Practical Exploitation of the Energy-Latency Tradeoff for Sensor Network Broadcast

Technical Report

August 2006

Matthew J. Miller
Department of Computer Science, and
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
mjmille2@uiuc.edu

Indranil Gupta
Department of Computer Science
University of Illinois at Urbana-Champaign
indy@cs.uiuc.edu

***Abstract*— As devices become more reliant on battery power, it is essential to design energy efficient protocols. While there is a vast amount of research into power save protocols for unicast traffic, relatively little attention has been given to broadcast traffic. In previous work [1], we proposed Probability-Based Broadcast Forwarding (PBBF) to address broadcast power save by allowing users to select a desired tradeoff between energy consumption, latency, and reliability. In this paper we extend our previous work in two ways. First, we introduce a new parameter that allows a tradeoff between reliability and packet overhead to give users more options. Second, we implement PBBF on the TinyOS platform [2] to evaluate it beyond the analysis and simulation from our previous work. Our evaluation demonstrates the tradeoffs possible using PBBF on sensor hardware.**

I. Introduction

The relatively small improvement in battery energy density recently [3] necessitates the need for energy efficient protocols to control the *rate* at which energy is being depleted. To this end, many power save protocols have been proposed to increase the time that a device's radio is sleeping while still providing an acceptable latency and throughput. Work in this domain focuses almost exclusively on unicast traffic. Our previous work on Probability-Based Broadcast Forwarding (PBBF) [1] was the first to explore the energy-latency tradeoff for broadcast traffic. Multihop broadcast is used in many wireless network applications. Some common uses of multihop broadcast include discovering routing paths, sinks querying sensors for data, and distributing code updates throughout the network.

With respect to broadcast, power save protocols generally expose two options to the user. First, if no power save is used, then the broadcast can achieve a relatively low latency, but at the expense of large energy costs to listen for broadcasts. The second option is to use the power save protocol. This option conserves much less energy than the first, but has a high latency that may be unacceptable to some applications.

In previous our work [1], we proposed a lightweight protocol to augment existing protocols that allows broadcast propagation to be more energy efficient while still achieving a desired latency. In this paper, we extend that work in two ways:

- *Introduce a parameter to control the reliability-overhead tradeoff:* Previously [1], we proposed two parameters (discussed in Section III) that present tradeoffs in energy consumption, latency, and reliability. By introducing a third parameter, described in Section IV, we allow another tradeoff in reliability and packet overhead.
- *Implement PBBF in TinyOS [2] on top of B-MAC [4]:* Our previous work explored PBBF via analysis and simulation. In this work, we implement the protocol in TinyOS [2], as described in Section V, and evaluate the performance in Section VI. Also, our previous work demonstrated PBBF on top of 802.11 Power Save Mode (PSM) [5]; in this work it is implemented on top of B-MAC [4] to demonstrate PBBF's versatility.

In Section II we survey some related work in this area. We review our previous work in Section III and then present a new extension in Section IV. Section V discusses our TinyOS [2] implementation. We evaluate the PBBF implementation in Section VI and survey lessons learned in Section VII. Section VIII concludes our paper.

II. Related Work

There have been many power save techniques proposed for unicast traffic. In [6], the authors survey many of these protocols. Our work [1] was the first to study power save in the context of broadcast traffic by proposing PBBF which is detailed in Section III. PBBF uses probability-based forwarding for energy efficiency. While PBBF was the first to propose this for power save, other protocols have used probabilistic broadcast forwarding for other reasons. Most notably, Haas et al. [7] designed a protocol where nodes only forward a broadcast with some probability, p. By doing this, the broadcast is capable of reaching most of the nodes in the network while reducing the overhead. This is based on the observation that a broadcast flood typically has a high level of

This work is supported in part by a NSF fellowship.

redundancy [8]. With PBBF, we try to use this redundancy to reduce the *energy* consumed by the broadcast.

TinyOS [2] is an operating system designed at Berkeley specifically for sensors. It favors simplicity and clean design by using a single-thread of execution and a component-based, modular architecture. B-MAC [4] is TinyOS's default power save protocol. In Section V, we integrate PBBF on top of B-MAC. B-MAC uses preamble sampling which means that the packet preamble is long enough to be detected by all nodes that are periodically sampling the channel in between sleep periods (i.e., the preamble must be slightly longer than the sleep time between sampling periods). When sleeping nodes sample the channel and detect the preamble, they remain on to receive the entire packet. See [4] for more details.

III. Probability-Based Broadcast Forwarding [1]

In this section, we review our previous work. Probability-Based Broadcast Forwarding (PBBF) can be used in conjunction with any power save protocol that has the following characteristics:

1) Nodes are scheduled to sleep at certain times and can be awakened on-demand when a neighbor wishes to communicate.
2) A mechanism exists which ensures that all of a node's neighbors will be awake at the same time to receive a broadcast.

In [1], we use IEEE 802.11 PSM [5] as the base protocol to demonstrate PBBF and in Section V, we use B-MAC [4] as the base protocol. The goal of PBBF is to achieve a specified reliability, with high probability, while allowing a wide-range of tradeoffs in energy consumption and latency. Specifically, we focus on two definitions of reliability in this work: (1) the average fraction of nodes that receive a broadcast and (2) the average fraction of broadcasts received by a node.

PBBF introduces two new parameters to a power save protocol: p and q. The first parameter, p, is the probability that a node rebroadcasts a packet in the current active time even though not all neighbors may be awake to receive the broadcast. With probability $(1-p)$, the node will wait to send the packet according to the power save protocol. The second parameter, q, represents the probability that a node remains on after the active time when it normally would sleep (the length of time that a node remains on is a parameter of the power save protocol being used). With probability $(1-q)$, the node sleeps as it would in the original power save protocol. Even with these modifications, a node still only rebroadcasts a packet once. In Section IV, we introduce a third parameter that allows a node to rebroadcast a packet twice for added reliability.

Figure 1 shows pseudo-code of changes to any sleep scheduling protocol required for PBBF. The original sleep scheduling protocol is a special case of PBBF with $p = 0$ and $q = 0$. The *always-on* mode (i.e., no active-sleep cycles) can be approximated by setting $p = 1$ and $q = 1$. PBBF may be slightly different from *always-on* in this case. For example, in synchronous protocols, there may still be byte overhead (e.g., sending advertisements) and temporal overhead (i.e., PBBF cannot send data packets during the advertisement window).

```
SLEEP-DECISION-HANDLER()
  /* Called at the end of active time */
  /* If stayOn is true, then remain on; else sleep*/
  stayOn ← false

  if DataToSend = true or DataToRecv = true
    then
         stayOn ← true
    else if UNIFORM-RAND(0, 1) < q
            then stayOn ← true


RECEIVE-BROADCAST(pkt)
  /* Called when broadcast packet pkt is received */
  if UNIFORM-RAND(0, 1) < p
    then SEND-BROADCAST(pkt)
    else ENQUEUE(nextPktQueue, pkt)
```

Fig. 1. Pseudo-code for PBBF.

Intuitively, we can see that the p and q parameters will have the following effects.

Energy: As q increases, energy consumption increases. Changing p has a negligible effect on energy consumption.

Latency: As q increases, latency decreases, provided that $p > 0$. As p increases, latency decreases, provided that $q > 0$.

Reliability: As q increases, reliability increases, provided that $p > 0$. As p increases, reliability decreases, provided that $q < 1$. When p increases, there is a greater probability that a node rebroadcasts the packet immediately. Thus, for a fixed $q < 1$, there is a greater chance that some of its neighbors do not receive the broadcast since they chose to sleep.

If the conditions listed above (e.g., $p > 0$ for latency and reliability as q increases) are not met, then the metric is not affected in that situation.

IV. PBBF Extension

As mentioned in Section III, the PBBF parameters p and q provide a tradeoff in energy consumption, latency, and reliability for broadcast dissemination. Now, we propose another parameter that can be used in PBBF that induces an overhead tradeoff in addition to the three aforementioned metrics (i.e., energy consumption, latency, and reliability). We denote this parameter as r and it behaves as follows. When a sensor decides to immediately transmit a broadcast packet according the p parameter (as described in Section III), it will broadcast the packet a second time with probability r. If the packet is broadcast for a second time, then the second transmission is advertised according to the sleep scheduling protocol's original protocol. The pseudo-code for this PBBF extension is shown in Figure 2.

We can see that the r parameter induces an overhead tradeoff into PBBF. By increasing r, we increase the reliability

```
RECEIVE-BROADCAST(pkt)
  /* Called when broadcast packet pkt is received */
  if UNIFORM-RAND(0, 1) < p
    then SEND-BROADCAST(pkt)
         if UNIFORM-RAND(0, 1) < r
           then ENQUEUE(nextPktQueue, pkt)
    else ENQUEUE(nextPktQueue, pkt)
```

Fig. 2. Pseudo-code for r parameter in PBBF.

of a broadcast at the expense of increasing the packet overhead in the network. At the extreme, if $r = 1$, then reliability should be close to 100% regardless of the p and q values, but each node is broadcasting every packet twice. This gives users yet another control parameter to achieve a desired tradeoff in the energy consumption, latency, reliability, and overhead planes.

V. IMPLEMENTATION

We implemented PBBF in TinyOS 1.1.15 [2] for the Mica2 Mote [9] sensors. This serves as a proof-of-concept for the protocol and provides results from a real-world communication environment. PBBF is implemented on top of a different sleep scheduling protocol than the 802.11 PSM protocol that was the basis for the simulations in [1]. This demonstrates the versatility of PBBF. Additionally, we added the extension to the PBBF protocol described in Section IV.

We chose to implement PBBF in TinyOS [2] since this is a widely used open-source operating system designed for sensors. Its adoption in the research community has led to a relatively stable system with a significant amount of documentation. For a hardware platform, the Mica2 [9] and Telos [10] Motes were available. We chose to use the Mica2 platform since it has two power save protocols implemented for it.

The two power save protocols available on the Mica2 platform were S-MAC [11] and B-MAC [4]. Either would have been an appropriate choice for our PBBF implementation. We chose to use B-MAC over S-MAC for several reasons. First, B-MAC is implemented in the core of TinyOS whereas S-MAC is an add-on that must be incorporated into the TinyOS separately. Thus, B-MAC has undergone more rigorous testing since it is used by nearly everyone that downloads TinyOS and does not require an extra effort to get it working. Second, the code for B-MAC was less complex and easier to understand. Thus, it was easier to make the necessary modifications for PBBF. Finally, B-MAC, unlike S-MAC, does not require time synchronization. This eliminates a major source of potential experimental errors.

As described in Section II, B-MAC uses preamble sampling for in-band power saving. Sensors wake up according to a specified duty cycle and carrier sense the channel. If the channel is idle, the sensor returns to sleep until the next scheduled carrier sense period. If the channel is busy, the sensor continues listening to channel in anticipation of receiving a pending data packet. When a node has data to transmit, it attaches a preamble longer than the duty cycle in order to guarantee that all nodes will carrier sense the channel at some point during the preamble and continue listening.

To implement PBBF on B-MAC, we make the following changes:

- When a node carrier senses the channel idle during its duty cycle, with probability q, it continues listening to the channel until its next scheduled carrier sensing period.
- When a node has a packet to rebroadcast, with probability p, it transmits the packet without the long preamble. In this situation, most of the node's neighbors will not carrier sense the preamble and, hence, not receive the broadcast packet at that time. With probability $(1-p)$, the node will rebroadcast the packet with the long preamble so that its neighbors will carrier sense it and receive the subsequent data packet.
- When a node rebroadcasts the packet *without* the long preamble (as discussed in the previous item above), with probability r, it will broadcast the packet a second time. This second broadcast will use the long preamble.

The architecture we used for our implementation is shown in Figure 3. The solid arrows in the figure represent the interface that connects two modules. The notation $A \xrightarrow{I} B$ indicates that component B implements interface I and that component A uses B's implementation of interface I. The dashed arrows indicate the message type that the connected module uses to send and/or receive via `GenericComm`. Details about the interfaces and packet types are in Appendix I. The `GenericComm`, `UART`, and `CC1000Radio`[1] components are already implemented in TinyOS. We made some modifications to the CC1000Radio modules, but used these components, for the most part, in their current TinyOS instantiation. We now describe the functionality of each component from Figure 3.

DummyBcastSrc: This is the application that we use to test PBBF. The node with ID 0 is chosen as the broadcast source and transmits a broadcast periodically according to a desired rate. The broadcast does not contain any useful data. Non-source nodes that receive broadcast packets pass information to the `Stats` module to collect experimental data.

`DummyBcastSrc` also serves as the link to the serial port (UART) for communication with a computer. The module also passes control packets (e.g., what p, q, and r parameters to use for a particular run) to `CtrlPktHandler`. Finally, it maintains all the timers for when an experimental run ends and when statistics are sent back to the broadcast source.

SimplePbbfBcast: The main functions of this module are duplicate suppression and queuing for `DummyBcastSrc`'s broadcast packets. Control packets are also passed to `CtrlPktHandler` and `Stats` is notified of *every* broadcast sent or received.

CtrlPktHandler: This module handles incoming control packets by setting the p, q, and r parameters to the values

[1] `CC1000Radio` is an abstraction for the three modules listed in the dotted lines.

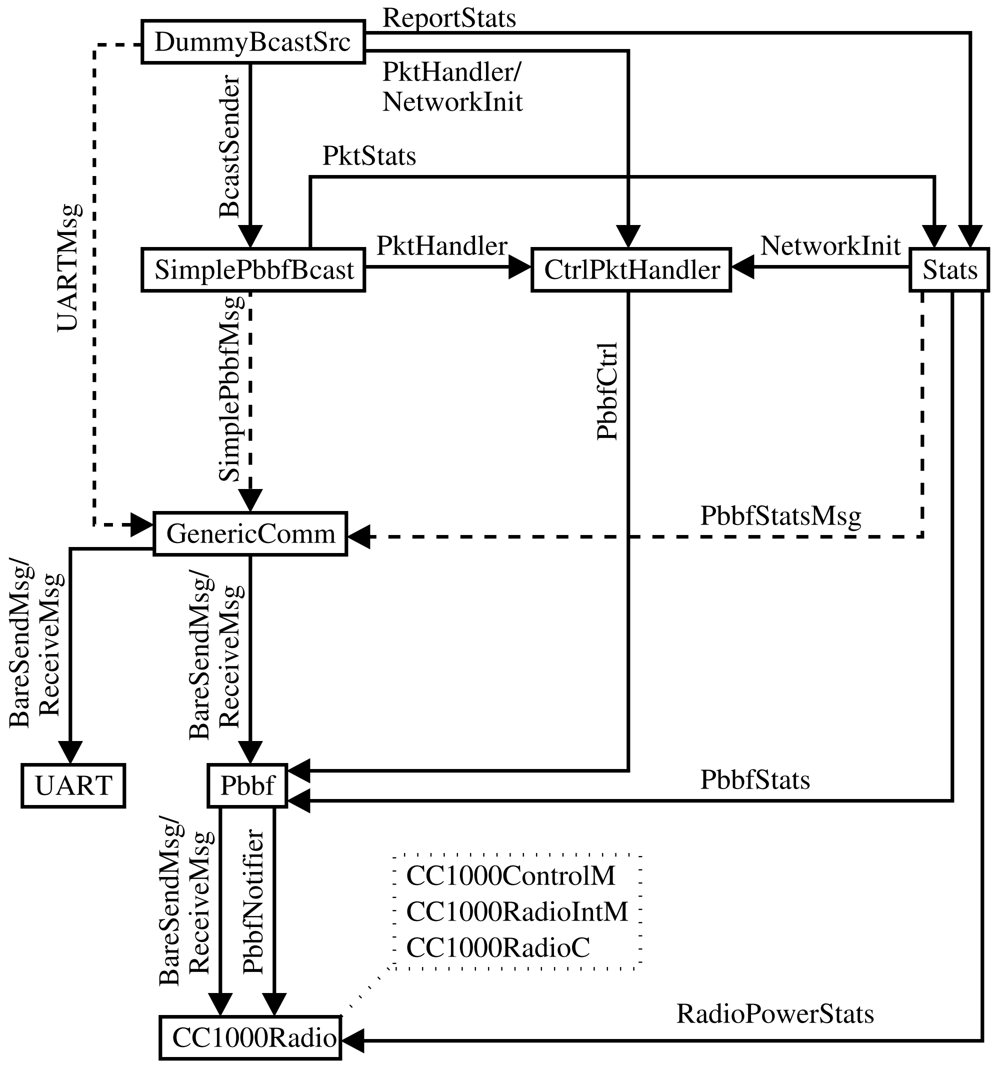

Fig. 3. TinyOS architecture for PBBF implementation. The solid rectangles are modules (`CC1000Radio` is an abstraction for the three modules listed in the dotted lines). The solid arrows represent the major interface(s) that connect modules. The incoming dashed lines to `GenericComm` represent the message types the connected module uses.

specified in the packet. This is done via its connection to `Pbbf`.

Stats: This module keeps track of statistics for our experiments and aggregates the information in packets to send back to the broadcast source. Via `DummyBcastSrc`, it keeps track of the end-to-end latency of received packet as well as the total number of unique, application-layer received packets. `SimplePbbfBcast` informs `Stats` of the total number of data packets sent and received. `Pbbf` signals to `Stats` when a packet was transmitted twice due to the r parameter (as discussed in Section IV). `CC1000Radio` signals this component whenever the radio switches to and from sleep mode to track the total fraction of time spent sleeping.

This module also provides a basic end-to-end retransmission scheme for added reliability in reporting experimental stats. We found this to be somewhat useful since the link layer retransmission scheme seemed to occasionally fail. One limitation of the link layer retransmissions in TinyOS is that no receiver is specified in the ACK packets. Thus, it is possible that both S_1 and S_2 send a packet to D at about the same time and, for whatever reason, D receives only, say, S_1's packet. However, the ACK send by D, which is intended for S_1 in this example, will also be overheard by S_2 and S_2 will considered its packet successfully received since the ACK does not specify if it is for S_1 or S_2. However, we did not run tests to determine if this was the source of occasional failures for link layer retransmissions since this was a peripheral issue that the end-to-end retransmissions seemed to fix. We mention it, though, as a possibility.

GenericComm: *(Existing TinyOS module)* This serves primarily to multiplex and demultiplex packets in TinyOS based on the packet type. Essentially, the packet type serves the role that ports do in traditional TCP/UDP communications

UART: *(Existing TinyOS module)* This component provides the lower level communication with the serial port.

Pbbf: This is the actual implementation of the PBBF protocol. It is placed between `GenericComm` and the `CC1000Radio` components. `GenericComm` is analogous to the network layer and `CC1000Radio` provides the medium access and the physical layer.

The p, q, and r values that `Pbbf` uses are input from `CtrlPktHandler`. B-MAC notifies `Pbbf` of a decision point for whether to sleep via the `PbbfNotifier` interface. At this point, `Pbbf` compares the current q value to a random number to decide whether to tell the radio to sleep, as would be normal operation, or continue listening to the channel, which is part of PBBF. For every packet received from `GenericComm`, `PBBF` decides, based on the p and r values, whether to use a long preamble and whether to transmit the packet twice, respectively. This layer also provides link layer retransmissions since this feature is not implemented in lower layers (i.e., `CC1000Radio`).

CC1000Radio: *(Existing TinyOS modules)* These components provide the lower level communication with Chipcon's CC1000 radio [12] found on Mica2 Motes. Additionally, the B-MAC [4] implementation is integrated into these components.

VI. Experimental Results

To test our implementation, we set up the following experiments. The broadcast source, with ID 0, was attached directly to a laptop via a MIB510CA board. This sensor also served as the sink for reporting statistics back to the laptop. Our Mica2 Motes used the 433 MHz frequency. We were constrained to using only nine Motes total, so the other eight Motes served as broadcast receivers.

Initially, we planned a multihop topology. However, statistics reporting proved far too unreliable for the environment in which we attempted this (see Section VII for more details). Thus, we only experimented on a topology where all of the devices were within range of the broadcast source (and each other). This setup was also beneficial since a limited number of Motes were available and PBBF relies on some amount of density to operate efficiently. Most importantly, this simple scenario is sufficient for demonstrating some of the key properties of PBBF.

In our experiments, the source transmitted a broadcast every 2.5 s. Each experiment ran for 30 s, which results in 11 packets being sent per run (the first packet is not sent immediately when the test commences). Each data packet uses the standard TinyOS format with 2 synchronization bytes, 5 header bytes, 2 CRC bytes, and a payload of 29 bytes. The default preamble adds an additional 8 bytes, though, as described in Section II, B-MAC increases the preamble length according to how much power saving is desired. In our tests, we set the B-MAC parameters to have a duty cycle of 135 ms and preamble size of 371 bytes. We note that when a sender decides to transmit immediately, according to the p parameter in PBBF, the preamble size is set to the default 8 bytes for that particular packet. We also note that the version of B-MAC we used carrier senses the channel for 8 ms once every duty cycle. If the channel is not carrier sensed idle, then B-MAC extends the time that it is awake for 32 ms. At the end of this 32 ms interval, B-MAC carrier senses again and will sleep or extend its listening for another 32 ms depending on if the channel is idle or busy, respectively. For statistics collection, once the sensor has run the experiment for the specified 30 s length, it switches power save off for 10 s and reports its data.

The metrics that we measured are:

- **Fraction of Time Not Sleeping:** Obtaining fine-grained energy measurements for the Motes requires special equipment. Thus, we use a coarse-grained metric where we track how much time a node spends with its radio not in the sleep state over the course of an experiment. Thus, the larger the fraction of time not sleeping, the more energy is generally being consumed by the radio.
- **Average Broadcast Latency:** This is the average latency from the time a packet is sent at the sender's application layer until the *data* begins transmission over the radio (i.e., after the preamble and synchronization bytes have been transmitted). For this, we use the time stamping implementation described in [13]. Again, this is not as fine-grained of a metric as we would like. However, this technique obviates the need for time synchronization among the nodes which would induce a large amount of complexity and overhead to our implementation. We only compute the latency for nodes that received a given broadcast.
- **Unique Data Packets Received:** We measure the average fraction of broadcasts sent by the source that are received by listening nodes.
- **Total Data Packets Received:** This is a measure of the receive overhead of the protocol. It is the average total broadcasts received divided by the number of broadcasts sent by the source. Since sensors filter duplicate broadcast packets (with respect to the source and sequence number), the total data packets received is greater than or equal to the unique data packets received.
- **Total Data Packets Sent:** This is a measure of the sending overhead of the protocol. It is the average total broadcasts sent by a node (excluding the broadcast source) divided by the number of broadcasts send by the source.

To test the effects of p, q, and r, we set their values to 0.0, 0.3, 0.7, and 1.0 and ran one experiment (with multiple broadcasts) for each of the 64 possible combinations of these three variables using these four values. For clarity of presentation, we omit some of the combinations in our graphs.

In Figure 4, we show the effects of p on energy consumption. When $q = 1$, obviously no energy savings occurs. When $r = 0$, the energy consumption decreases with p because less

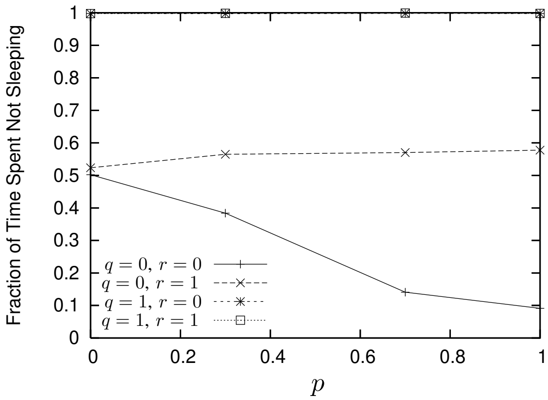

Fig. 4. Energy consumption.

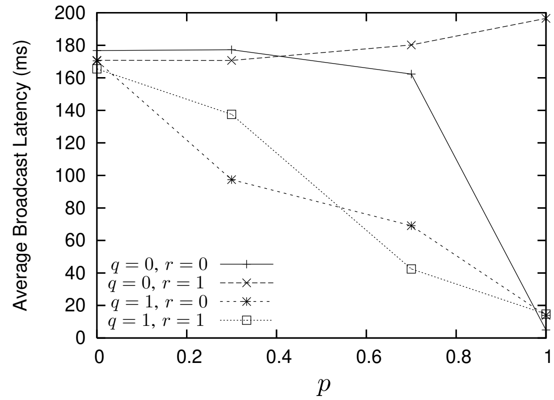

Fig. 5. Average broadcast latency.

packets are being received by the nodes (it is proportional to the corresponding curve shown in Figure 6). If a sensor is receiving only a few packets, then it will be sleeping most of the time. When $r = 1$, every packet transmitted immediately is also transmitted a second time using the power save protocol. This results in increased overhead and reliability. However, since more packets are being sent and received, the energy consumption is greater than for $r = 0$.

The curve is approximately flat when $q = 0$ and $r = 1$ because the number of transmissions that follow the B-MAC power save protocol remain the same. In B-MAC, such transmissions, with their long preambles, consume significantly more energy than the immediately sent packets. Thus, the dominating energy consumption component, the packets sent using the power save protocol, remains constant. The number of immediate sends increases as p increases, but the effect on the curve is small.

The effect of p on latency is shown in Figure 5. We can see how p improves the latency when $q = 1$. This improvement comes at the expense of energy consumption. The sharp drop-off in the $q = 0$, $r = 0$ case occurs because of a large decrease in reliability (shown in Figure 5). This is due to the fact that the latency is only computed for sensors that receive a broadcast. So, the few sensors that happen to receive the broadcast will do so with a small latency when p is high and $q = 0$. As an example, when $p = 0.3$, the reliability of the broadcast is about 80%. However, the slight decrease in latency when $p = 0.7$ comes at the expense of achieving only a 20% reliability. The $q = 0$, $r = 1$ case actually shows an increase in latency because the second packet being transmitted is what is usually being received. The second transmission occurs only after the first transmission that uses a short preamble.

Figure 6 shows, as expected, that if $q = 1$ or $r = 1$, the reliability is 100%. However, if both q and r are zero, then the reliability steadily decreases with p to the point of almost 0% reliability.

The overhead for receiving and sending in the protocols can be seen in Figure 7 and Figure 8, respectively. As expected, when $r = 1$, we get twice the overhead for both sending and receiving when compared to $r = 0$. In both figures, we see that the overhead increases as p increases when $r = 1$ since more packets are sent according to the p parameters and, hence, more packets are sent a second time according to the r parameter. In the case where $q = 0$ and $r = 0$ in these figures, we see a decrease in overhead since more packets are being transmitted according to the p parameter and less neighbors are listening at that time (since $q = 0$).

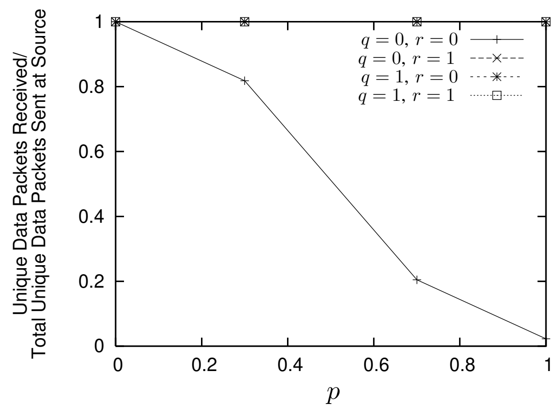

Fig. 6. Reliability of broadcast.

Consistent with our simulation results in [1], Figure 9 shows that an increase in q causes a linear increase in energy consumption. The $p = 1$ case uses significantly less energy than the other three cases at lower values of q. This is due to a decreased reliability compared with the other three cases. When no packets are being transmitted using the power save protocol, sensors sleep more since they are receiving fewer packets. This is similar to what was seen in Figure 4.

Figure 10 shows the effect of q on latency. In the $p = 1$, $r = 1$ case, the latency shows a linear decrease with q. This is because when $q = 0$, most of the broadcasts received are from the second transmission. However, when $q = 1$, the broadcasts received are from the first transmission instead of the second rebroadcast (as determined by the r parameter). In the $p = 1$, $r = 0$ case, when $q = 0$ the latency is low due to the low reliability (as shown in Figure 11). After this point, however, there is a gradual decrease in latency as more broadcasts are

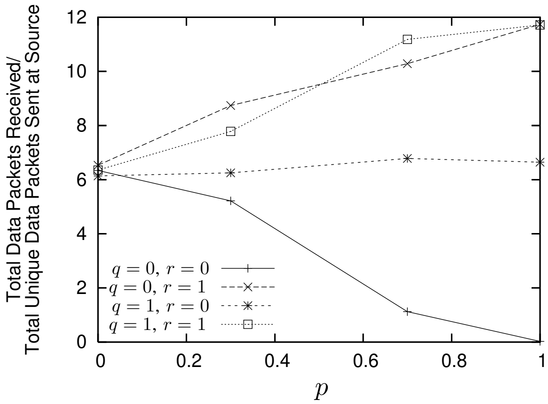

Fig. 7. Reception overhead.

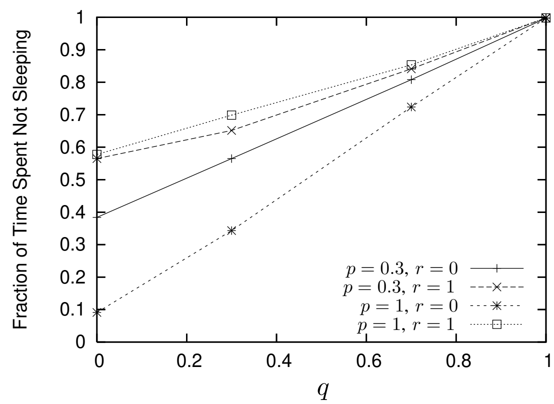

Fig. 9. Energy consumption.

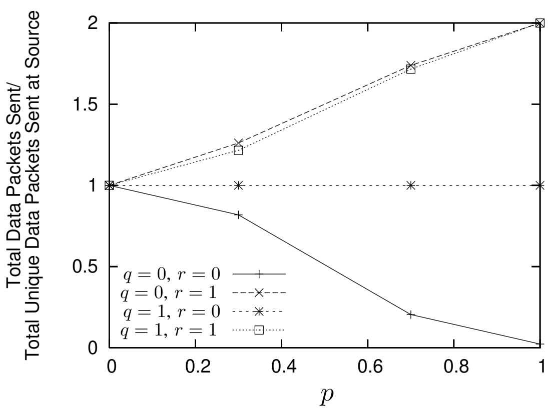

Fig. 8. Transmission overhead.

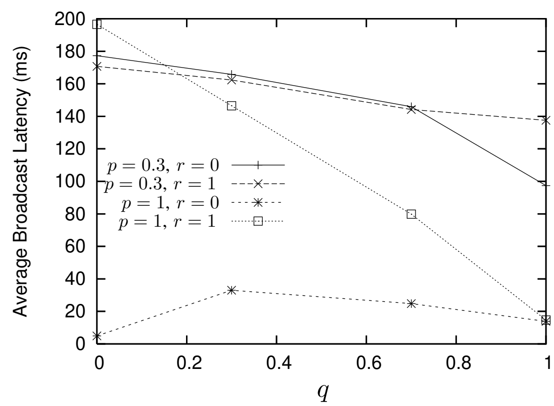

Fig. 10. Average broadcast latency.

received directly from the source rather than rebroadcasts by a neighbor. When $p = 0.3$ and $q = 0$, we can see that $r = 1$ can actually have a *negative effect* when compared with $r = 0$ due to the increased contention from the extra overhead induced.

The fraction of broadcasts that are received is shown in Figure 11. The interesting cases are only $p = 1$, $r = 0$ and $p = 0.3$, $r = 0$ since the other two curves have enough redundancy to give 100% reception. In both cases we can see how the reliability improves at q increases (the other two curves are flat at 100% regardless of q's value).

Finally, we tested the effects of r in our implementation. Figure 12 shows energy consumption. When $p = 1$, $q = 0$, we can see that the nodes use more energy due to the increase in reliability that the increasing r is providing. The reliability improvement with r is illustrated in Figure 13.

Figure 14 and Figure 15 show the overhead for receptions and transmissions, respectively. These results show that the overhead doubles when $p = 1$ and $q = 1$ as r goes from 0 to 1. This occurs because when $p = 1$, each sensor will transmit each broadcast once when $r = 0$ and twice when $r = 1$. When $p = 0$, we see no effects on the overhead, with respect to r, as expected. When $p = 1$ and $q = 0$, then the overhead is zero when $r = 0$ due to the lack of reliability. The increasing reliability with r causes the overhead to increase linearly.

From these results, we see that our implementation in TinyOS shows the same trends as we observed in simulation for energy consumption, latency, and reliability as a function of the p and q parameters. Additionally, we have shown the effects of a new parameter, r, which can improve reliability at the expense of extra packet overhead. Our figures show the quantitative performance of these three parameters on sensor hardware.

VII. Lessons Learned

In this section, we list some lessons that we learned as a result of our implementation.

Lesson 1: *A small fraction of seemingly trivial tasks will take a large fraction of your time.*
Getting a reliable serial connection between a Mote and the laptop proved extremely time consuming. The need for root access limited our PC choices to laptops. Most laptops are equipped with only USB ports and not serial ports. However, the USB-to-serial adapters that we tried tended to produce non-deterministic errors where serial communication would succeed about 10-20% of the time. After devoting significant time working under the assumption that the operating system needed configured correctly, we eventually had to purchase a laptop with a native serial port.

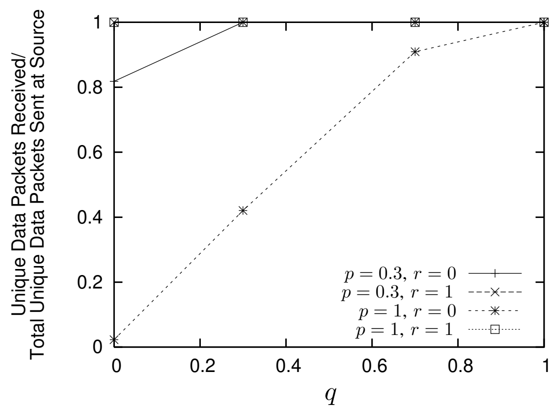

Fig. 11. Reliability of broadcast (all curves overlap except for $(q = 0, r = 0)$.

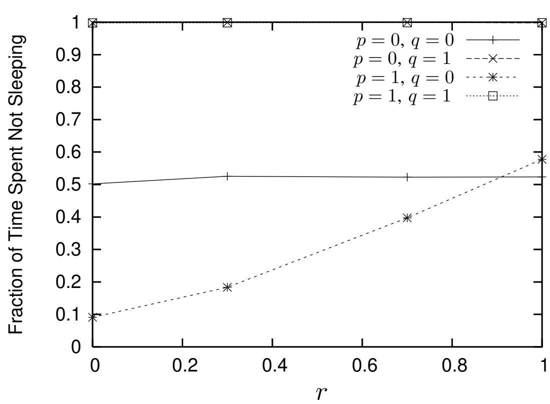

Fig. 12. Energy consumption.

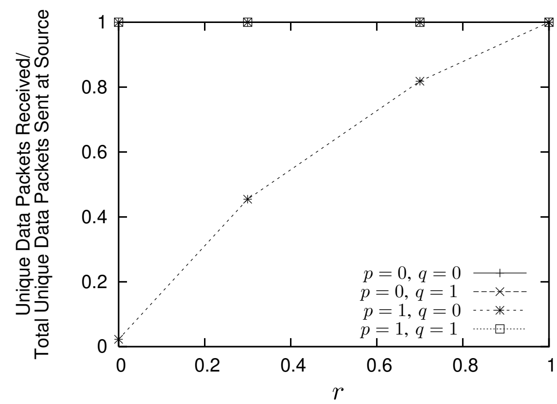

Fig. 13. Reliability of broadcast.

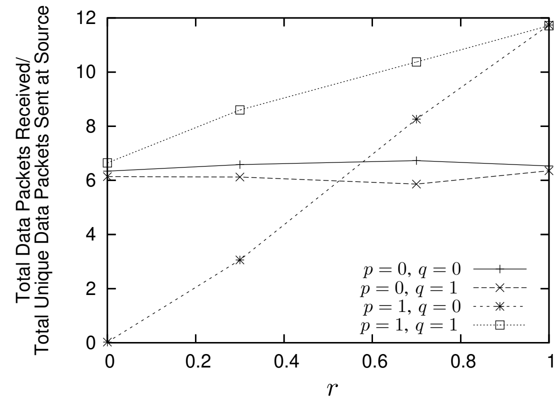

Fig. 14. Reception overhead.

Similarly, some aspects of TinyOS code are poorly documented and commented (though, overall, the documentation is good relative to other open source projects). Thus, some questions that could be answered quickly by someone familiar with the system took a much longer time to figure out by perusing code, documentation, and running applications. Examples include finding that link layer retransmissions were not implemented and discovering how to code them correctly. Another example is determining the differences among the routing protocols provided in TinyOS and discovering how to use these components correctly.

Lesson 2: *Multihop topologies are much more difficult to create than in simulation.*

We found this particularly difficult in an indoor setting. Contrary to our assumption that one could just place devices in a large seminar room spaced by a fixed distance, we discovered that this task requires much more in the way of measurement studies in a specific location to create a topology. Factors such as the height of a device, its distance to other objects, and asymmetry of communication links proved extremely complex in our chosen environment. A much more rigorous measurement study of a location or, better, a specifically designed testing area are needed to create multihop topologies.

Lesson 3: *Statistic collection and software updates are extremely difficult without a wired backplane.*

A significant amount of effort was needed to create a system for collecting statistics that did not interfere with experimental runs. Without an out-of-band channel available, each node had to unicast its statistics for a run back to the sink using the same channel on which the experiments were run.

This was exacerbated by the fact that the experiments required power save protocols to be used, which decreased reliability and increased latency for packets. Our solution was to use local timers with a large "fudge" factor to account for synchronization errors so that all nodes would report their statistics at about the same time and could turn power save mode off while this was being done. Thus, one node's statistics collection was not interfered with by another node's statistics reporting.

Another major difficulty with the lack of a wired backplane is that every time a change is made to the code, each sensor must be manually connected to the laptop and receive the uploaded code. This approach is obviously neither scalable nor desirable during the debugging phase.

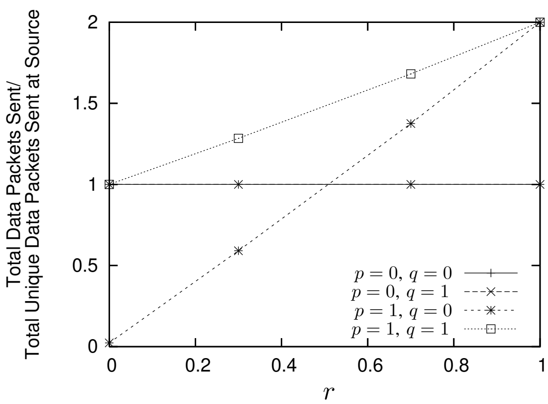

Fig. 15. Transmission overhead.

Lesson 4: *Debugging is difficult.*
Essentially, the only output available is three LEDs. No `gdb` or `printf` statements can be used when things do not go as planned. The simulator that is available with TinyOS is of some use, but some lower level hardware abstractions are not available (e.g., the time-stamping mechanism) or significantly differ from the Mica2 implementation (e.g., the channel bitrate in the simulator is hard-coded to be twice that of the Mica2 hardware). Again, debugging is an area that would greatly benefit from a wired backplane. Though, even this is made difficult by the fact that every module that needs debugged must be wired to the backplane component and it must create a new packet type to communicate its data.

Lesson 5: *Buffer management is difficult.*
In TinyOS, when an upper layer sends a packet to a lower layer, it is responsible for protecting the memory allocated to that packet until lower layers signal that they are finished handling it. Lower layers are responsible for sending back pointers to packet memory locations when they signal that they are finished. Coding must be done carefully to ensure that upper layers never reuse memory being handled by lower layers and that lower layers return memory pointers consistent with what upper layers are expecting.[2]

VIII. CONCLUSION

In our previous work [1], we proposed a lightweight protocol, PBBF, that allows lower latency broadcast propagation in power save networks in a energy efficient manner. Using this protocol, a user has more fine-grained control over the energy consumption for a broadcast to achieve a desired latency and reliability.

In this work, we have proposed a PBBF extension for improved reliability at the cost of increased packet overhead. Additionally, we designed and implemented a PBBF architecture in TinyOS [2]. Our evaluations show the energy consumption, latency, reliability, and overhead tradeoffs possible using PBBF on Mica2 [9] hardware.

[2]For example, an upper layer has two packet queues from which it is sending. The pointers to the packets at the heads of these queues are q_1 and q_2, respectively. Thus, if an upper layer sends the packet at pointer q_1, it would expect that the corresponding signal to indicate that the send is done will return pointer q_1 so that it knows which queue just finished being serviced. If a lower layer erroneously returns a different pointer, the upper layer cannot correlate which queue is being signaled as serviced.

REFERENCES

[1] M. J. Miller, C. Sengul, and I. Gupta, "Exploring the Energy-Latency Trade-off for Broadcasts in Energy-Saving Sensor Networks," in *IEEE ICDCS 2005*, June 2005.

[2] TinyOS Community Forum, `http://webs.cs.berkeley.edu/tos`.

[3] T. Starner, "Thick Clients for Personal Wireless Devices," *IEEE Computer*, vol. 35, no. 1, pp. 133–135, January 2002.

[4] J. Polastre, J. Hill, and D. Culler, "Versatile Low Power Media Access for Wireless Sensor Networks," in *ACM SenSys 2004*, November 2004.

[5] IEEE 802.11, *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 1999.

[6] C. E. Jones, K. M. Sivalingam, P. Agrawal, and J. C. Chen, "A Survey of Energy Efficient Network Protocols for Wireless Networks," *ACM Wireless Networks*, July 2001.

[7] Z. J. Haas, J. Y. Halpern, and L. Li, "Gossip-Based Ad Hoc Routing," in *IEEE Infocom 2002*, June 2002.

[8] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu, "The Broadcast Storm Problem in a Mobile Ad Hoc Network," in *ACM MobiCom 1999*, August 1999.

[9] MICA2 Mote Datasheet, `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/6020-0042-05_A_MICA2.pdf`.

[10] J. Polastre, R. Szewczyk, and D. Culler, "Telos: Enabling Ultra-Low Power Wireless Research," in *ACM/IEEE IPSN/SPOTS*, April 2005.

[11] W. Ye, J. Heidemann, and D. Estrin, "An Energy-Efficient MAC Protocol for Wireless Sensor Networks," in *IEEE Infocom 2002*, June 2002.

[12] Chipcon CC1000 Datasheet, `http://www.chipcon.com/files/CC1000_Data_Sheet_2_1.pdf`.

[13] M. Maróti, B. Kusy, G. Simon, and Á. Lédeczi, "The Flooding Time Synchronization Protocol," in *ACM SenSys 2004*, November 2004.

APPENDIX I
PBBF INTERFACES AND PACKET FORMATS IN TINYOS

A. Packet Formats

1) SimplePbbfMsg:

```
typedef struct SimplePbbfMsg {
   // Source can be either the broadcast
   // source or a predefined UART address
   uint16_t source;
   // A broadcast sequence number that is
   // unique per source
   uint8_t seqno;
   // How many hops the packet has traveled
   // from the broadcast source
   uint8_t hopCount;
   // The p value that the node should use
   uint8_t pVal;
   // The q value that the node should use
   uint8_t qVal;
   // The r value that the node should use
   uint8_t rVal;
   // Sequence number of the test run for
   // this packet
   uint8_t runSeqno;
   // Timestamp for stats
   // (granularity = 1/921.6 kHz)
   uint32_t latency;
} __attribute__ ((packed)) SimplePbbfMsg;
// packed attribute removes field padding
```

```
// on Mica2 architecture
```

2) PbbfStatsMsg:

```
typedef struct PbbfStatsMsg {
   // Source ID of the node reporting the
   // stats
   uint16_t nodeId;
   // Total amount of data packets sent by
   // SimplePbbfBcast
   uint16_t totalDataSent;
   // Total amount of data packets resent
   // according to the r parameter
   uint16_t totalDataResent;
   // Total data packets received by
   // SimplePbbfBcast (includes duplicates)
   uint16_t totalDataRecv;
   // Total application level packets
   // received (duplicates suppressed)
   uint16_t totalAppRecv;
   // Average end-to-end latency from
   // broadcast source to this node
   uint32_t avgLat;
   // Fraction of time the node's radio
   // was not sleeping 0=0%, 255=100%,
   // uniform spacing between
   uint8_t fracOnTime;
   // Sequence number of the test run for
   // which stats are being reported
   uint8_t runSeqno;
} __attribute__ ((packed)) PbbfStatsMsg;
// packed attribute removes field padding
// on Mica2 architecture
```

3) UARTMsg: UARTMsg can be of type either SimplePbbfMsg or PbbfStatsMsg.

B. Interfaces

1) PBBF Interface:

```
interface PbbfControl {
   // Each parameter can be set to one of
   // 11 discrete values corresponding to
   // probability values between 0.0 and
   // 1.0 spaced uniformly.
   command void setPLevel(uint8_t);
   command void setQLevel(uint8_t);
   command void setRLevel(uint8_t);
}

interface PbbfNotifier {
   // Signals PBBF module when a sleep
   // decision needs made
   event result_t sleepDecisionPoint();
   // PBBF tells the signaling module
   // whether or not to sleep
   command void setSleep(bool);
}
```

2) Broadcast Send Interface:

```
interface BcastSender {
   // Used by the application to send
   // a broadcast packet
   command result_t send(TOS_MsgPtr);
   event result_t sendDone(TOS_MsgPtr,
      result_t);
}
```

3) Stats Handling Interface:

```
interface PktStats {
   // A packet was sent, bool tells
   // whether it was a control packet
   command void SentPkt(bool);
   // A packet was received, bool tells
   // whether it was a control packet
   command void RecvdPkt(bool);
   // Collect the stats from the received
   // packet.  uint32_t is the latency of
   // the packet since it was sent by the
   // source.
   command void HandleRecvdStats(
      TOS_MsgPtr, uint32_t);
   // Signal that the stats have been
   // handled
   event result_t HandleRecvdStatsDone(
      TOS_MsgPtr);
}

interface RadioPktStats {
   // Signal to the stats module when the
   // radio switches on and off
   event result_t radioPoweredOn();
   event result_t radioPoweredOff();
}

interface ReportStats {
   // Used by the stats collection module
   // to transmit collected stats back
   // to the sink.
   command result_t ReportStats();
   event result_t ReportStatsDone(
      result_t);
}

interface PbbfStats {
   // Signal to the stats module when PBBF
   // sends a packet twice according to
   // the r parameter.
   event result_t didSecondSend(
      TOS_MsgPtr);
}
```

4) Control Packet Handling Interface:

```
interface NetworkInit {
   // Signals a component when a control
   // packet has been received to
   // initialize the current test run
   // for an application.  The input
   // parameters gives a unique sequence
   // number to identify the test run.
   event result_t isInitialized(uint8_t);
}

interface PktHandler {
   // When a control packet is received,
   // pass to the control packet
   // handling module.
   command result_t handle(TOS_MsgPtr);
   event result_t done(
      TOS_MsgPtr, result_t);
}
```