoids the need for clinicians to program the sensor node software, but provides adequate control over the network's operation to support diverse application requirements. The sensor node software is implemented using the Pixie operating system [29], which supports a *resource aware* programming model. We exploit Pixie's ability to track energy and bandwidth in real time, as described below.

Sampling and Activity Filter: Under normal operation, a Mercury node samples its various sensors into *sample blocks* each holding a chunk of raw sensor data. Sample blocks are passed to downstream modules for further processing and storage. Sample blocks can contain multiple channels of sensor data. Mercury nodes provide a *sampling control* interface allowing the base station to tune the sampling rate and set of active channels. For example, the sampling mode may specify that the accelerometer and gyroscope are to be sampled at a rate of 100 Hz.

Mercury provides an *activity filter* module that saves energy during periods when a sensor is inactive or otherwise producing uninteresting data. The sensor uses the accelerometer to determine whether the sensor is moving with a simple algorithm that triggers whenever the difference between subsequent values exceeds a threshold. If the sensor is not moving, the filter drops the accelerometer data (so it will not be processed or stored) and disables any other sensors, such as the power-hungry gyroscope sensor. The activity threshold can be tuned by the base station; setting it to zero effectively disables the filter.

When the sensor begins to move, the activity filter begins an *active period*. During active periods, all other sensors that are enabled by the current sampling mode (e.g., the gyroscope) are sampled, and the data is passed downstream to modules for feature extraction and data storage. Recall from Figure 2 that the gyroscope consumes much more energy than the accelerometer, so this approach can significantly reduce energy consumption during inactive periods. This approach is similar to the activity filter in SATIRE [15].

Data Storage: The SHIMMER sensor supports up to 2 GB of MicroSD flash, allowing the node to store the complete raw signal for later retrieval. Each sample block that passes the activity filter is assigned a 32-bit monotonically increasing *block sequence number*. In our current prototype, a sample block is 1200 bytes plus metadata, which is equivalent to 1 sec of sensor data sampled at 100 Hz across 6 channels. Sample blocks are timestamped when the first sample in the block is recorded. Sensor nodes synchronize their local clocks using the FTSP [30] protocol.

Sample blocks are logged to flash as a simple append-only log. Given the large size of the flash we do not expect storage capacity to be an issue, since the flash can store several weeks of data. The block sequence number can be used to determine the location of the block in flash since blocks are constant size. As described below, the base station driver can disable sample block storage on a node in order to save energy. This would permit the sensor data to still be used for feature extraction, although the raw data could not be downloaded later.

Feature Extraction: As discussed earlier, computing features on each sensor node can save considerable bandwidth (and therefore, energy) when retrieving data from the network. Mercury provides a standard suite of feature extractors that are used across a number of applications we have studied. These feature extractors are efficient to compute on sensor nodes. Of course, some applications may require custom feature-extraction algorithms. This logic is contained in its own software component and is straightforward to customize for a given application should the need arise.

Based on previous clinical studies [19, 37], Mercury provides five standard feature extraction algorithms: maximum peak-to-peak amplitude; mean; RMS; peak velocity; and RMS of the jerk time series. These features are efficient to compute on the fly as sensor data is being acquired. Features are computed over multiple overlapping subwindows of the raw signal, which is helpful in reducing aliasing effects. The number and size of the subwindows can be configured by the application developer at compile time. In the applications we have studied, features are computed over ten 5-sec overlapping subwindows over each 30 sec window of sensor data.[2]

This set of features can be computed over 6 channels of accelerometer and gyroscope data in 150 ms for each sample block. Features computed over a window are combined to produce a single *feature block* that is stored to flash. For a 30-sec window size, 5-sec subwindow size, 5 features, and 6 channels, each feature block consumes 600 bytes, not including metadata. As with sample blocks, features are assigned an increasing sequence number used for later retrieval from flash. Note that although a feature block is only half the size of a sample block, features represent 30 sec worth of data, whereas a sample block represents just 1 sec. This is an effective compression ratio of 60:1. Even if we were to compress the raw data and assume a generous compression ratio of 90%, features consume nearly an order of magnitude less bandwidth.

Each feature block includes the range of sample block IDs that correspond to those features. This makes it possible for the driver to download a feature block, inspect its contents, and optionally download the corresponding raw samples if the data is deemed of sufficient value. Note that if raw sample storage is currently disabled by the driver (as described below), this will be an empty range.

Heartbeats: Sensor nodes periodically transmit a *heartbeat* packet to the base station that provides essential information used to schedule data retrieval. The heartbeat includes the number of sample and feature blocks stored on the node, a global timestamp based on the FTSP time synchronization protocol, and status flags used for debugging. Heartbeats are transmitted whenever a new feature block is stored by the node, or every 60 sec if the node is in an inactive state.

Heartbeats include estimates of the node's energy consumption and radio link quality to the base station. Energy use is estimated using the *software energy metering* component built into Pixie [29], which tracks energy use over time using an empirical model that considers the state of each hardware device (CPU, radio, flash, etc.) over time. Previous work [29] has shown that this model achieves up to 98% accuracy. Radio link quality is based on Pixie's bandwidth estimation algorithm which measures the mean transmission delay for each packet to the base station, including ACKs and retransmissions. Based on these radio and energy measurements, the base station is able to carefully schedule data transfers from sensor nodes to achieve lifetime targets and avoid excessive bandwidth use, as described below.

[2]From an energy perspective, it only makes sense to compute features for which the cost of computation and transmission requires less energy than transmitting the raw samples. Due to memory constraints, a node cannot hold 5 sec of raw data in memory. Therefore, features that require more than one pass on the raw data must read the data from flash storage on each pass. As Figure 2 shows, this additional cost could outweigh the cost of transmitting the raw signal.

Operation	Description
lastHeartbeat(n)	Get time last heartbeat received
energyConsumed(n)	Get energy consumed
linkState(n)	Get estimate of radio link
dataAvailable(n)	Get amount of data stored
downloadFeature(n,ids)	Download feature blocks
downloadSample(n,ids)	Download sample blocks
sampleMode(n,rate,chans)	Set sampling rate and channels
storageMode(n,bitmask)	Set storage bitmask
activityThreshold(n,val)	Set activity threshold

Figure 5: **The Mercury driver API.**

Reliable Transfer Protocol: Mercury provides a simple end-to-end reliable transfer protocol similar to Flush [21]. The base station transmits a request containing a sensor node ID, data type (raw data or features), and a range of block addresses that it wishes to retrieve. The node reads the requested data from flash, segments it into radio packets, and transmits each packet (using ARQ) to the base station. Selective NACKs are used to request missing packets.

To save energy, sensor nodes make use of a low-power listening MAC [39] that duty cycles the radio to listen for incoming packets infrequently (we currently use an 4% duty cycle). This requires the base station to transmit a train of packets when sending commands to sensor nodes; however, given that the base station is continuously powered, transmissions to the base station do not incur this overhead. Because the base station schedules data transfers centrally, Mercury minimizes latency and energy consumption by disabling the default CSMA MAC during downloads. Note that this causes control packets (such as heartbeats) to be delayed by an ongoing data download. We limit the duration of a download packet burst to bound this delay.

3.2 Application Driver

The core of the Mercury system is the *application driver*, which runs on the base station and coordinates the sensor network's operation to manage data sampling, storage, and acquisition. The driver is responsible for implementing policies that are appropriate to meet the clinical requirements of the application in terms of data quality and sensor lifetime. Mercury provides a narrow API that permits the driver to inspect the sensor node state, configure sampling and storage modes, and drive data downloads. A wide range of application-specific policies can be implemented on top of this API to target different clinical requirements.

The driver API is shown in Figure 5. Several operations query the state of the sensor nodes based on the most recently received heartbeat from each node. *lastHeartbeat(n)* returns the time that the most recent heartbeat was received from node *n*, allowing the driver to ignore stale information (e.g., when nodes are out of radio range of the base station). The *energyConsumed* and *linkState* queries allow the driver to track the energy consumption and quality of the radio link for each node, which is useful for adapting sensor node behavior and driving downloads.

Downloads are performed using the *downloadFeature* and *downloadSample* operations, which take the node ID and range of feature or sample block IDs as arguments. These are asynchronous calls which queue up the transfer for execution by the Mercury download manager; the driver receives a callback when the transfer is complete. The range of block IDs stored by a node is returned by the *dataAvailable* call. Recall that feature blocks store the range of corresponding sample blocks (if any), so a typical policy is to download the latest feature block, then queue the corresponding sample blocks for download if the features indicate interesting activity.

The driver can tune the sampling rate, set of channels sampled, and whether data is stored by the sensor nodes using the *sampleMode* and *storageMode* calls. These features allow the driver to trade off data quality and energy consumption, depending on clinical requirements. *activityThreshold* tunes the amount of data passed by the sensor activity filter.

4 Application Case Studies

In this section, we present two applications developed for the Mercury architecture: a system for monitoring neuromotor activity of Parkinson's Disease patients, and another for detecting epileptic seizures. Both of these systems are being developed in conjunction with clinical researchers at a local hospital, where Mercury is being used to capture data on several patient populations.

4.1 Parkinson's Disease Monitoring

The Mercury-based Parkinson's Disease (PD) monitoring system is intended to characterize fluctuations in a patient's motor activity over the course of a day. PD patients experience tremor, muscle rigidity, and sluggish movements (bradykinesia) associated with the disease, which can be controlled through medications and other treatments, such as deep brain stimulation. Involuntary movements (dyskinesia) are associated with medication intake, making it necessary to tune the timing and dosage of medications to minimize complications.

In the planned study, patients are monitored at home over the course of several weeks. The clinician is primarily interested in the features computed from the raw signal, however, if the features for a given time window indicate motor fluctuations of interest, the raw sensor data corresponding to those features should be downloaded as well. Typically, the patient will put on the sensors in the morning and take them off to recharge each night. A battery lifetime of at least 24 hours is desirable, although longer lifetimes (48-60 hours) will ensure continuous monitoring if the patient forgets to recharge the sensors every night.

Using the Mercury API, we have developed a range of drivers to support this application. The **standard** driver makes no attempt to save energy, and simply performs round-robin downloads from each sensor node, with a preference for downloading feature blocks before sample blocks. Feature blocks are downloaded in FIFO order, but samples

```
for n in nodes:
 # Screen out disconnected nodes
 if (lastHearbeat(n) > TIMEOUT): continue
 # Screen out nodes without enough energy
 if (energyConsumed(n) > target(n)): continue
 # Screen out nodes with poor radio links
 if (linkState(n) < LINKTHRESH): continue
 # Get list of available feature blocks
 # (fbs) and sample blocks (sbs)
 fbs,sbs = dataAvailable(n)
 if len(fbs) != 0:
    fb = downloadFeature(n, fbs[0])
    # Assign score to corresponding
    # sample blocks
    assignSampleScore(fb)
 elif len(sbs) != 0:
    sbs.sort(scoreSortFn)
    downloadSample(n, sbs[0])
```

Figure 6: **Pseudocode for the throttle downloads driver.**

are downloaded according to a scoring function, as described below.

Recall that raw samples are relatively expensive to download: 30 sec of raw data requires transferring 60 times more data than the features covering the same span. Therefore, the driver prioritizes sample blocks according to an application-defined *scoring function* that represents the clinical value of the data. This ensures that the network will invest its limited resources on acquiring the data with the highest clinical value. When a feature block is downloaded, the features are used to compute the score for the corresponding sample blocks.

The scoring function can take many forms. In our current system, we assign the score of a sample block to be the maximum value of the *peak-to-peak amplitude* feature over all channels and subwindows in the corresponding feature block. This approach prioritizes samples that contain greater range of movement, which are more likely to contain events of interest to a clinician studying Parkinson's Disease. Note that the scoring function can be readily modified simply by changing the driver code on the base station, without reprogramming the sensor nodes.

Throttling Downloads: The standard driver yields clinically-relevant data, but cannot ensure that the nodes meet a target lifetime. We have developed several alternative drivers for this purpose. The **throttle download** driver collects data opportunistically by only performing a download if the node has "excess energy" according to an *energy schedule*. Pseudocode for this driver is shown in Figure 6. Recall that each node's activity filter saves energy by disabling gyro sampling and storage when the node is idle. Therefore, the energy consumption profile of a node will vary over time, based in part on how often it moves.

The energy schedule is computed based on the initial battery capacity C and lifetime target t_l. At a given time t, the node should have at least $\hat{e}(t) = C - (C/t_l \cdot t)$ joules of energy left in its battery in order to satisfy the lifetime target. Given the node's own estimate of its energy consumption $e(t)$, the driver computes the *energy surplus* $s = \hat{e}(t) - e(t)$.

The driver will only attempt a download from the node if $s \geq 0$.

Another feature of the **throttle download** driver is that it will avoid downloading data from a node with a weak radio link. If the node's most recent estimate of its link quality (based on calling *linkState(n)*) indicates a throughput of lower than 30 packets per sec for bulk transfers, we exclude the node from consideration. This saves energy by avoiding a potentially large number of NACKs and retransmissions during the transfer.

Throttling Gyro: As an alternative to limiting data downloads, the **throttle gyro** driver disables the gyroscope sensor on nodes that are running hot according to the energy schedule. Although the activity filter on each node already disables the gyro during inactive periods, turning it off during some *active* periods can save a substantial amount of energy. The tradeoff, of course, is reduced data fidelity. Our clinical colleagues have indicated that this is an acceptable sacrifice during times when the node is unable to meet its lifetime target otherwise. The driver monitors each node's energy state and computes an energy schedule as described above. If energy consumption exceeds the schedule, the node is sent a *samplingMode* command to disable the gyro sensor altogether. If energy consumption falls behind the schedule, the gyro is re-enabled.

Other Policies: It is easy to combine the above policies in various ways. The **throttle gyro and downloads** driver limits both downloads and gyro sampling when energy is limited, getting the combined effect of both techniques. The **throttle storage** driver disables storage for raw samples when energy is constrained, causing the node to only buffer feature blocks. This saves energy for the additional flash writes, but makes the raw sample data unavailable for later download. The **throttle sampling** driver disables all data sampling when energy is limited. This can substantially increase lifetime, but severely impacts data quality.

Each of these drivers is only a few lines of Python code, using the API provided by Mercury for monitoring node state, changing node parameters, and performing downloads. In Section 6 we demonstrate the effectiveness of these policies under a wide range of conditions.

4.2 Epileptic Seizure Detection

The second application involves detection of the onset of epileptic seizures. This system is intended for use in a hospital setting where patients are observed for several days while withholding anticonvulsant medications, thereby permitting seizures to occur. Rapid detection of seizure activity, especially continuous convulsive seizures (status epilepticus), is important since this can be a life-threatening condition.

Detecting whole-body (tonic-clonic) seizures from accelerometer and gyroscope data involves complex analysis that cannot be performed in real time on the sensor nodes themselves [6]. Therefore, if a seizure is suspected, the driver must first download the raw signal from all of the sensors worn by the patient. Of course, continuous raw signal downloads would rapidly deplete the nodes' batteries, so the system must take care to download signals only if there is some preliminary indication of seizure activity. Note that a single node does not have enough information to determine if a seizure is occurring. Therefore, it is necessary to combine information across multiple sensors to discriminate seizures from non-seizure activity.

In this application, it is assumed the patient is generally within radio range of the base station (located in the patient's hospital room or nearby) and therefore we do not expect the patient's sensors to experience disconnectivity or extremely poor radio links for extended periods of time. However, we do assume fluctuations in the radio link. Also, we make no explicit attempt to throttle data sampling or downloads to meet a lifetime target, since low detection latency is the chief concern. Instead, if the battery charge for a node falls below a threshold, the driver sends a notification (e.g., to a nurse) to replace the node.

The driver operates as follows. Feature blocks are downloaded from nodes in a round-robin fashion, as before. Features spanning the same time window are assigned a *score* based on how likely they represent seizure activity. Currently, we use the same scoring function as in the PD system, which measures the motion envelope of the node over a time window. If more than k nodes have a score over a given threshold T, a seizure is suspected, and the driver rapidly downloads the raw sample that span this time window from all nodes. The raw data is subsequently processed at the base station to determine if a seizure is indeed present, and if so, an emergency notification is sent to the hospital staff.

To minimize latency for detection of seizure onsets, the driver assigns a score to each sample block in a manner that causes blocks earlier in the time window following a detection trigger to be downloaded first. This allows the driver to quickly acquire data following a new trigger, even though there may be pending data from previous triggers. Since it is not necessary to acquire the complete raw signal to determine if a seizure is present, this policy prevents spurious triggers from delaying new sample downloads.

The two parameters of interest are the score threshold T and the number of nodes required to trigger a download cycle, k. We keep T fixed at a value that has been shown to capture all possible seizures, but which may also capture non-seizure activity from the patient's normal movements. Setting k to 1 will trigger downloads whenever a single node moves in such a fashion, causing many false positives. Setting k to 8 will only trigger downloads if all nodes indicate a possible seizure, possibly missing some subtle seizure events. In Section 6 we evaluate the accuracy and latency for seizure detection for different settings of this parameter.

4.3 Other Applications

We are exploring a range of other applications based on the Mercury platform. Two specific studies we are planning include monitoring of rehabilitation exercise in patients being treated for stroke and measuring ambulation in patients with chronic obstructive pulmonary disease (COPD). These applications have very different requirements. The COPD study would require long battery lifetimes to monitor patients at home, whereas the stroke study would be conducted primarily in the lab. Also, the COPD study would involve physiological monitoring using ECG and respiration rate sensors, in addition to accelerometers worn on the legs. The Mercury architecture is designed to support a wide range of sensors and application requirements. Mercury's driver API should make it possible to rapidly build up the appropriate network management policies without requiring complex programming of the sensor nodes.

5 Implementation

We have implemented a complete prototype of the Mercury system. The sensor node software is implemented in NesC [16] using the Pixie [29] operating system. We leverage Pixie's dataflow programming model, which maps well onto the structure of this system. Pixie also provides accurate and inexpensive runtime estimation of energy consumption and radio link quality to the base station.

The Mercury base station components are implemented in Python. The application driver is a Python module that invokes the Mercury API shown in Figure 5. We have found that implementing drivers for a range of applications and policies is fairly straightforward, often involving just a few lines of code.

Simulation Environment: Mercury makes it possible to explore a wide range of policies for managing the sensor network's operation. To assist clinicians in understanding the impact of the driver's policy on lifetime and data quality, Mercury provides a detailed simulation environment that accurately models sampling, storage, feature computation, and data transfer behavior of real sensor nodes, as well as the corresponding energy costs. The simulator can be driven by traces of real sensor data acquired separately, or using a range of synthetic trace generators to model varying types of user activity. The simulator requires just seconds to model several hours of sensor node activity, making it appropriate for rapid offline estimation of the impact of a change to the driver code. We have performed extensive validation of the simulation against real sensor nodes, and find that it produces results that are indistinguishable from those produced by the actual sensor motes under similar conditions.

6 Evaluation

In this section, we present a detailed performance evaluation of Mercury along several axes. First, we study the effectiveness of Mercury's energy-saving features. Second, we perform an extensive study of the Parkinson's Disease monitoring system while varying sensor activity levels and lifetime targets. Third, we evaluate the seizure detection system in terms of accuracy, false positive rate, and latency. Finally, we present results from a deployment of the PD monitoring system (worn by one of the coauthors) to demonstrate the complete system being used in a realistic setting.

6.1 Evaluation Methodology

The results in this section were obtained using a combination of live deployment, testbed experiments, and simulation. The testbed setup consists of 8 SHIMMER nodes and a single iMote2 connected to a PC as a base station. The simulation environment is described in Section 5, and we have confirmed that the results are nearly identical to those observed in the testbed. We have also performed extensive measurements of Mercury on live deployments in which we wear the sensor nodes, as described below.

To induce variations in activity level and link quality, each node in the testbed is driven by a *tracefile* stored in ROM

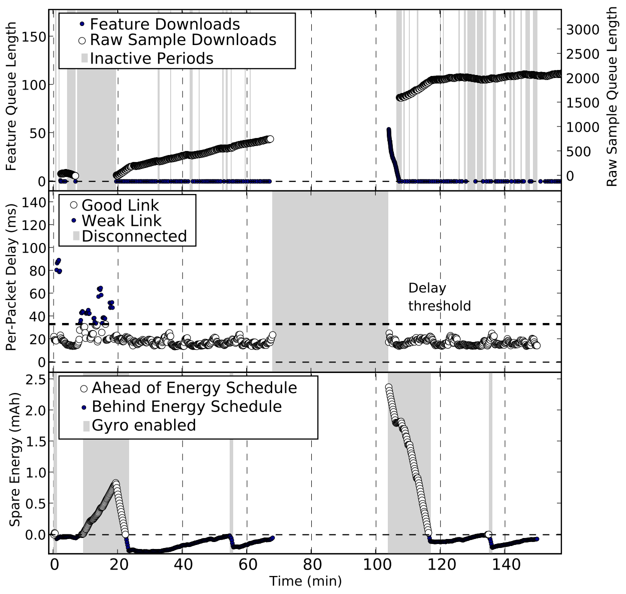

Figure 7: **Detailed view of a single node's behavior.** *This data is from a single node during one of the lab deployments of Mercury. The top figure shows feature and sample queue lengths; the middle figure the radio link quality; and the lower figure the amount of spare energy according to a lifetime target of 24 h.*

that specifies sensor data values and packet radio loss statistics every second. Nodes emulate sensor data acquisition and randomly drop packets according to the tracefile. The simulator models these conditions as well. We model sensor movement as a Poisson arrival process that produces a mean amount of "active" versus "inactive" periods during the course of the run. We vary the activity level of each node independently between 25% to 100%. For the seizure detection system, we additionally model seizure onset as a Poisson process that causes all 8 nodes on the body to experience elevated levels of movement.

We model variations in the radio link quality between sensor nodes and the base station. The radio link varies between lossless, lossy, and disconnected periods. By default, the radio link quality distribution is 25% lossless, 50% lossy, and 25% disconnected. At a given time, the radio link is identical for all nodes, to model conditions that would arise when sensors are worn together on a patient moving in the home.

6.2 Node Dynamics

To get a handle on the complex dynamics of bandwidth and energy use in Mercury, Figure 7 shows a detailed view from an experiment using the Mercury Parkinson's monitoring platform, in which one of the coauthors wore the sensor nodes for over 5 h. We have conducted extensive on-body tests of the complete system spanning more than 13 h. During this experiment, the user performed various activities around the lab (walking, sitting, typing, etc.), and left the lab several times to induce packet loss and disconnection. The figure shows a representative trace from a single node during the experiment, using the **throttle gyro** driver described in

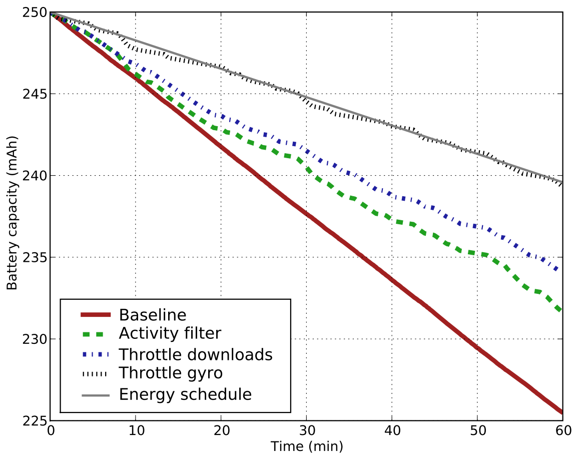

Figure 8: **Benefit of energy saving features in Mercury.** *Each line represents the energy consumption of a node with different energy-saving features enabled. A energy schedule for a 24 hour lifetime target is shown for reference.*

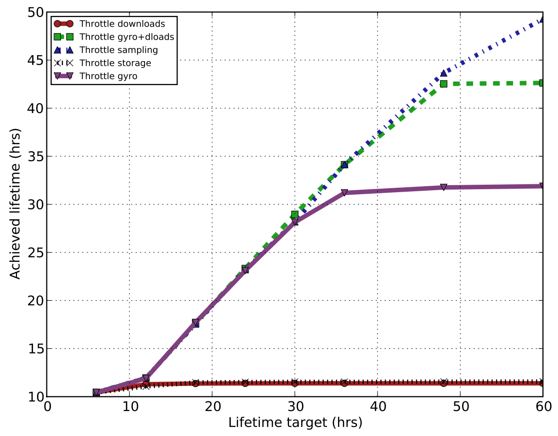

Figure 9: **Achievable lifetime under a range of driver policies.** *Results show the mean across 8 nodes, where all achieved lifetimes are within 14.5% of the mean. This is for a worst-case scenario where the sensors are active at all times. The policy used by the application driver has a significant impact on the maximum battery lifetime.*

Section 4.1. The graph shows activity periods (in which the sensor was moving), times when the node was disconnected from the base station, and times when the gyro was enabled or disabled.

From the figure, we can make the following observations. The driver drains the feature queue whenever there is good radio connectivity, while sample blocks are downloaded less frequently. Due to bandwidth limitations, the driver is unable to download all of the sample blocks stored by the node, explaining the gradual increase in the queue length over time.

The radio link quality (shown as the delay for successful transmission to the base station, including ARQ) varies considerably during the trace. The *delay threshold* shown in the figure is the limit above which the driver will not attempt to download data from the node. Finally, note that the energy profile of the node varies based both on movement and download activity, as expected. The driver toggles the node's gyro on and off depending on energy availability. Note that the spare energy hovers very close to zero, indicating that the system is effective at meeting the lifetime target of 24 h.

6.3 Energy Saving Features

Next, we evaluate the effectiveness of Mercury's energy saving features, including the activity filter, throttling data downloads, and throttling the gyroscope. Figure 8 shows the energy usage profile from a single node in several experiments in which we selectively disabled each feature.

The *baseline* system disables all features, and shows that the node expends energy at a high rate. Enabling the activity filter greatly improves battery lifetime by disabling gyro sampling, feature computation, and data storage when the sensor is not moving. The **throttle downloads** driver saves additional energy, but does not do as well as the **throttle gyro** driver, which is effective at meeting the target lifetime.

Figure 9 shows the achievable lifetime for different driver policies as the target lifetime is increased from 6 to 60 hours. These experiments model a worst-case scenario where sensors are moving at all times. The **throttle downloads** and **throttle storage** drivers can obtain a peak lifetime of around 12 h. **Throttle gyro** does much better, with a peak of around 32 h, while throttling both gyro and downloads achieves nearly 42 h. Throttling both gyro and accelerometer scales almost linearly, but has a negative impact on data quality.

6.4 Data Coverage for Parkinson's Monitoring

The most important metric for the PD monitoring system is the amount of data from active periods that can be retrieved by the network. Therefore, the performance metric we choose for the PD monitoring system is *coverage* as the ratio between the amount of data downloaded by the base station and the amount of data that would have been sampled by an "ideal" network. The (hypothetical) ideal network is not subject to energy limitations and records complete sample blocks during active periods with both the accelerometer and gyroscope sensors enabled. Using an ideal network as the baseline ensures that our analysis is not skewed by the driver's tuning of the node sampling parameters.[3]

We ran a series of experiments using 8 nodes while varying each node's activity level (from 25% to 100%) and the lifetime target (from 6 h to 60 h). Each experiment emulated variable radio link conditions as described above. The data we show in this section is aggregated across 8 SHIMMERs and we use the sum of the amount of data downloaded from 8 nodes to compute coverage. Figure 10 shows results for the **throttle gyro** driver. Recall that this driver prioritizes features over raw samples, so sample coverage is lower than that for features. The volume of sample data simply outstrips the network's download capacity.

[3]Otherwise, a driver that disabled all sampling would achieve a coverage of 100% by fiat, since nodes would sample no data that needed to be downloaded.

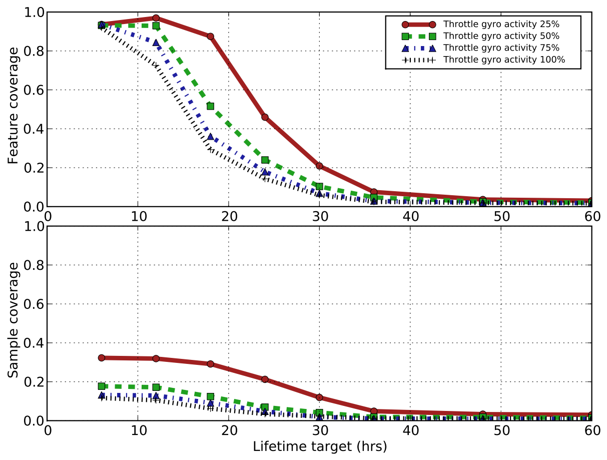

Figure 10: **Impact of varying lifetime target and activity level on data coverage for throttle gyro.** *As lifetime targets increase, the amount of data that can be downloaded from each sensor given the energy budget decreases. Increased sensor activity also effects data coverage. Data is aggregated across 8 nodes.*

One complication to note is that **throttle gyro** dynamically enables and disables the gyroscope sensor. During periods when the gyro is disabled, the data produced by the node contains only accelerometer readings. This degraded data is still valuable for assessing PD motor fluctuations, but has lower quality than with the gyro enabled. We define *full-resolution coverage* as the fraction of blocks downloaded that contain both accelerometer and gyro sensor data, and *degraded-resolution coverage* as the fraction of blocks that only include accelerometer data. In Figure 10, only full-resolution data coverage is shown: we do not give ourselves "credit" for non-gyro data blocks.

We further explore the impact of different driver policies on coverage in Figure 11. These experiments are driven by a trace of real sensor data, and each policy is described in Section 4.1. Policies that tune the gyro have two segments: full-width bars represent full-resolution data coverage, and half-width bars represent degraded-resolution data coverage.

The **throttle gyro** policy maintains degraded feature coverage of 100% across a range of lifetime targets, indicating that it is able to collect a substantial amount of data despite the severe energy constraints. **Throttle storage** maintains high coverage for features, but sample coverage is severely reduced, since this policy disables storage for raw samples when energy is limited. No policy is able to achieve greater than about 25% coverage for sample blocks (when always logging samples), due to limited radio bandwidth. The **throttle downloads** policy falls off rapidly since it is unable to sustain lifetimes longer than about 12 h. The upshot is that toggling the gyro does the best job at meeting long lifetimes while maintaining good (albeit degraded) data coverage.

6.5 Lab Experiment

The aforementioned laboratory experiments (Figure 7) provide us with an opportunity to observe Mercury's behavior in a realistic setting with sensors worn by one of the

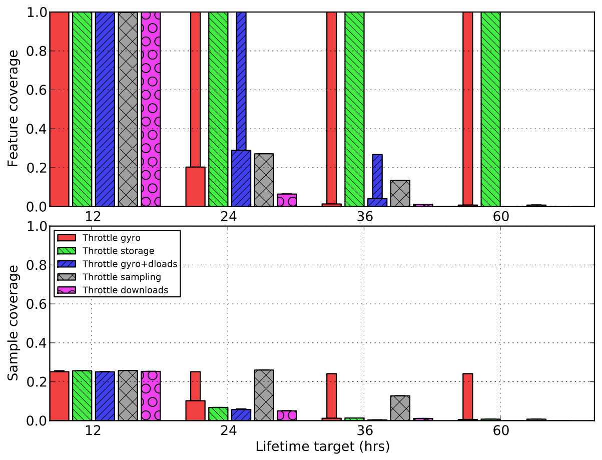

Figure 11: **Data coverage for a range of driver policies.** *In the figure, full-width bars represent coverage for full-resolution data, whereas half-width bars represent coverage for degraded-resolution data (without gyro sampling enabled). Data is aggregated across 8 nodes.*

Node	*Feature coverage*	*Sample coverage*
1 (left upper arm)	97%	34%
2 (left lower arm)	98%	34%
3 (right upper arm)	98%	30%
4 (right lower arm)	96%	30%
5 (left upper leg)	98%	46%
6 (left lower leg)	96%	41%
7 (right upper leg)	91%	39%
8 (right lower leg)	90%	34%
Total	96%	36%

Figure 12: **Summary of data coverage from the lab deployments.**

coauthors. We conducted a series of long-term experiments which yield over 13.5 h worth of data.

During the experiments, nodes exhibited a wide range of activity levels and radio link conditions as we would expect in a real home or hospital setting. Figure 12 summarizes the data coverage for each node; all nodes obtain 90% or greater feature coverage, although the total raw sample coverage is only 36%, owing to limited radio bandwidth. The lower-body nodes exhibit higher coverage for sample blocks, because these nodes were generally less active and therefore stored less data for download. We also observe that the radio link varies across nodes being worn on different parts of the body, as shown in Figure 13. Note that there is not an obvious relationship between the link quality and the sensor's location on the body.

6.6 Epileptic Seizure Detection

Finally, we evaluate the seizure detection system, described in Section 4.2. Since the goal is to provide accurate detections with low latency, we evaluate the system with detection accuracy and latency as metrics. Recall that the application driver operates by downloading feature blocks in round-robin fashion, and triggering a full download of 30 sec of raw samples from all nodes when at least k nodes exhibit a high degree of movement.

Two primary factors affect the performance of this appli-

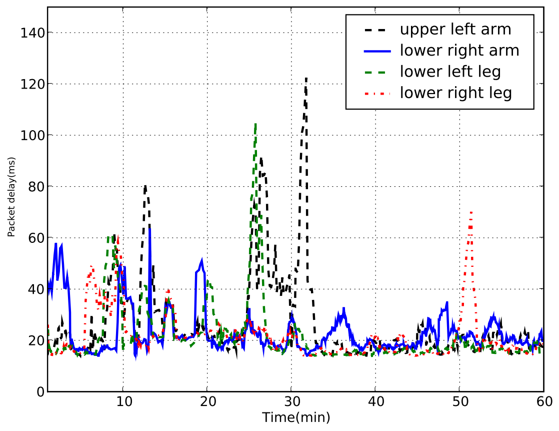

Figure 13: **Radio link quality variation across sensor nodes.** *Measured ARQ packet delays are shown for four nodes in the 8-node lab test. Higher delays indicate lossy links. As the figure shows, there is significant variation in the link conditions for individual nodes.*

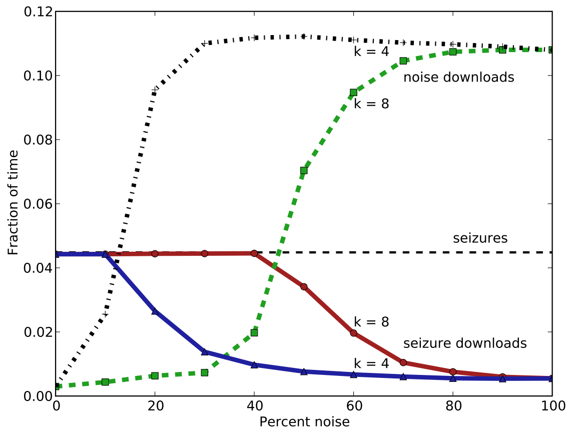

Figure 14: **Impact of varying noise.** *This figure shows the effect of increased noise on the data captured by the network for all 8 nodes.*

cation. The first is the degree of *noise* induced by routine movements by the patient, which can be misconstrued as seizure activity, therefore triggering unnecessary data downloads. The second is the threshold number of nodes k required to trigger a bulk data download. For low values of k, spurious movements can trigger false positive downloads. For large values of k, some subtle seizure events could be missed. To simplify the evaluation, we only consider whole-body (tonic-clonic) seizures that affect all nodes; therefore a threshold $k = 8$ should achieve the best performance.

Varying Movement Noise: We ran a series of experiments with the seizure detection system using 8 nodes sampling accelerometer and gyro data at 100 Hz. In these experiments, seizures are modeled as a Poisson arrival process with an onset probability of 5%. This is, of course, extremely high, but we are interested in the performance of the system under heavy load. We varied the amount of movement noise

k	*True positives*	*False positives*
1	0.34	0.11
2	0.33	0.11
3	0.34	0.11
4	0.34	0.11
5	0.35	0.11
6	0.42	0.11
7	0.83	0.08
8	0.95	0.07

Figure 15: **Impact of detection threshold.** *Data is shown for seizure probability of 5%. Movement noise is based on a trace of real sensor data.*

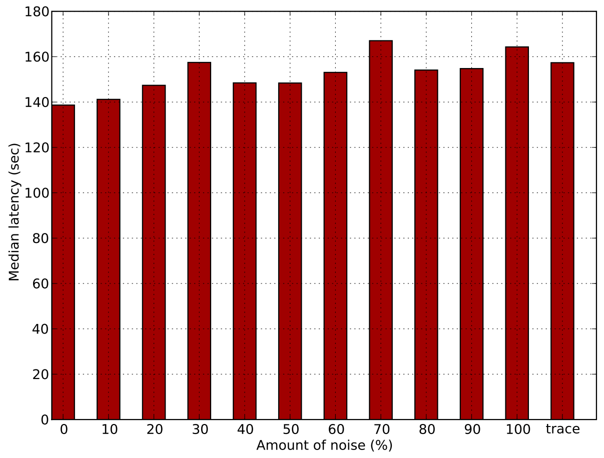

Figure 16: **Seizure detection latency vs. noise.** *This is for a 3 hour run with 8 nodes in which the patient has seizures 10 % of the time.*

occurring independently on each sensor.

Figure 14 shows the total amount of data downloaded from all 8 nodes for seizure periods versus noise periods, with two values of the detection threshold k (4 and 8). For low movement noise, the network downloads 100% of the seizure data. As the noise increases beyond a certain amount, false triggers and limited bandwidth conspire to cause the network to download a large amount of non-seizure data, causing some seizure signals to be lost. Using $k = 8$ performs better, as expected, since more spurious movements are filtered out. On the other hand, this approach would miss certain subtle seizure events. Note that the total volume of sample data downloaded from the network is no greater than 11%, which is due to radio bandwidth limitations.

Varying Detection Threshold: The parameter k controls how many nodes must report high movement before a download cycle is initiated. Figure 15 shows the true positive ratio and false positive ratio for increasing values of k, with seizure onset probability of 5%. Movement noise and radio bandwidth fluctuations are based on a trace captured using real sensors over part of the lab deployment described earlier. For $k = 8$, the system captures 95% of the seizure signals. Due to the high degree of noise in the trace, lower values of k lead to less accuracy. The number of false positives (times when the network downloaded full body data but no seizure was present) is no more than 11%.

Detection Latency: We define *detection latency* as the time between the start of a seizure and the time by which at least 10 sec of raw sample blocks covering the seizure signal have been downloaded from all nodes. The base station can rapidly analyze this raw signal to determine whether a seizure is in fact underway. The detection latency depends on several factors, including the time for the base station to download and analyze feature blocks and the time required for enough raw samples to be downloaded so that we have at least 10 sec of data from each node to analyze.

From a clinical perspective, it is critical that tonic-clonic seizures are detected in less than 5 minutes so that hospital staff can be alerted [11]. Figure 16 shows the detection latency as movement noise increases. Recall that the driver's policy prioritizes raw samples immediately following a trigger, which minimizes the delay arising from spurious download cycles induced by movement noise. The rightmost bar shows the results when a trace of real human activity is used instead of synthetic noise data. As the figure shows, the maximum detection latency is around 170 sec, well below our 300 sec threshold. The primary limiting factor here is the transfer throughput for raw signals from the nodes.

6.7 Preliminary Deployment

Mercury is currently used in several patient studies at Spaulding Rehabilitation Hospital. To date, these deployments have been focused on data collection in a clinical setting. However, our preliminary experiences have yielded tremendous insight into the challenges that we expect to face in a home setting and motivate many of the design choices made in Mercury.

An earlier version of the Mercury platform has been used to collect data on 6 patients over the last several months. Four of these patients are involved in a study of deep brain stimulation (DBS) for Parkinson's Disease. We have undertaken 8 data recording sessions for each patient over a 3-month period, in which each patient wears 9 SHIMMER sensors and performs a series of predetermined tasks while their DBS parameters are adjusted. The sensor data is being used to build predictors of the severity of Parkinsonian symptoms and to gain an understanding of how different DBS parameters affect those symptoms.

The two remaining patients are undergoing treatment for epilepsy. Mercury has been used to capture up to 12 h of accelerometer and electromyogram (EMG) data per day for a 5-day period for each patient. The specific number and placement of the sensors varies depending on the nature of the seizures. The earlier version of Mercury used in these studies assumes a good radio link to the base station and manual data retrieval by the operator. Feedback from the medical users has resulted in a large number of enhancements reflected in our current design to improve ease-of-use, automatic failure recovery, and longer battery lifetimes. We anticipate rolling out the current Mercury software as described in this paper at the hospital, as well as in several patients' homes, in the coming months.

7 Related Work

A number of previous projects have investigated use of wearable sensors for motion analysis, activity classification, and monitoring athletic performance. We briefly describe previous work on wearable sensor platforms, resource adaptivity, and sensor processing algorithms.

Wearable Sensor Platforms: The field of body sensor networks [27] has developed a range of wearable platforms for measuring movement [8, 10, 36] and physiological parameters [28, 46]. One closely related project to Mercury is SATIRE [15], which is designed to identify a user's activity based on accelerometer and GPS sensors integrated into "smart attire" such as a winter jacket. SATIRE is based on a single application (activity classification) and is not intended for clinical applications that require high-fidelity data. Rather, its focus is on broadly classifying movement into a set of "stationary" and "non-stationary" activities.

Compared to the SHIMMER sensors used in our study, SATIRE uses relatively bulky motes using AA batteries for power. Two AA batteries have roughly 10 times the energy capacity of our lightweight Li-poly battery, but with a combined mass of 50 g, 5 times that of the entire SHIMMER package. This solution is not appropriate for fine-grained movement studies where sensors are worn directly on a patient's body and must not interfere with routine activities. Also, minimizing sensor weight is a major concern when dealing with elderly patients. Mercury's requirement of very low-power operation drives many design features of our system not found in SATIRE. SATIRE does not track energy and bandwidth consumption, nor does it adapt its behavior based on resource availability. Also, near-real time data acquisition is not a requirement in their system and therefore not addressed.

LiveNet [43], based on the MIThril [10] wearable architecture, shares many goals with Mercury. The system is based on a PDA worn by the patient connected to individual sensors using wires. Given the choice of hardware, LiveNet devices are much bulkier than the SHIMMER sensor nodes. LiveNet is designed for applications involving a small number of sensors, but is impractical for monitoring limb movements for each body segment. Likewise, MOCA [13] is a low-cost motion capture system based on several accelerometers wired to a PC or PDA. However, the use of multiple wires running to each of the patient's limbs is highly undesirable for long-term wear.

A number of BSN projects have focused on monitoring athletic performance. Examples include assisting professional skiers [32], analyzing tennis serves [1], measuring pitch in baseball [3], and martial arts training [22]. While some of these systems involve high data rates and low-latency data delivery [3], most of them are intended for short-term deployments, such as during training sessions.

Resource Awareness and Fidelity: Adapting data fidelity as a function of resource availability and activity load is a major goal of Mercury. Several projects have looked at one or both of these requirements, mostly outside the wearable domain. Eon [42], Levels [23], and Pixie [29] provide programming primitives for adapting to energy availability in a sensor network. These systems focus on managing resources on a single node, rather than a network. Lance [45] provides a framework for optimizing data retrieval from a sensor network, but focuses exclusively on bulk data transfers, and does not provide methods for tuning sampling or storage operations performed by nodes.

Resource adaptation in mobile and pervasive computing systems has been widely studied, although with an emphasis on supporting traditional applications running on laptops and PDAs. Odyssey [35] supports adaptations to changing network bandwidth [35], energy [14, 26], and computational load [34]. ECOsystem [47] tunes OS scheduling parameters to manage battery life, while Puppeteer [24] supports bandwidth adaptations. Compared to these systems, Mercury is tailored for the needs of high-fidelity motion analysis in a body sensor network, which involves a very different set of design tradeoffs.

BodyQoS [48] proposes a communication layer for body sensor networks that provides quality-of-service guarantees using link estimation and admission control. In contrast, the Mercury driver centrally coordinates download schedules on a per-transfer granularity, yielding control over the type and amount of data acquired from nodes, as well as energy drain.

Algorithms and Classifiers: Much previous work has investigated algorithms and classifiers for extracting information from inertial sensors. These can be broadly categorized as those running on the sensors themselves, or on a back-end device with more substantial processing capacity. The traditional approach has been to sample data on the sensors and deliver it to a back-end for offline processing [2, 4, 31]. However, this approach potentially requires a large amount of data transfer, that does not scale with the limited bandwidth and energy resources in a typical body sensor network. In addition, these systems generally assume that the back-end is always within radio range of the sensors.

To overcome these limitations, a number of algorithms have been developed to run on resource-constrained sensor nodes. These include feature extraction [17, 19, 37], gait analysis [40, 44], fall detection [7, 12, 25], and activity classifiers [5, 15, 18, 33]. These algorithms are largely complementary to Mercury and could be incorporated in the sensor node software. Mercury uses a hybrid of on-node feature extraction coupled with back-end processing on the base station based on a combination of features and raw signals.

8 Future Work and Conclusions

Mercury represents an important step towards longitudinal monitoring of neuromotor diseases in a home setting. Combining wearable, wireless sensors with sophisticated data analysis can greatly improve our understanding of these diseases and the most effective methods of treatment. In this paper, we have presented the Mercury architecture and described techniques for managing energy and radio bandwidth consumption to achieve long lifetimes and high data quality. We are currently finalizing laboratory testing of Mercury with the goal of deploying the system in several patients' homes over the next year.

As future work, we intend to extend the Mercury system to support a wearable base station, such as a cell phone or iMote2, that can be used to collect and process sensor data on the body itself without requiring an external base station. This will require careful balancing of computation and communication within the network to ensure acceptable battery lifetimes. We are also studying additional clinical applications of Mercury in home and hospital settings for monitoring patients being treated for chronic obstructive pulmonary disease and stroke. Experience gained through our work with Parkinson's and epilepsy patients will inform our future developments on the platform.

9 References

[1] A. Ahmadi, D. Rowlands, and D. James. Investigating the translational and rotational motion of the swing using accelerometers for athlete skill assessment. *Sensors, 2006. 5th IEEE Conference on*, pages 980–983, Oct. 2006.

[2] F. R. Allen, E. Ambikairajah, N. H. Lovell, and B. G. Celler. Classification of a known sequence of motions and postures from accelerometry data using adapted gaussian mixture models. *PHYSIOLOGICAL MEASUREMENT*, 2006.

[3] R. Aylward and J. A. Paradiso. A compact, high-speed, wearable sensor network for biomotion capture and interactive media. In *IPSN '07*, New York, NY, USA, 2007. ACM.

[4] J. Boyle, M. Karunanithi, T. Wark, W. Chan, and C. Colavitti. Quantifying functional mobility progress for chronic disease management. *EMBS '06*, pages 5916–5919, 2006.

[5] T. R. Burchfield and S. Venkatesan. Accelerometer-based human abnormal movement detection in wireless sensor networks. In *HealthNet '07*, pages 67–69, New York, NY, 2007.

[6] B. Ceulemans, B. Vanrumste, P. Colleman, S. Omloop, and K. Cuppens. Detection of nocturnal epileptic seizures of pediatric patients using accelerometers. In *Belgian Day on Biomedical Engineering*, 2007.

[7] J. Chen, K. Kwong, D. Chang, J. Luk, and R. Bajcsy. Wearable sensors for reliable fall detection. *IEEE-EMBS'05*, pages 3551–3554, 2005.

[8] 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.

[9] D. W. Curtis, E. J. Pino, J. M. Bailey, E. I. Shih, J. Waterman, S. A. Vinterbo, T. O. Stair, J. V. Guttag, R. A. Greenes, and L. Ohno-Machado. Smart: An integrated wireless system for monitoring unattended patients. *Journal of the American Medical Informatics Association*, 15(1):44–53, 2008.

[10] R. DeVaul, M. Sung, J. Gips, and A. Pentland. Mithril 2003: applications and architecture. *IEEE International Symposium of Wearable Computing*, 2003.

[11] O. Devinsky. Tonic-clonic seizures. `http://www.epilepsy.com/epilepsy/seizure_tonicclonic`, 2004.

[12] C. Doukas and I. Maglogiannis. Advanced patient or elder fall detection based on movement and sound data. *Pervasive-Health'08*, pages 103–107, 30 2008-Feb. 1 2008.

[13] E. Farella, L. Benini, B. Riccò, and A. Acquaviva. Moca: A low-power, low-cost motion capture system based on integrated accelerometers. *Advances in Multimedia*, 2007(82638), 2007.

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

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

[16] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to net-

worked embedded systems. In *Proc. Programming Language Design and Implementation (PLDI)*, June 2003.

[17] 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.

[18] J. He, H. Li, and J. Tan. Real-time daily activity classification with wireless sensor networks using hidden markov model. *Engineering in Medicine and Biology Society, 2007. EMBS 2007. 29th Annual International Conference of the IEEE*, pages 3192–3195, Aug. 2007.

[19] T. Hester, R. Hughes, D. Sherrill, B. Knorr, M. Akay, J. Stein, and P. Bonato. Using wearable sensors to measure motor abilities following stroke. In *BSN '06*, April 2006.

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

[21] S. Kim, R. Fonseca, P. Dutta, A. Tavakoli, D. Culler, P. Levis, S. Shenker, and I. Stoica. Flush: A Reliable Bulk Transport Protocol for Multihop Wireless Networks. In *Proc. SenSys'07*, 2007.

[22] D. Y. Kwon and M. Gross. Combining body sensors and visual sensors for motion training. In *ACE '05: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 94–101, New York, NY, USA, 2005. ACM.

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

[24] 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.

[25] Q. Li, J. A. Stankovic, M. A. Hanson, A. T. Barth, J. Lach, and G. Zhou. Accurate, fast fall detection using gyroscopes and accelerometer-derived posture information. *BSN'09*, pages 138–143, 2009.

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

[27] B. Lo and G.-Z. Yang. Architecture for Body Sensor Networks. In *Perspective in Pervasive Computing*, pages 23–28, September 2005.

[28] K. Lorincz, D. Malan, T. R. F. Fulford-Jones, A. Nawoj, A. Clavel, V. Shnayder, G. Mainland, S. Moulton, and M. Welsh. Sensor Networks for Emergency Response: Challenges and Opportunities. *IEEE Pervasive Computing*, Oct-Dec 2004.

[29] K. Lorincz, B. rong Chen, J. Waterman, G. Werner-Allen, and M. Welsh. Resource aware programming in the pixie os. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 211–224, New York, NY, USA, 2008. ACM.

[30] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi. The flooding time synchronization protocol. In *Second ACM Conference on Embedded Networked Sensor Systems*, November 2004.

[31] M. J. Mathie, B. G. Celler, N. H. Lovell, and A. C. F. Coster. Classification of basic daily movements using a triaxial accelerometer. *Medical and Biological Engineering and Computing*, 2004.

[32] F. Michahelles and B. Schiele. Sensing and monitoring professional skiers. *Pervasive Computing, IEEE*, 2005.

[33] E. Miluzzo, N. D. Lane, K. Fodor, R. Peterson, H. Lu, M. Musolesi, S. B. Eisenman, X. Zheng, and A. T. Campbell. Sensing meets mobile social networks: the design, implementation and evaluation of the cenceme application. In *SenSys '08*, pages 337–350, New York, NY, USA, 2008. ACM.

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

[35] 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.

[36] C. A. Otto, E. Jovanov, and A. Milenkovic. A wban-based system for health monitoring at home. In *Proceedings of the 3rd IEEE EMBS International Summer School and Symposium on Medical Devices and Biosensors (ISSS-MDBS 2006)*, Boston, MA, 2006.

[37] 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.

[38] A. Pentland. Healthwear: medical technology becomes wearable. *Computer*, 37(5):42–49, May 2004.

[39] 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.

[40] A. Salarian, H. Russmann, F. Vingerhoets, C. Dehollain, Y. Blanc, P. Burkhard, and K. Aminian. Gait assessment in parkinson's disease: toward an ambulatory system for long-term monitoring. *Biomedical Engineering, IEEE Transactions on*, 51(8):1434–1443, Aug. 2004.

[41] Sentilla Tmote Sky. `http://www.sentilla.com/pdf/eol/tmote-sky-datasheet.pdf`.

[42] 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.

[43] M. Sung, C. Marci, and A. Pentland. Wearable feedback systems for rehabilitation. *Journal of NeuroEngineering and Rehabilitation*, 2(1):17, 2005.

[44] M. Visintin, H. Barbeau, N. Korner-Bitensky, and N. E. Mayo. A new approach to retrain gait in stroke patients through body weight support and treadmill stimulation. *Stroke*, 29(6):1122–1128, June 1998.

[45] G. Werner-Allen, S. Dawson-Haggerty, and M. Welsh. Lance: optimizing high-resolution signal collection in wireless sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 169–182, New York, NY, USA, 2008. ACM.

[46] A. Wood, G. Virone, T. Doan, Q. Cao, L. Selavo, Y. Wu, L. Fang, Z. He, S. Lin, and J. Stankovic. Alarm-net: Wireless sensor networks for assisted-living and residential monitoring. Technical Report CS-2006-11, University of Virginia, 2006.

[47] H. Zeng, X. Fan, C. S. Ellis, A. Lebeck, and A. Vahdat. ECOSystem: Managing Energy as a First Class Operating System Resource. In *ASPLOS'02*, San Jose, CA, 2002.

[48] 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.