# PowerBench: A Scalable Testbed Infrastructure for Benchmarking Power Consumption

Ivaylo Haratcherev† Gertjan Halkes Tom Parker Otto Visser Koen Langendoen
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology, The Netherlands
{i.j.haratcherev,g.p.halkes,t.e.v.parker,o.w.visser,k.g.langendoen}@tudelft.nl

***Abstract*— The focus of the sensor network community on energy-efficiency has produced a string of novel MAC, routing, and data-aggregation protocols. Their power consumption has mainly been assessed through simulations, i.e. by counting the fraction of time spent in sending, receiving, and computing, and multiplying that by figures taken from data sheets or isolated (single-node) power measurements. In contrast we present PowerBench, a 24-node testbed capable of recording the power consumption of all nodes in *parallel* with a 5 kHz sampling rate and 30 µA resolution. This is accomplished by means of a low-cost interface board featuring a shunt resistor and an A/D converter, whose output is collectively sampled by an embedded Linux platform (Linksys NSLU2).**

**The experience with the PowerBench testbed so far is twofold. First, we have determined that –much to our surprise– timer-based estimations can match true, measured power consumption values within a few percent. Second, we have experienced that a graphical display of the power traces is an effective means to study (and debug) protocol behavior; in particular, inter-node related timing issues can be easily viewed from the state (IDLE/COMPUTE/RX/TX) changes embodied in the power data.**

## I. Introduction

One of the key characteristics that defines the field of Wireless Sensor Networks (WSNs) is the focus on energy consumption, and the efforts in reducing that to near zero levels, in order to reach the goal of being able to deploy large-scale networks built out of small, autonomous devices operating without human assistance. Typical WSN deployment scenarios involve nodes powered by small batteries, so (average) energy consumption must be limited to tens of µW to keep the recurring maintenance costs within economic feasibility when compared to the one-time installation costs of traditional, wired networks. As today's sensor nodes consume in the order of mW when processing and tens of mW when communicating, the WSN research community has looked extensively at the problem of reducing energy consumption.

Common approaches for achieving energy efficiency include duty cycling the radio at the MAC layer and sleep/awake scheduling at the routing/clustering layer. These can be very effective with researchers claiming energy savings of a factor of ten to a hundred –or even more– over conventional protocols like IEEE 802.11 (CSMA/CA). The true gains, however, remain to be determined as most results are based on simulations using coarse energy-consumption models; in the case of MAC protocols for example, one generally maintains statistics about what percentage of time the radio is used for transmitting, receiving, and listening (for incoming traffic) to compute the energy a node consumes by factoring in the current drawn in each particular radio state as specified in the data sheet. A slight improvement is to measure the power consumption of each state using an expensive oscilloscope hooked up to a node performing a ping-pong test. To the best of our knowledge, however, we are not aware of any experiments that verify that single-node consumption rates may be extrapolated to large, multihop networks.

Until recently the WSN community heavily depended on simulation results; only 10 % of the papers mentions the term testbed, with an even smaller fraction actually using one for experimental evaluation. Fortunately, as WSN technology is maturing, the number of experimental testbeds is rapidly expanding, as well as the number of nodes within. Nowadays, testbeds with 100+ nodes are readily available (e.g., MoteLab [13], Kansei [3], TWIST [5]), which provides researchers with a firm grip on the notoriously unpredictable wireless channel. In that respect the Deployment Support Network approach [2] offers the unique possibility to exercise (and debug) protocols in real-world conditions by attaching wireless monitoring nodes instead of running cables to each sensor node. However, regarding the task of determining the energy efficiency of a protocol, none of the testbeds provide support to actually measure the power consumed by each node.

In this paper we present PowerBench, a scalable testbed infrastructure for benchmarking power consumption. PowerBench is centered around a low-cost interface board capturing the power consumption of a target TNOde (a Mica2 clone) in the testbed by means of a shunt resistor and an A/D converter (30 µA resolution). Up to eight interface boards are connected to a modified Linksys NSLU2 device (an embedded Linux platform) sampling the output of the A/D converters in parallel at a rate of 5 kHz. The samples are time stamped and sent out over an Ethernet backbone to a central host storing the power data of the complete testbed. Our current configuration consists of 24 nodes (4 rooms with 6 nodes each), generating a continuous feed of 180 KB/s. After each experimental run the power traces can be graphically displayed for detailed analysis, or processed into a set of per-node statistics (average power consumption, etc). Preliminary experience with the PowerBench testbed includes the surprising observation that

† Supported by NWO (the Dutch National Science Foundation) in the CONSENSUS project.

crude three-level, timer-based estimations can match true, measured power consumption values within a few percent. This important result provides credibility to (unvalidated) estimation-based research performed on other testbeds

## II. Related work

Due to a lack of tools, researchers initially resorted to ad-hoc solutions when quantifying the energy consumption of their algorithms and protocols. The focus was on accounting for the induced wireless communication as the radio is generally the part of a sensor node that consumes most energy. At higher levels in a protocol stack, simply counting the number of message transfers was considered good enough; at lower levels protocols were instrumented to record the time spent in each radio state in order to provide more accurate power-consumption estimates.

$$Energy = \sum_{state\ j} P_{state\ j} \times t_{state\ j} \qquad (1)$$

The level of detail, i.e. the number of states would vary from a crude two (RX/TX), via a reasonable three (RX/TX/SLEEP), to an elaborate six or more when taking radio state transitions into account. Eventually, this approach found its way into mainstream network simulators like NS-2 [7] and GloMoSim/Qualnet [9], as well as WSN-specific simulators like PowerTOSSIM [12], AEON [8], and Prowler [1]. Besides estimating the power consumption of the radio, both PowerTOSSIM and AEON also consider the other hardware components in a sensor node (e.g., CPU, sensors, and LEDs). The resulting level of accuracy was determined to be within 5 % (AEON) and 10 % (PowerTOSSIM) for a number of TinyOS applications running on a single node; AEON has the edge as it is emulating at the instruction level, while PowerTOSSIM performs a discrete event simulation at the basic block level.

With the recent move to experimentation on testbeds, mainly inspired by the lack of realism regarding the simulation models of wireless channels, some testbeds like MoteLab include one node hooked up to a digital multimeter for accurately measuring the power consumption of that node. This single probe provides valuable insight into the actual behavior of an application, but it is hard to expand the scope to the full testbed due to the costs of the measuring equipment. Some extrapolating, however, is possible by feeding the measured power consumption levels back into equation (1) while actually recording at each node the time spend in each state. A large-scale study is desired to determine the validity of this approach, which implies the need for low-cost, but accurate equipment to measure the power consumption of an individual node. We are aware of two dedicated hardware designs addressing this need.

The first design, called SPOT [6], uses the standard current sensing technique of putting a shunt resistor between the supply and the sensor node (MicaZ). The voltage drop across that resistor is amplified and fed to an A/D converter stage. The analog-to-digital conversion is performed with a voltage-to-frequency converter (VFC), and two counters (accumulating time and energy) to provide the large dynamic range of

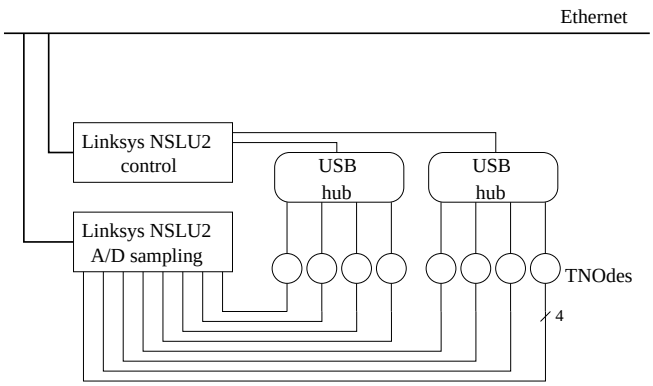


Fig. 1. Testbed building block.

10.000:1 needed for precisely monitoring sleep currents. While this VFC-based method provides good accuracy and high sampling rates, there are some disadvantages. First, it is more costly than a simple single-chip successive approximation ADC solution (see Section III), which makes it less useful for large-scale deployment. Second, as the authors noticed themselves, the oscillator and the VFC can be sources of significant noise in the system, which must be suppressed to avoid interfering with neighboring modules like the wireless sensor node itself. (This requires additional decoupling circuitry and quite expensive metal shielding.) Third, the SPOT design is such that the measurements must be processed by the host sensor node that the SPOT board is attached to. This interferes with normal operation as counters must be read, and their values must be processed on-the-fly, which seriously compromises the usability of this approach.

In contrast to SPOT, the approach by Milenkovic et al. [10] is truly unobtrusive as they use a non-contact current probe (Extech 380946) in combination with a high-end data acquisition card (National Instruments DAQCard-AI-16XE-50). This setup allows for high accuracy and sampling rates, but only at considerable cost limiting the use to a single node only; the authors' claim of using "inexpensive" equipment is not backed up by retail prices of US$ 230 and US$ 1500 for the current probe and acquisition card, respectively.

## III. Testbed design

The primary objective of designing the PowerBench infrastructure was to be able to measure the power consumption of all 24 nodes in our testbed, which translates into the requirement for a low cost solution. In Delft we are using TNOdes, which are similar to the familiar Mica2 nodes, in combination with a separate programmer board powering a TNOde and offering convenient wired access by means of an USB interface for programming and serial I/O. The idea was to re-design the programmer to provide the additional functionality of measuring the power consumption of the TNOde hosted by it. The target was to keep the costs of the additional components (e.g., A/D converter) below €10, roughly 25 % of the total price of a programmer. This price increase would allow us to scale to any number of nodes in our testbed (see Section IV).

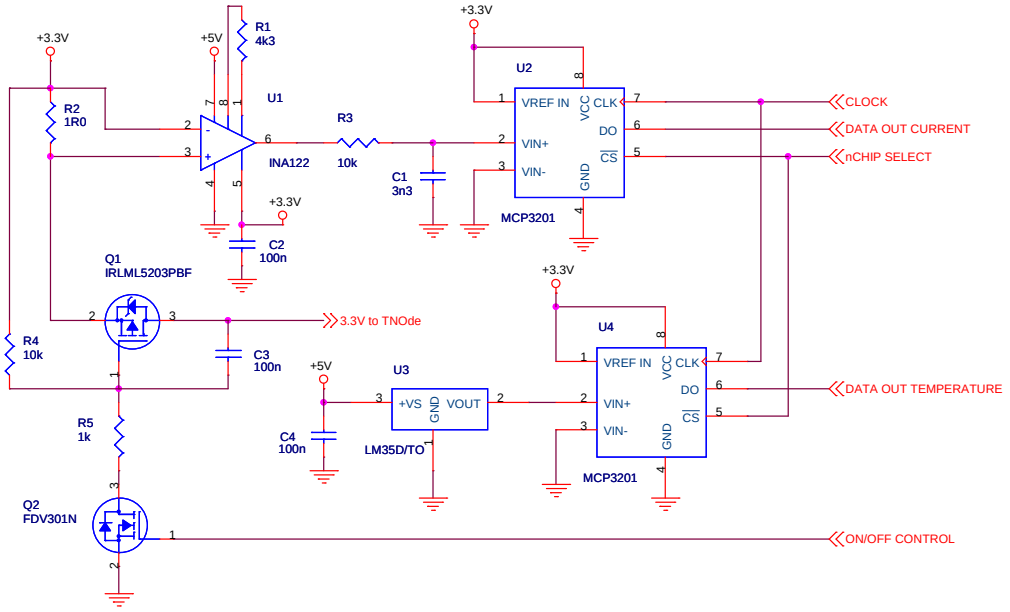


Fig. 2. Schematics of the power measurement module extension.

Further requirements on the testbed infrastructure include a minimum accuracy and time resolution of the power samples, as derived from our objective to study energy-efficient MAC protocols. An effective carrier sense operation with the CC1000 radio takes about 0.5 ms [11], so a sampling rate in the order of 5-10 kHz would suffice to observe these basic (shortest) events. As we want to limit the measurement errors to 5 %, and the base current drawn by an active TNOde is about 5 mA (CPU running idle, radio off), the sampling resolution must be below 250 µA. Capturing the ATmega 128L processor in sleep mode would be nice, but is not essential as the current drawn then drops to 30 µA rendering it insignificant for MAC protocols like B-MAC [11] waking up every couple of hundred milliseconds.

A final set of requirements is derived from the desire to process (view) the power data after it has been captured by the testbed. Such off-line processing requires accurate time-stamping of (batches of) raw power samples, and (some) local processing would reduce the delay between measurement and time-stamping. Also, it is essential that the stream of power data can be aligned with the serial output of the TNOde such that internal protocol events can be used to "explain" particular power profiles. Again, the finest granularity of events is in the order of milliseconds, which translates to a time synchronization in the order of 100 µs.

Figure 1 shows the basic architecture of the PowerBench testbed. The central component is a pair of Linksys NSLU2 devices (a low cost, embedded Linux platform) accessible by Ethernet. One of them is used for controlling the application running on the TNOdes in the testbed; it can be instructed to install (flash) specific program images, start/stop execution, and handles serial I/O. The second Linksys device is dedicated to capturing the raw ADC samples produced by the modified programmer board (see below). Note that we initially planned to use a simpler design featuring only one Linksys device by reusing the USB interface chip on the programmer board to access the A/D converter in parallel with the serial I/O. However, a hardware fault in the USB interface chip forced us to implement dedicated circuitry (wires + Linksys device) for handling the power data.

Figure 2 provides the schematics of our design modifications to the TNOde programmer that allow for external power measurements. We use the standard current measurement method of an ADC sensing the voltage drop across a shunt resistor (R2) placed in series between the supply and the sensor node. To avoid any disturbance in the operation of a programmer and the sensor node hosted by it, we ensure that the voltage drop is small (max 60 mV). This signal must be amplified before it can be read by an ADC. The amplification (of a factor 51.5) is done by an op-amp (U1), which also inverts the amplified signal and conditions it to be referenced to the 3.3 V supply. Then the signal is fed through a low-pass filter consisting of a resistor (R3) and a capacitor (C1). To avoid aliasing effects the cut-off frequency of this filter is about 5 kHz–half of the maximum required sampling rate of 10 kHz– in accordance with Shannon's sampling theorem. For the analog-to-digital conversion we use a simple (cheap) single-chip successive-approximation 12-bit ADC with an SPI interface, made by Microchip (U2). The step size of the ADC (with 3.3 V reference) is about 806 µV corresponding to an ideal current resolution of 16 µA. The full current dynamic range is 65 mA. This range was selected based on the maximum possible power consumption by the sensor node (about 60 mA) and a small safety margin. Note that since the voltage signal is inverted and referenced to the 3.3 V supply, the digital codes read from the ADC have an inverse meaning as well, with `0xFFF` corresponding to zero current.

For temperature-related calibration purposes and for sanity checking of the power measurement data, the programmer design includes a temperature sensor (U3) and ADC (U4) pair. To save on costs, we only mount one temperature sensor per testbed building block (i.e. one per Linksys device). Another

useful feature, supported by our design through transistors Q1 and Q2 and the surrounding passive components, is the ability to cut off the power to the sensor node completely. This allows us to simulate node failures according to some given scenario and study protocol behavior in a controlled manner.

## IV. Implementation details

The basic building block of the PowerBench infrastructure, a pair of Linksys devices (cf. Figure 1), can host up to 8 TNOdes and must be replicated to support larger configurations. For example, the current testbed installed into the ceilings of 4 rooms in our department consists of 24 nodes and 8 Linksys devices (one pair per room). Experiments can be started from any PC connected to the Ethernet backbone, with the power data being streamed back to the PC by the Linksys devices.

The costs of the additional components for a programmer amount to €8.68, which is within the €10 budget. When taking all components of the PowerBench infrastructure (4 modified Linksys devices, 24 programmer modifications, and ADC wirings) into account, the costs total to €640, or €27 per node. This is affordable given the basic price of €78 for a TNOde/programmer pair. Of course some manual labor has to be factored in as well for making the Linksys modifications and special wirings (24h in total, or 1h/node) and for installing the equipment, but as this does not require expert skills it should not be much of a concern.

Besides hardware the PowerBench infrastructure includes various software components to make it all work. There is software for controlling the execution of the application programs on the testbed, software for sampling the A/D converters, and a set of tools to process the recorded power consumption traces. We also developed an extensive debugging interface (using serial I/O) that has proven itself to be very useful. In the remainder of this section we provide additional detail regarding these software components.

### A. Runner software

The software that controls the execution of application programs on the PowerBench testbed was designed to make the life of both user and administrator simple. In particular, the software is self configuring such that adding, removing, and replacing TNOdes and Linksys controllers can be handled automatically, i.e. the user can simply request execution on $N$ nodes without further ado. For experimentation purposes, however, a user can request a specific set of nodes such that a measurement can be repeated with the exact same configuration. To rule out interference by other users executing another application on a different part of the testbed, users can lock the whole testbed for the duration of their experiments.

The software consists of two parts: a *server* program running on each Linksys device and a *runner* program running on the user's own computer. Software updates, which occurred frequently during the development of the testbed, are downloaded from a central repository. Protocol version numbers help to detect incompatible versions of the *server* and *runner* software. The runner program takes a single application image (compiled from TinyOS code for the TNOde target) and distributes that to the Linksys servers involved in the particular run. Each server then checks for all nodes attached to it if they need to participate, and if so inserts the node number (as TOS_LOCAL_ADDRESS) into a copy of the image and flashes that into the node. Next the server waits for a start command from the runner to reset the node and start capturing output (serial data or power traces). The start command is sent out as one broadcast packet over the Ethernet to ensure synchronized execution and data capture across all nodes in the run. All data is streamed back to the runner program at the user's computer, who can then process the raw output. An example of the serial output is shown below:

```
11:1422736: 00 1A 00 1B 00 0D F4 33 00 1C 00 01 1E 2A
00:1432098: 7E 42 7D 5E 00 11 DE 0C 00 00 02 00 18 00 00 00 ...
00:1435625: 7E 42 7D 5E 00 11 DE 0C 00 00 14 00 17 00 00 00 ...
00:1436718: 7E 42 7D 5E 00 11 DE
00:1436878: 0C 00 00 16 00 18 00 00 00 F0 00 17 00 D2 C1 7E
18:1513929: 00 00
```

The format of the data is <node ID:time stamp:data>. Time is measured in ticks of 1/10,000 seconds since the start of the program. The data is a simple byte stream and needs to be interpreted by the user. In the example above we can see a mixture of TOS_Msgs (delimited by `7E` markers) and our own UARTDebug support (lines marked `11` and `18`, see Section IV-D). Note that information may be spread across multiple lines, as is the case with the third TOS_Msg (lines `00:1436718` and `00:1436878`)

### B. Sampling software

We adapted the original Linksys NSLU2 firmware (a modified Linux 2.4.22+ kernel) to match our hardware modifications and include a sampling driver. A major change was the reconfiguration of 7 GPIO pins to interface to the switchboard/wiring hooked up to the 9 ADCs (8 TNOdes + 1 temperature sensor) in a testbed building block (cf. Figure 1). As we operate the Linksys at its peak performance under strong real-time constraints, a number of bugs surfaced in the USB subsystem and kernel time-keeping code that we fixed along the way. In addition we added proper initialization of the CPU GPIO pins and corrected some basic errors in the handling of endianness, all of which resulted in a stable kernel.

Our sampling software, which is part of the *server* program, communicates with the ADCs on the TNOde programmers via the SPI protocol made available in user space. We control the chip enable lines, generate the clock signal, and capture the 12-bit data samples from the 9 ADCs in parallel. Each ADC value is paired with the corresponding node ID, and one timestamp is attached to the collection of 9 readings before it is sent out over the Ethernet to the user who initiated the run. The maximum sampling rate that we can reliably sustain on a Linksys device is 5 kHz, which meets the requirements to be able to observe basic events like a carrier sense operation (taking about 0.5 ms).

### C. Trace processing software

The power data that is received by the *runner* program from the testbed consists of raw ADC values. These values need further processing to be useful. The first step consists

of converting (inverting and scaling) the ADC values to power consumption in mA. However, because of hardware differences the values obtained from each programmer have to be adjusted by a small offset to ensure correct power consumption values. To this end we have calibrated the power consumption of each individual sensor node in the testbed. Once the ADC values have been converted to true power consumption figures, the data can be used to generate plots and calculate total power consumption over the trace (see Section V for sample results).

### D. UARTDebug

Our initial experiences with developing and analyzing MAC protocols on the PowerBench testbed indicated that the standard TinyOS methods of debugging on node hardware (dumping 39 byte TOS_Msg's onto the serial port) were inadequate. Specifically, the most useful data when debugging a MAC protocol tends to be information regarding changes in state variables, with the amount of actual data required being no more than 4 bytes, and we ideally require being able to output as many of these state changes as possible. Wrapping a few bytes of state change in a 39-byte TOS_Msg was an unacceptable level of overhead, especially given the relatively low serial data rates (57k6) on our hardware. We therefore developed the UARTDebug system, a flexible and extensible system for state-change debugging. The core of UARTdebug is in three parts:

- Module-specific *events files*, listing the events that the user wishes to be able to record from a particular TinyOS module, along with information regarding the quantity of data associated with an event (0-4 bytes) e.g. for an update of a counter value, the associated data would be the value of the counter. Parts of this file are generated by a support program.
- A UART interface module running on the sensor node, used for metadata generation and buffering of debug messages.
- A client-side parser program for translating the UART output into human-readable form.

The sections of the events file generated by the support program are also used to provide module-specific #define's to make wiring the debugging module simple, along with enabling simple per-module and system-level enabling/disabling of specific debugging data. A sample output illustrating the startup of a Surge + Multihop application with local modifications for debugging is shown below.

```
00 0.0489 MultiHopLEPSM.Debug DUPLICATE
00 0.0489 MultiHopEngineM.Debug ROUTE_SELECT_FAIL
00 0.0489 MultiHopEngineM.Debug _MSG_RCVD 13 (0x0D)
00 0.0489 MultiHopEngineM.Debug _MSG_ORIGIN 0 (0x00)
00 0.0489 MultiHopEngineM.Debug MSG_LOCAL
00 0.0489 MultiHopEngineM.Debug MSG_FWDING
00 0.0489 MultiHopLEPSM.Debug _SDELTA_NEG 12 (0x0C)
00 0.0489 MultiHopLEPSM.Debug DUPLICATE
00 0.0489 MultiHopEngineM.Debug ROUTE_SELECT_FAIL
00 0.0489 SurgeM.Debug SURGE_ADC_START
00 0.0489 SurgeM.Debug __SURGE_ADC_READ 2 (0x0002)
```

Note that the time seems to be stuck at 0.0489 s, but this is an artefact of our setup in which the timestamps are generated by the *server* program on the Linksys device in combination with buffering on the TNOde's side. The order of events, however, is always correct and the accuracy of time synchronization between Linksys devices (in the order of a few milliseconds) has been good enough to unravel many bugs at various levels in the protocol stack.

## V. Results

After having installed the PowerBench testbed we were anxious to use it to falsify (or validate) the estimation-based approach widely used for assessing the energy efficiency of protocols. First, however, we needed to validate the accuracy of our modified programmer board and software-based sampling method.

### A. PowerBench validation

We started off by performing a careful static calibration of the power measurement subsystem for each programmer. We replaced the normal TNOde with a resistor to precisely control the current to be measured. That current was measured simultaneously by a precise ($\leq$1%) multimeter and our power trace capturing setup. We recorded the difference and incorporated that into a calibration table (one entry per node) that assures compensation for both absolute and proportional errors. (This table is used by the *runner* when processing the raw ADC values.) By using resistors with different values the whole current measurement range (0-65 mA) could be calibrated.

The dynamic accuracy of the setup depends on the one hand on the noise of the op-amp and the ADC, and on the other hand on the bandwidth of the measured signal (the node current) and the acquisition sampling rate. Unfortunately we do not have access to the necessary (and quite expensive) equipment to test the real dynamic measurement accuracy of our setup. However, since we do not work with high sampling rates (5 kHz) and high bandwidth signals ($\leq$5 kHz) we are confident that most of the dynamic errors come from the noise of the ADC and the op-amp. The magnitude of this noise expressed in terms of ADC code difference is about $\pm$1 LSB, resulting in a dynamic resolution of about 30$\mu$A.

### B. Estimation versus measurements

With the calibrated testbed we engaged in the following experiment to determine the accuracy of the estimations derived from tracing the radio states. We modified the core CC1000ControlM module from the TinyOS protocol stack to intercept all state changes of the radio; at every change we read out an internal, high-accuracy timer and update the amount of time spent in the current state (RX/TX/SLEEP). This corresponds to the simple three-state model commonly used to estimate the energy consumption of MAC protocols. The resulting statistics are periodically output over the serial link. We multiply the measured time in each radio state with a combined figure for CPU and radio power consumption for that state. This is because the CPU of the nodes consumes more energy during transmission and reception than during sleep, for two reasons: Firstly, the radio on the nodes is a byte based radio, which means that each byte received or

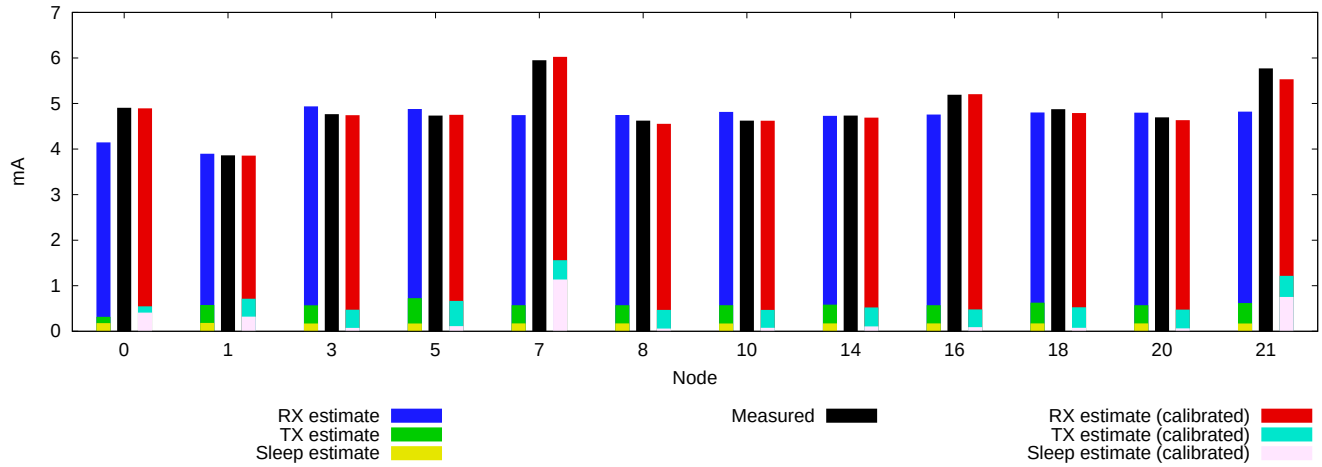


Fig. 3. Estimated power consumption broken down into sources based on average power consumption for each state (first bar) real power consumption (second bar) and estimated power consumption broken down into sources based on per-node calibrated values (third bar) for 12 nodes.

transmitted causes an interrupt on the CPU. Secondly, to receive that interrupt the CPU cannot go to its lowest power mode, but instead has to stay in idle mode.

We used the instrumented CC1000ControlM module to estimate the power consumption of the canonical TinyOS Surge application, with Multihop routing and two different MAC protocols; B-MAC [11] and Crankshaft [4]. At the same time we used the power traces to derive the real power consumption at each of the 24 nodes in the testbed. Figure 3 shows the estimated average power consumption and the measured power consumption for running B-MAC configured to use Low-Power-Listening with a 12 Hz polling rate, i.e. performing a carrier sense every 83 ms. (Note that we only show the results for 12 out of 24 nodes to enhance readability.)

The first bar shown for each node is an estimate based on the *average* power consumption of all nodes for the three radio states. The second bar shows the real power consumption as measured by our testbed. As can be seen from the figure, using the average consumption numbers can lead to significant errors in estimating the power consumption of a node. For example node 7 has an estimated power consumption of 4.75 mA, but a real power consumption of 5.95 mA, an undershoot of 21 %. However, if we use per-node calibrated values for the different states we get far more accurate values, as shown by the third bar. In the example of node 7 it is also clear what is causing the difference between the first estimate and the measured power consumption: node 7 uses a lot more power in sleep mode than is estimated with the average power consumption figures. However, even with the calibrated estimates, some estimates can still be off by several percent for a specific run as shown by the results for node 21 (-5.0 %).

From the estimates that are based on the average consumption figures we can see that nodes have to listen a lot. This is because at the default transmit power setting the network is a two-hop network. This means that when one node is transmitting, most of the others can hear this. Node 0 is the sink collecting all Surge data, and consequently transmits the least; it only has to send routing advertisements. Furthermore, we see that node 5 transmits a little more than most others.

TABLE I

ERROR AS A PERCENTAGE BETWEEN ESTIMATED AND REAL POWER CONSUMPTION FOR THE SURGE APPLICATION WITH DIFFERENT MACS.

| Protocol | Average | Std. dev. |
|---|---|---|
| B-MAC [always on] | -1.62 % | 0.97 |
| B-MAC [12 Hz poll] | -1.03 % | 1.16 |
| B-MAC [5.5 Hz poll] | -1.91 % | 1.44 |
| Crankshaft | -13.48 % | 5.93 |

This is because after some time it will start forwarding messages for the few nodes that cannot directly communicate with the sink (node 0).

For the B-MAC protocol we experimented with different polling frequencies, to see the impact of the radio switching. Table I shows the average error over 2 runs with a total of 46 traces. (The estimations are obtained with per-node calibrated power consumption information.)

Much to our surprise it was quite easy to obtain fairly accurate estimations for the B-MAC protocol; on average the estimations are off by only 2 %. The experiments with polling did show that radio state transitions cannot be ignored completely, with higher errors for more frequent polling. The underlying issue is that during the SLEEP-to-RX state transition of the radio the CPU is kept in a busy-waiting loop for correct timing. The energy consumed during this busy-waiting cannot be ignored. However, by simply modeling the power consumed by the CPU as an additional fraction of time spent in RX mode, we were able to obtain accurate estimations of the total consumed power. This result demonstrates that the standard practice of disregarding the intricacies of radio state transitions when estimating power consumption is valid, for B-MAC that is.

The results for the Crankshaft protocol paint a different picture. Using the radio states to model the energy consumption for Crankshaft leaves us with a significant underestimate of the real power consumption (-13 % on average). The reason for this is the inefficient implementation of the Crankshaft protocol. Crankshaft is a slotted protocol, but nodes do not wake up in each slot (see Section V-C.2 for additional detail).

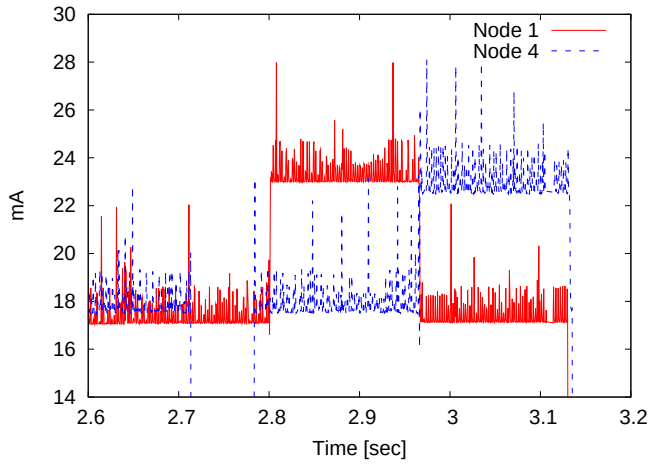


Fig. 4. Power trace of a node (4) running B-MAC switching to transmit mode immediately after receiving a message (from 1). Receiving consumes approximately 18 mA, transmitting 23 mA.

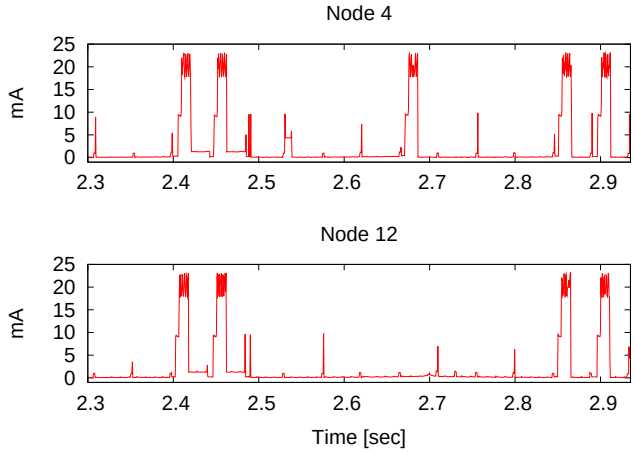


Fig. 5. Power trace for nodes running Crankshaft. Node 4 (top graph) and node 12 (bottom graph) should listen to the same slots. The bottom graph demonstrates the missing power spike at 2.68 seconds which would correspond to turning the radio to receive mode.

However, for ease of implementation the current version of the protocol *does* wake up the CPU at each slot. Waking up the CPU and the subsequent processing also costs energy. Because the Crankshaft protocol keeps the radio turned off for most of the time, this limited CPU use still has a significant impact on the estimate. If the CPU were to only wake up for the slots in which the radio is turned on, the estimate would improve. As a quick fix, we did attempt to model the power consumed by the CPU when the radio was off, but failed to do so accurately due to most processing taking place *inside* interrupt handlers, which is very difficult to account for. This problem was also observed by Landsiedel et al. and explains why AEON (emulator-based) yields more accurate results than PowerTOSSIM (estimation-based) in certain cases [8].

In summary, our results demonstrate that the times the radio spends in different states (RX/TX/SLEEP) can be used to calculate accurate estimates of the real energy consumption, provided the CPU is used sparingly.

### C. Trace visualization

The power traces from the nodes can also be used for debugging purposes. Many artifacts are much easier to see from a visualization than from textual debug output. Although the debug output can be visualized as well, the power traces provide a natural visualization by plotting the consumed power versus the time. We cannot provide quantitative evidence on how much the visualization of the power traces helps in debugging, but we will provide some anecdotal evidence.

*1) B-MAC contention anomaly:* When running Surge with B-MAC as the underlying MAC protocol we observed that in some cases where contention was high, a node would switch to send mode almost immediately after receiving a message (see Figure 4). This seemed to be contrary to the back-off mechanism which B-MAC is advertised to employ. The expected behavior is for the node that has just received a message to wait for a small random amount of time to do a clear channel assessment before starting to send itself. Of course it is possible that the random amount of time is actually zero, but further examination of the trace showed that this behavior occurs quite frequently.

Careful examination of the B-MAC implementation for the Mica2 platform revealed that there is in fact a deviation from the advertised behavior. Rather than picking a new random back-off period after receiving a message, the time remaining from a lost contention is used as the back-off period. Also, when a node starts its contention during a data part of a message sent by a neighboring node, it will count down the back-off timer. These two artifacts have the effect that the back-off period is not uniformly random, but rather skewed towards short periods. Although the quick switching behavior does not seem to have a large impact on the protocol workings, it does increase the chance of collisions. Particularly in high contention situations the chances of collisions are increased because of the skewed distribution of the back-off periods.

Having the power traces from all 24 nodes was invaluable for finding the quick switching behavior. By plotting all traces in a single graph it quickly becomes apparent that something strange is happening. If one were to have the traces from only two nodes it would be likely that there would be a message from another node between the messages from the instrumented nodes, hiding the quick switching behavior. Trying to find this behavior in the debug data or the simulator output is possible, but requires far more effort.

*2) Crankshaft modulo bug:* In the Crankshaft protocol time is divided into frames, and frames are subsequently divided into slots. All frames have the same structure with respect to the slots. The frame starts with a set of unicast slots, followed by one or more broadcast slots. The number of unicast and broadcast slots is fixed, but can be changed at compile time. A node has to listen to each broadcast slot, and to one unicast slot. The unicast slot a node has to listen to is determined by taking the MAC address modulo the number of unicast slots. For example, when there are 8 unicast slots, a node with MAC address 4 will listen in the fourth unicast slot, as will a node with MAC address 12.

During implementation of the MAC protocol, it was initially

decided that the number of unicast slots would be 16. However, later on during the implementation process the number of unicast slots was made a compile time constant. The code was changed for determining when a neighboring node would be awake, but the code to determine in which slot to listen was accidentally left untouched.

The result of this oversight can be seen in Figure 5. The two graphs show the power traces for node 4 and 12, with the number of unicast slots set to eight and the number of broadcast slots set to two. One can clearly see the two consecutive spikes in power consumption for the broadcast slots for two consecutive frames, where the radio was set to receive mode (at 2.40, 2.45, 2.85 and 2.9 seconds). For node 4 we can also see another spike between the two sets of broadcast slots at 2.68 seconds, which corresponds to unicast slot 4. In the bottom graph, which shows the power trace for node 12, the spike in the middle is missing.

Of course the modulo bug could also have been found either by simulation or by inspecting the debug output from the serial port. However, detecting that the start of slot marker is missing from the the simulator output or the debug output is made difficult by all the other messages. In the visualization it is immediately clear. Although for this case it is not necessary to have all 24 traces available, it is necessary to have the trace of a node with a MAC address in a specific range. If the traces of a limited set of nodes were available, there would be a good chance that none of these nodes had appropriate MAC addresses, and the bug would have been missed.

## VI. Conclusions

In this paper we have presented PowerBench, a scalable testbed infrastructure for benchmarking power consumption. PowerBench includes hardware components as well as software for capturing the power traces of all nodes in the testbed in parallel. These traces are stored for off-line processing and viewing, aiding in the analysis and debugging of the energy footprint of various protocols. PowerBench was developed for the TNOde architecture, which is similar to the familiar Mica2, and is validated to accurately measure power consumption with a 30 µA resolution at a sampling rate of 5 kHz. This is precise enough to trace events at the lowest level in the protocol stack, for example B-MAC performing a (periodic) carrier sense and then going to sleep immediately.

The software developed for the PowerBench testbed includes programs to run applications on the nodes, collect the resulting power traces, and to visualize them or to compute the average power consumption. The PowerBench hardware consists of a modified programmer, measuring the power consumption by means of a shunt resistor, op-amp, and ADC, and a set of Linksys devices monitoring the ADCs (8 nodes per device). The costs of the additional components amount to €8.68 per programmer; when factoring in all components of the PowerBench infrastructure the costs total to €27 per node, or 34 % of a node/programmer pair.

We have installed a 24-node testbed in our labs and used it for two purposes. First, we have compared the traditional method of estimating energy consumption –based on counting the time spent in each radio state– with the true measurements provided by PowerBench. Much to our surprise the estimates can be tuned (calibrated) to perfection with errors generally below 2 %, provided the CPU is used sparingly. Second, we used it to study/develop MAC protocols and provided examples of anomalies detected through visualizing the power traces obtained on the PowerBench testbed.

## References


[1] A. Barberis, L. Barboni, and M. Valle. Evaluating energy consumption in wireless sensor networks applications. In *10th Euromicro Conf. on Digital System Design Architectures, Methods and Tools*, pp 455–462, Lübeck, Germany, Aug. 2007.

[2] M. Dyer, J. Beutel, L. Thiele, T. Kalt, P. Oehen, K. Martin, and P. Blum. Deployment support network - a toolkit for the development of WSNs. In *4th European conference on Wireless Sensor Networks (EWSN'07)*, pp 195–21, Delft, The Netherlands, Jan. 2007.

[3] E. Ertin, A. Arora, R. Ramnath, V. Naik, S. Bapat, V. Kulathumani, M. Sridharan, H. Zhang, H. Cao, and M. Nesterenko. Kansei: A testbed for sensing at scale. In *Fifth Int. Conference on Information Processing in Sensor Networks (IPSN/SPOTS)*, pp 399–406, Nashville, TN, Apr. 2006.

[4] G. Halkes and K. Langendoen. Crankshaft: An energy-efficient MAC-protocol for dense wireless sensor networks. In *4th European conference on Wireless Sensor Networks (EWSN'07)*, pp 228–244, Delft, The Netherlands, Jan. 2007.

[5] V. Handziski, A. Köpke, A. Willig, and A. Wolisz. TWIST: a scalable and reconfigurable testbed for wireless indoor experiments with sensor networks. In *2nd Int. Workshop on Multi-hop Ad Hoc Networks: from Theory to Reality (REALMAN '06)*, pp 63–70, Florence, Italy, May 2006.

[6] X. Jiang, P. Dutta, D. Culler, and I. Stoica. Micro power meter for energy monitoring of wireless sensor networks at scale. In *Sixth Int. Conference on Information Processing in Sensor Networks (IPSN/SPOTS)*, pp 186–195, Cambridge, MA, Apr. 2007.

[7] V. Kakadia. Energy model update in ns-2. `http://www.isi.edu/ilense/software/smac/ns2_energy.html`, 2005.

[8] O. Landsiedel, K. Wehrle, and S. Götz. Accurate prediction of power consumption in sensor networks. In *IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*, pp 37–44, Sydney, Australia, May 2005.

[9] C. Margi and K. Obraczka. Instrumenting network simulators for evaluating energy consumption in power-aware ad-hoc network protocols. In *12th Int. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pp 337–346, Volendam, The Netherlands, Oct. 2004.

[10] A. Milenkovic, M. Milenkovic, E. Jovanov, D. Hite, and D. Raskovic. An environment for runtime power monitoring of wireless sensor network platforms. In *37th Southeastern Symposium on System Theory (SSST)*, pp 406–410, Tuskegee, AL, Mar. 2005.

[11] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *2nd ACM Conf. on Embedded Networked Sensor Systems (SenSys 2004)*, pp 95–107, Baltimore, MD, Nov. 2004.

[12] V. Shnayder, M. Hempstead, B. Chen, G. Werner-Allen, and M. Welsh. Simulating the power consumption of largescale sensor network applications. In *2nd ACM Conf. on Embedded Networked Sensor Systems (SenSys 2004)*, pp 188–200, Baltimore, MD, Nov. 2004.

[13] G. Werner-Allen, P. Swieskowski, and M. Welsh. MoteLab: A wireless sensor network testbed. In *Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS) associated with IPSN'05*, Los Angeles, CA, Apr. 2005.