802.11 Power Management Extensions to Monarch *ns*

John Dorsey and Dan Siewiorek

December 2004
CMU-CS-04-183

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We describe our improvements to the IEEE 802.11 support in the Monarch version of the *ns* network simulator. Our extensions add several management features needed for power management, such as beacon frames and the timing synchronization function. We implement the queuing and traffic announcement facilities at the core of power management. A method for recording the energy consumption of the wireless interface is also presented. Several other enhancements are discussed, such as support for the 802.11b high-rate PHY and corrections to existing features in the simulator. These changes greatly increase the usefulness of *ns* for researchers studying the performance of *ad hoc* networks under power management.

This work was supported by an Intel Ph.D. Fellowship, NSF grant CCR0205266, and Army Research Office grant DAAD19-01-1-0646.

Keywords: *Ad hoc* networks, Power management, Network simulation.

Contents

List of Figures

1 Introduction

IEEE 802.11 wireless LANs have become very popular MAC and PHY layer technologies in *ad hoc* networking research. These wireless network interfaces experience high energy consumption[1] while in the *idle* state, waiting to send or receive frames [4, 5, 7]. A power management mode has been defined to reduce the energy costs of the idle state, but it exhibits poor latency performance in multihop infrastructureless environments.

Many methods to improve the performance of multihop *ad hoc* networks under 802.11 power management have been proposed. These range from proactive relay election [1] to reactive timer-based approaches [14] to demand-driven route negotiation [3]. What these methods all have in common is their use of *power management suspension*. When a node is "active" under any of these schemes, it suspends the use of the 802.11 features which reduce energy consumption at the expense of latency.

We have extended the Monarch version [2] of the *ns* network simulator [6] to support research into 802.11 power management in *ad hoc* networks. Our changes are extensive, more than tripling the size of the 802.11 code. They include an implementation of several management features such as beacon frames and the timing synchronization function. We have implemented the queuing and traffic announcement features that underly 802.11 power management in the infrastructureless environment. We have added instrumentation to measure the energy consumption of the simulated wireless interface. In the course of de-

[1] In this report, we focus on the energy consumption of the entire network interface, which includes components such as a firmware processor, modem, RF converter, and RF amplifier. This is distinct from the energy radiated onto the wireless medium, which tends to be orders of magnitude smaller.

veloping these extensions, we evaluated several nonstandard improvements to various 802.11 features, which we describe.

In this report, we reference sections — also called "clauses" — of the 802.11 specification [12] using the shorthand "§." For example, "(§7.1)" refers to the section on MAC frame formats in the specification. When describing a section of some other document, including this report, we will indicate so explicitly.

It is our hope that this improved simulator will be helpful to future researchers investigating power management in *ad hoc* networks. We welcome any feedback on experiences with these extensions.

2 802.11 Management Features

The most familiar aspects of the 802.11 design are the *control* features, which regulate access to the wireless medium. These algorithms remain essentially unchanged in the presence of power management. Section 11 of the 802.11 specification [12] defines the *management* features which allow stations to coordinate at a higher level. These features enable timer synchronization, which is critical for the *rendezvous* nature of 802.11 power management. They also enable stations to coordinate their frame exchanges with the use of the low-power *doze* state.

2.1 Synchronization

The fundamental unit of organization in an 802.11 network is the basic service set, or BSS. A BSS is a group of stations acting under the same coordination function. For instance, the 802.11 distributed coordination function (DCF) is the well-known CSMA/CA method of statistical multiplexing using Request to Send (RTS), Clear to Send (CTS), Data, and Acknowledgment (ACK) frames. When a

BSS is operating without access to an infrastructure network, it is termed an *independent* basic service set, or IBSS. This is the mode of operation relevant to *ad hoc* networking.

All stations in a BSS are required to maintain a timer synchronized to a common clock. In infrastructure networks, the access point provides the synchronization by periodically broadcasting beacons. In an IBSS, there is no such distinguished station, so a distributed algorithm is used to synchronize the timers.

2.1.1 Station Timers

The microsecond-resolution timer at each station is required to be accurate within $\pm 0.01\%$ by the specification (§11.1.2.4). We simulate timer drift by randomly assigning each station a drift value, and adjusting timer samples by the amount of the drift. This method assumes that, following a timer update, the timer deviates from "absolute" (*ns*) time at a fixed rate.

Time is continuous in *ns*; most events in the simulator are dispatched by real-valued timers. Physical movement and signal propagation are two aspects of the simulator which occur in absolute time. A station's timer may be faster or slower than absolute time, according to its assigned drift value. For an interval $\Delta\tau$ measured using a station's timer, the corresponding absolute interval Δt is given by $\Delta t = \frac{\Delta\tau}{1+d}$. The timer drift d is, for each station, chosen from a uniform random distribution on $[-0.01\%, 0.01\%]$, consistent with the accuracy requirements of the specification.

To sample the station timer, we need the absolute time t_{now}, available from `Scheduler::instance().clock()`. We also need two additional pieces of state: the absolute time of the last timer update, t_{up}, and the value of the station

timer at that update, τ_{up}. We can then compute the station timer value associated with the present moment:

$$\tau_{\text{now}} = \left[\left(t_{\text{now}} - t_{\text{up}}\right) \times (1 + d)\right] + \tau_{\text{up}}$$

Several timer drift methods are provided which return the value of the station timer, or convert between absolute time and station time:

- `Mac802_11::get_tsf_timer()` returns the current station timer value, rounded to the nearest microsecond.
- `Mac802_11::adjusted_time(double t)` returns the duration of absolute time that would elapse if the duration `t` were measured using the station's timer.
- `Mac802_11::tsf_time(double a)` returns the microsecond-resolution duration that would be measured by the station's timer when the absolute duration `a` elapses.

2.1.2 Beacon Frames

The stations in an IBSS periodically contend to generate a beacon frame, which describes the parameters of the network and synchronizes station timers. Beacon frames are transmitted to the broadcast address, and have frame type 00_{two} and subtype 1000_{two}. They are recognizable in the *ns* trace by the first octet of the frame control field, "`[80...`," as shown below:

```
s 0.000656894 _1_ MAC --- 0 MAC 80 [8010 0 ffffffff 1 290]
```

Figure 1 illustrates the other fields in an 802.11 frame dump. Each such line begins with a *type* indicating whether the event records a frame being sent ("`s`"),

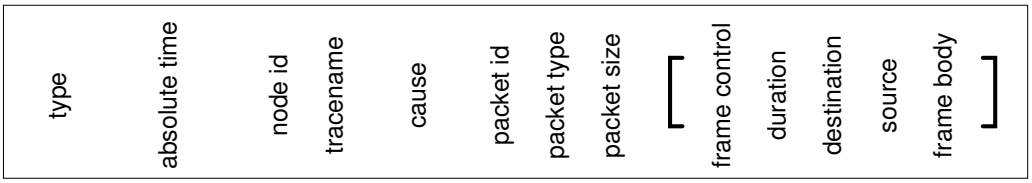


Figure 1: *ns* trace fields for 802.11 frames.

received ("`r`"), or dropped ("`D`"). Next, the absolute simulation time of the event is displayed, in seconds. The node address and the trace name "`MAC`" follow. The next field is used for drop events to indicate the reason why the frame was discarded. The causes are defined in *cmu/cmu-trace.h*; for example, the cause "`PWR`" is described in Section 2.2.2. The packet id is used by higher-layer agents such as `DSRAgent`; all 802.11 control and management frames have an id of `0`. 802.11 frame events have a packet type of "`MAC`," followed by the frame size in octets. The next five fields (enclosed within "`[...]`") expose the Frame Control field (§7.1.3.1), Duration (§7.1.3.2), Destination Address (§7.1.3.3.4), Source Address (§7.1.3.3.5), and the first two octets of the Body (§7.1.3.5) for Data frames.

In our implementation, beacons contain the following parameters:

- **Beacon Interval** (§7.3.1.3), the period between successive target beacon transmission times (TBTTs). This value is expressed in 802.11 time units (TUs), which are 1,024μs in length. For example, our implementation encodes a beacon interval of 200ms as 196 TUs, or 200.704ms.

- **Capability Information** (§7.3.1.4) describes features of the network such as the use of encryption or access to an infrastructure network. Our implementation indicates infrastructureless operation by clearing the Extended Service Set (ESS) bit and setting the IBSS bit. We also clear the Contention-Free (CF) Pollable, CF Poll Request, and Privacy bits.

- **Timestamp** (§7.3.1.10) contains a 64-bit encoding of the sender's station timer, in microseconds. This representation keeps time for about 584,542 years before rolling over.

- **Service Set Identity** (§7.3.2.1), or SSID, is a variable-length field (up to 32 octets) which names the extended service set to which this network belongs. Our implementation uses a zero-length field, indicating the broadcast SSID.

- **Supported Rates** (§7.3.2.2) lists the data rates in the basic rate set (which must be supported by all stations in the network) and supported rate set (which are optional). Our implementation lists the 1Mbps and 2Mbps rates in the basic rate set, encoded as 82_{sixteen} and 84_{sixteen}, respectively. We also list 11Mbps as a supported rate, encoded as 16_{sixteen}.

- **DS Parameter Set** (§7.3.2.4) lists the current channel of this direct sequence spread spectrum (DSSS) network. Our implementation indicates channel 1 — 2,412MHz (§15.4.6.2) — which is consistent with the initialization of `NetIf/SharedMedia` in the script *cmu/scripts/run.tcl*.

- **IBSS Parameter Set** (§7.3.2.7) describes an IBSS-specific parameter, the duration of the Announcement Traffic Indication Message (ATIM) window within each beacon interval. The ATIM window is explained in Section 2.2.1 of this report. The value is expressed in TUs; for example, our implementation encodes an ATIM window of 40ms as 40 TUs, or 40.96ms.

All stations in the IBSS participate in a distributed beacon generation algorithm (§11.1.2.2). A station initializes an IBSS by broadcasting a beacon bearing the parameters for the IBSS, including the beacon interval. The time at which

this beacon is transmitted, coupled with the beacon interval parameter, defines the complete sequence of target beacon transmission times (TBTTs) for the IBSS.

Every station which subsequently joins the IBSS will participate in the beacon generation algorithm at each TBTT. A station joins the IBSS by receiving a beacon and adopting the parameters contained therein. Our implementation treats the time at which a first beacon is received to be a TBTT.

Beacon generation uses a different backoff timer than that used for normal frame transmission. At each TBTT we set `Mac802_11::use_alt_backoff`, which indicates that use of the normal backoff timer (`Mac802_11::mhBackoff`) is disabled. We then set the beacon backoff timer `Mac802_11::mhBeacon` to a time drawn from a uniform distribution on $[0, 2 \times \text{aCWmin} \times \text{aSlotTime}]$, where aCWmin is obtained from the DSSS `PHY_MIB::CWMin` and aSlotTime is obtained from `PHY_MIB::SlotTime`. If a beacon arrives while the backoff timer is running, then the timer and beacon transmission attempt are canceled. If the backoff timer expires and no beacon has arrived, then the station transmits a beacon. Unless power management is in use, the station then resumes the use of the normal backoff timer.

2.1.3 Timing Synchronization Function

A station must adopt any parameters in a beacon received with a later timestamp than the station's own timer (§11.1.4). In our implementation, stations initialize with a timer value of zero. They then join the IBSS by receiving a beacon with a timestamp greater than zero, and adopt the timestamp, beacon interval, and ATIM window duration values contained in that beacon.

By adopting later timestamps, stations synchronize their timers to their fastest neighboring station. The scalability of this method has been studied [8]; the fastest stations can lose synchronization with the rest of the IBSS. This possibility has not caused practical problems in our own experiments.

2.2 Power Management

The 802.11 IBSS design addresses the high energy costs of the radio idle state with a *rendezvous*-based power management scheme. Inactive stations can place their transceiver electronics in a low-power *doze* state, which prevents the wireless interface from sending or receiving frames. Using the synchronized station timers, all IBSS stations periodically wake and exchange announcements about their offered traffic. Stations which do not need to remain awake and send or receive data may return to the doze state.

2.2.1 Theory of Operation

Conceptually, the wireless network interface can be in one of four major states: *doze*, *idle*, *receive*, or *transmit*. Transitions between *idle*, *receive*, and *transmit* are governed by the medium access algorithm itself. For example, a station which sends a directed data message first waits in the *idle* state for the medium to become available. It then moves to the *transmit* state to send a request to send (RTS) frame, returns to *idle*, then moves to *receive* when the clear to send (CTS) frame arrives, and so forth.

Transitions to and from the low-power *doze* state are managed above the medium access control level. At each TBTT, all stations must emerge from *doze* and be prepared to send or receive beacon frames. Following beacon generation,

the stations remain awake and exchange *announcements* about their scheduled traffic activity. An announcement is an indication that a station has buffered data frames addressed to a named destination. Stations which announce their destinations, or which receive announcements addressed to themselves (or the broadcast address), must remain awake until the next TBTT. The remaining stations return to the low-power *doze* state.

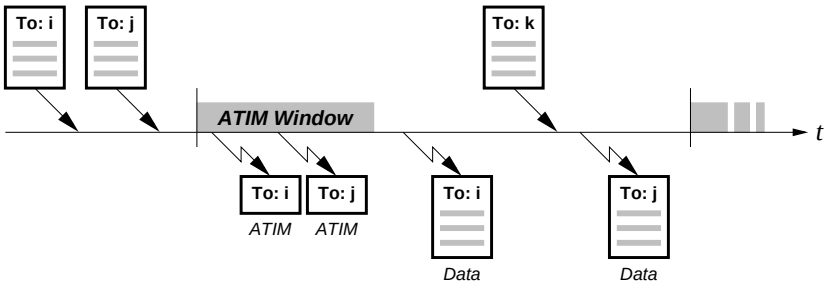


Figure 2: 802.11 ATIM frame transmission.

Figure 2 shows the traffic announcement process. When data frames are passed down to the MAC layer, the MAC implementation must check to see if the frame destinations might be awake. If the destinations are not known to be awake, the frames are buffered until the next beacon interval. At each TBTT, all stations wake and enter an *ATIM window*, the duration of which is a parameter of the IBSS. During the window, no data frames may be sent. Stations exchange ATIMs to indicate their scheduled data activity. Following the ATIM window, stations transmit buffered data frames to those destinations which were successfully announced.

2.2.2 Power State Transitions

In addition to the major power states of *doze*, *idle*, *receive*, and *transmit*, our implementation separately models the transitions into and out of the *doze* state. Published measurements of 802.11 interfaces indicate that *doze* transitions are

non-ideal and have a distinct power rate [10, 11]. Our implementation simulates a 250μs transition to *doze* (`TO_DOZE_TIME`) and a 250μs transition out of *doze* (`FROM_DOZE_TIME`).

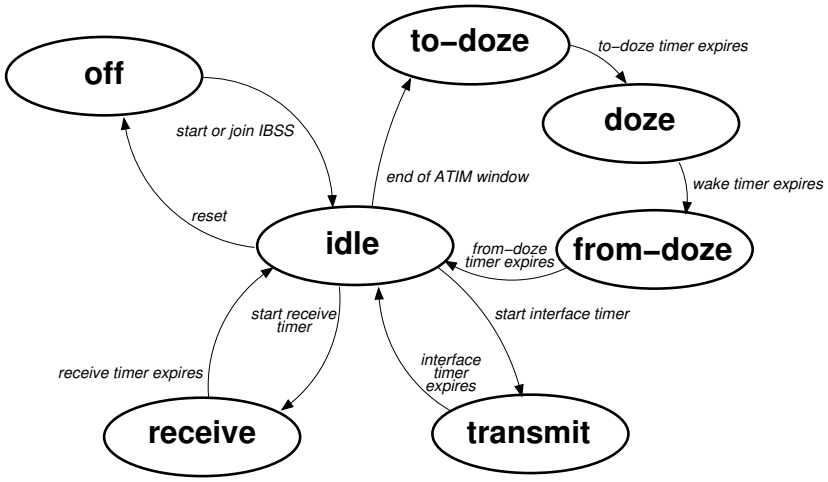


Figure 3: Power state machine.

Figure 3 shows the state machine implemented by our power model. All stations begin the simulation in the *off* state. The script *cmu/scripts/mobile_node.tcl* schedules node 1 to start the IBSS at time zero, at which point that node's MAC interface transitions to the *idle* state. All other stations are scheduled to try and join the IBSS at time 1.0, when they also transition from *off* to *idle*. All stations reset their interfaces at the end of the simulation, returning them to the *off* state.

When a new frame arrives at a station, `Mac802_11::recv()` is called to process the reception. If the interface is in the *off* state, or is dozing as will be described later, the frame is dropped. Dropped frames appear in the frame trace with the cause "`PWR`," as shown below:

```
D 0.201508211 _2_ MAC PWR 0 MAC 80 [8010 0 ffffffff 1 130f]
```

If the frame is not dropped, and can otherwise be received (*i.e.*, a frame collision has not occurred), then the receive timer `Mac802_11::mhRecv` is started. At the same time, the interface transitions to the *receive* state. The receive timer

expires when reception completes, and the interface returns to the *idle* state in `Mac802_11::recvHandler()`.

When an outgoing frame becomes available for transmission, the interface timer `Mac802_11::mhIF` is started, and the interface transitions to the *transmit* state. Once the transmission is complete, the interface timer expires. The interface returns to the *idle* state in `Mac802_11::txHandler()`.

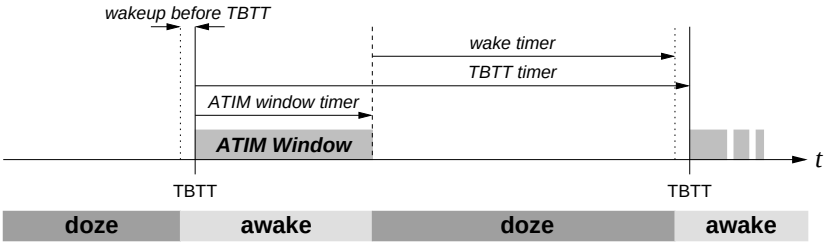


Figure 4: Relationship between TBTT, ATIM window, and wake timers.

When a station uses power management, it enters the low-power *doze* state during beacon intervals in which it is not scheduled to send or receive frames. *Doze* state transitions are managed by several new timers as shown in Figure 4. At each TBTT, a station restarts its TBTT timer, `Mac802_11::mhTBTT`, which expires in one beacon interval. In addition, a power managing station enters the ATIM window, and starts the ATIM window timer `Mac802_11::start()`, which expires at the end of the window.

At the end of the ATIM window, a station must decide whether it will remain awake, or return to the *doze* state. Broadly, the rule is that a station cannot enter *doze* if it sends or receives an ATIM frame — addressed to itself or to a multicast address — during the ATIM window (§11.2.2.3). Specifically, the station cannot doze once it *transmits* an ATIM frame, regardless of whether the frame is acknowledged or not (in the case of a directed ATIM). Also, a station that wins beacon contention and transmits a beacon frame cannot doze.

Our implementation decides to doze in `Mac802_11::end_atim_window()` if *all* of the following conditions hold:

- The station has not received any ATIM frames addressed to itself or to the broadcast address in the current ATIM window.
- The station has not transmitted any beacon frames in the current beacon interval. (This can be overridden, as described in Section 4.3.)
- The station has not transmitted any broadcast ATIM frames, has not abandoned a directed ATIM frame after exceeding the retry count, has not abandoned a directed ATIM frame at the end of the ATIM window, and is not in the process of transmitting an ATIM frame.
- Use of the *doze* state is not disabled using power management suspension, described in Section 4.1.
- The duration until the next TBTT is greater than the time before a TBTT at which a dozing station would normally wake.

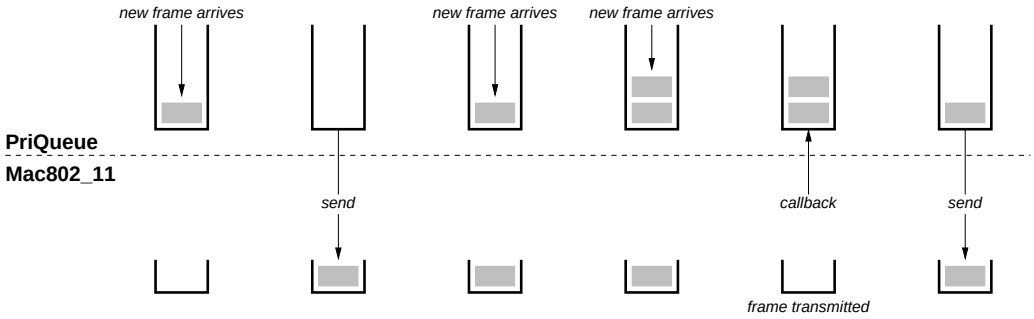
Once a station decides to doze, it schedules its entry into the *doze* state via an intermediate *to-doze* state by setting the to-doze timer `Mac802_11::mhToDoze`. Once this timer expires, the interface is set to the *doze* state and the wakeup timer `Mac802_11::mhWake` is started. This timer will expire a short time before the next TBTT. The reason for the early wakeup is to make sure that a station does not miss hearing a beacon transmission due to timer asynchronism. Stations wake 3ms prior to the TBTT (`DEFAULT_WAKEUP_BEFORE_TBTT`). When the wakeup timer expires, the interface enters the *from-doze* state and sets the from-doze timer `Mac802_11::mhFromDoze`. Once this timer expires, the interface is fully awake and returns to the *idle* state.

While the interface is in the *to-doze*, *doze*, or *from-doze* states, it can neither send nor receive frames. The test `Power::is_dozing()` returns true when the interface is in any of these states. Also, the test `Power::is_off()` returns true when the interface is off.

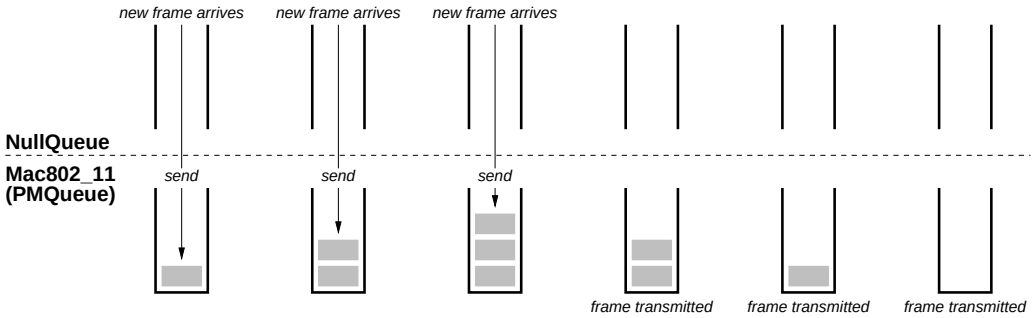
2.2.3 Queuing

The `PriQueue` link layer message queue buffers messages from higher protocol layers before handing them off to the `Mac802_11` interface. This queue releases one message at a time to the MAC layer, and waits for that message to be transmitted or abandoned before releasing another. Figure 5(a) shows the partial sequence of events when three messages are dispatched to the link layer back-to-back. The first message is passed to the MAC code using `Mac802_11::send()`. While this message is being transmitted, the remaining messages arrive at the link layer queue, where they are buffered. Once the MAC layer transmits the frame, it calls back to the link layer queue, which releases the next message.

When used with IBSS power management, this design can lead to low medium utilization. Suppose the three messages passed to the link layer were destined for addresses A, B, and A. The first message to reach the MAC layer will trigger an ATIM traffic announcement for A in the next ATIM window (if necessary). After the data frame to A is transmitted, the next message — to B — is released. B has not been announced, so the MAC layer waits until the *next* ATIM window to announce B, and then later, send the data frame to B. A similar delay occurs when the final frame — again, to A — is sent, since A was not announced during the beacon interval in which the frame to B was sent.



(a) Without power management, using PriQueue.



(b) With power management, using NullQueue and PMQueue in the 802.11 layer.

Figure 5: Message queuing with PriQueue and NullQueue.

Instead of passing messages to the MAC layer one at a time, we should instead pass each message as soon as possible. This allows the MAC layer to announce all of its pending destinations at the earliest possible time. Figure 5(b) shows this approach, replacing `PriQueue` with a trivial `NullQueue`, which passes messages directly to the `Mac802_11` interface immediately. Upon arriving at the MAC layer, the messages are placed in a `PMQueue`, the contents of which are inspected at the beginning of each ATIM window. A list of message destinations is constructed, and these destinations are announced with ATIM frames.

`PMQueue` is always used when 802.11 IBSS power management is enabled in our implementation. Section 8.1 explains how to override `PriQueue` with `NullQueue` when running *ns*.

2.2.4 ATIM Frames

Following a TBTT, ATIM announcements are sent "after a Beacon frame is either transmitted or received by the [station]" (§11.2.2.1). Specifically, our implementation begins the ATIM process with `ATIMQueue::begin_announcements()` under the following conditions:

- The station has won beacon contention, and transmitted a beacon.
- The station has received its first beacon during the current beacon interval. This includes the special case of the first beacon a station receives after trying to join an IBSS, which defines its sequence of TBTTs, and therefore, ATIM windows.
- Optionally, the station has abandoned its attempt to transmit a beacon. (Described in Section 4.4.)

When ATIM transmission begins, there are two sources for the destination addresses which will be announced by `ATIMQueue`. The highest priority source contains addresses for which an announcement has been scheduled independent of the `PMQueue` contents. This kind of scheduling is described in Section 4.2. The other source is the `PMQueue` itself, which returns a complete list of its current destination addresses with `PMQueue::get_destinations()`. Regardless of source, a station announces an address at most once during an ATIM window.

ATIM frames have type 00_{two} and subtype 1001_{two}. They are recognizable in the *ns* trace by the first octet of the frame control field, "`[90...`," as shown below:

```
s 5.622507966 _3_ MAC --- 0 MAC 52 [9010 103 2 3 0]
```

Announcements for the broadcast address are always considered successful. Directed announcements, however, must be acknowledged to succeed. Our implementation retries unacknowledged ATIMs up to the relevant retry limit (§9.2.5.3).[2] We never precede a directed ATIM with RTS/CTS.

When a directed ATIM fails to elicit an acknowledgment after the threshold number of retries, we treat the destination as *unreachable*. Following the ATIM window, messages in the `PMQueue` are passed back to `Mac802_11` in two stages. First, those messages addressed to unreachable stations are returned using `Mac802_11::send_unreachable()`, which invokes the transmit failure callback for each such message. Next, messages for announced destinations are passed in FIFO order using `Mac802_11::send_now()`. Messages having unannounced destinations remain buffered until a later beacon interval.

3 Changes to Existing 802.11 Implementation

In the course of adding the management features described in Section 2, we identified several aspects of the existing code which could be improved or updated. The Monarch *ns* code implemented the original 1997 version of 802.11 [12], which supported a peak channel rate of 2Mbps. The newer 802.11b [13] increased that rate to 11Mbps, and is far more common at the time of this writing. We added support for the newer standard, and implemented the PHY Layer Convergence Protocol in greater detail. We also corrected several instances of wrong updates to the MAC backoff timer and Contention Window.

[2]The Monarch code ships with the parameter aRTSThreshold (§11.4.4.2.15) set to zero, meaning that *all* directed ATIM and data frames are retried up to aLongRetryLimit (§11.4.4.2.17). The specification recommends a value of 3000 bytes for aRTSThreshold.

3.1 802.11b High-Rate PHY

Proper support for the 802.11b high-rate PHY requires more than simply changing the channel rate constant from 2Mbps to 11Mbps. An 802.11b IBSS which retains backwards compatibility with 802.11 stations must transmit at a number of distinct channel rates. Section 2.1.2 listed the *basic* and *supported* rates for our implementation. The rules for the use of these rates are as follows (§9.6):

- All control frames (RTS, CTS, ACK) must be transmitted at one of the rates in the basic rate set. Our implementation uses 2Mbps.
- All multicast or broadcast frames, regardless of type, must be transmitted at one of the rates in the basic rate set. Again, we use 2Mbps.
- Directed data or management frames may be sent at any supported rate. Our implementation uses 11Mbps.

Further complicating matters is the PHY Layer Convergence Protocol — described in Section 3.2 — which adds a preamble and header to all transmitted frames. The header and preamble are transmitted at 1Mbps (§15.2.3). This is a correction from the original Monarch code, which sent the *entire* frame at the prevailing channel rate. A consequence of the original behavior was that the simulator understated the amount of time required to transmit a frame.

Several macros have been defined in *cmu/mac-802_11.h* to compute frame transmission times. `BASIC_RATE(len)` and `OPTIONAL_RATE(len)` return the amount of time required to send a frame of size `len` octets at 2Mbps and at the prevailing channel rate (11Mbps), respectively.

3.2 PHY Layer Convergence Protocol

For completeness, our implementation adds a PHY Layer Convergence Protocol (PLCP) header to each frame (§15.2.3). This header describes the length of the frame, and the bit rate at which the frame is being transmitted. The `struct dsss_plcp_frame` contains placeholders for the PLCP preamble (§15.2.3.1) and CRC (§15.2.3.6). It also contains the Start Frame Delimiter, Signal, Service, and Length fields (§15.2.3.2–15.2.3.5).

Of special interest is `Mac802_11::TX_Time(p)`, which now computes the amount of time required to transmit packet `p` using the value stored in the PLCP Length field. This value is set using the frame transmission time macros described in Section 3.1. As a result, the transmission timer `Mac802_11::mhIF` is set more accurately than in the original code.

3.3 Corrections to 802.11 Implementation

In the course of reviewing the 802.11 specification, we corrected several minor errors in the Monarch code. The first has to do with the station retry counters (§9.2.4), which are to be reset to zero following the transmission of a broadcast frame. The existing code was not performing the reset; this has been fixed.

There are two station retry counters, SSRC for frames up to aRTSThreshold bytes in size, and SLRC for longer frames. SSRC is to be reset to zero when an ACK for a frame of *any* size is received. Additionally, SLRC is to be reset to zero if the frame which elicited the ACK was longer than aRTSThreshold (§9.2.4). The existing code would reset one counter or the other, but never both; this has also been fixed.

Finally, the Contention Window parameter (§9.2.4) determines the amount of random backoff a station observes prior to a frame transmission attempt. This parameter increases in size every time a transmission attempt *fails*. Specifically, it should only increase when one of the station retry counters is incremented. The existing code contained unnecessary increases of the Contention Window in `Mac802_11::check_pktRTS()` and `Mac802_11::check_pktTx()`, which are called prior to the *first* transmission attempt for an RTS or data frame, respectively. This can increase the duration of the random backoff before any failures have been observed. These superfluous increases have been removed.

4 Nonstandard 802.11 Enhancements

We have experimented with several changes to the behavior of 802.11 under power management. Some of these modifications address latency issues arising from the traffic announcement facility described in Section 2.2.1. Others were attempts to improve aspects of 802.11 which yield conspicuously poor behavior in multihop environments. By default, these enhancements are disabled, but can be individually selected using `Mac802_11::set_options()`. This method accepts the options descriptor shown in Figure 6. The five `mac_options` are described in Sections 4.1–4.5.

```
struct mac_options {
        bool pm_suspension;
        bool fast_wakeup;
        bool no_beacon_keepawake;
        bool atims_after_abandoned_beacon;
        bool bcast_atim_implies_awake;
};
```

Figure 6: The MAC options descriptor.

4.1 Power Management Suspension

The 802.11 IBSS power management design introduces latency to message transmission by buffering frames until their destinations can be announced. In a multihop environment, these delays accumulate, and can add several seconds to end-to-end delivery latency. Several proposals to reduce latency under power management have been published [1, 3, 14]. These all work by *suspending* the use of power management at some nodes. The idea is that if station A knows that station B is awake (*i.e.*, not using power management), then A can immediately send frames to B without waiting for the announcement process.

There are several ways by which a station can learn that its neighbor is suspending power management. Exact methods are beyond the scope of this report, but in general the techniques involve explicit negotiation or inference based on observed message activity. Section 5 presents two examples of the latter type.

Our implementation supports four methods relevant to power management suspension. These are enabled when the `pm_suspension` option is set:

- `Mac802_11::suspend_pm()` causes the station to never doze at the end of an ATIM window. All outgoing frames have the Power Management bit (§7.1.3.17) of their Frame Control field cleared after this method is invoked.
- `Mac802_11::resume_pm()` restores the use of the *doze* state to the normal 802.11 behavior.
- `Mac802_11::neighbor_suspend_pm(nsaddr_t neighbor)` adds the MAC-layer `neighbor` to a table of destinations for which ATIM announcements are not generated, and to which frames may be transmitted immediately.

- `Mac802_11::neighbor_resume_pm(nsaddr_t neighbor)` causes the removal of `neighbor` from the table of suspending neighbors. The named `neighbor` is removed at the beginning of the next ATIM window.

4.2 Fast Wakeup

Power management suspension (Section 4.1) is a means of reducing the latency of message transmission to stations which are known to be active. In an on-demand routing protocol such as Dynamic Source Routing [9], the identities of awake neighbors may not be known in advance of some operations. For example, a DSR traffic source discovers routes to traffic sinks by *flooding* a Route Request message through the network. Using the 802.11 MAC, this flooding is implemented by propagating broadcasts of the Route Request.

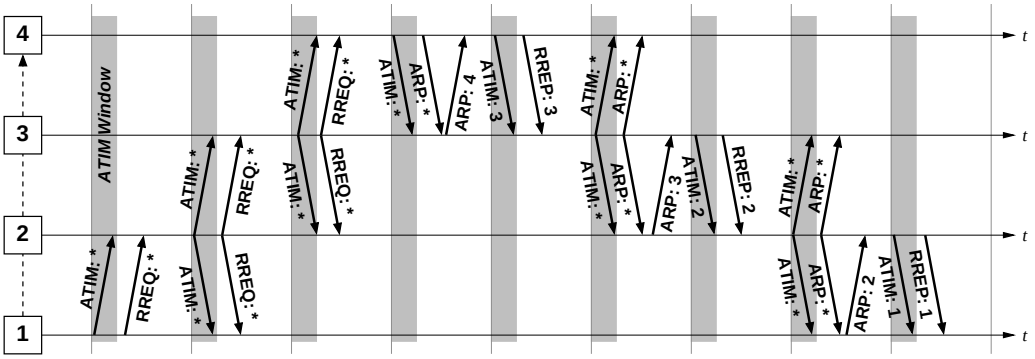


Figure 7: Route Discovery using IBSS power management.

Figure 7 shows the interaction between IBSS power management and the Route Discovery procedure. Source 1 is discovering the route (1, 2, 3, 4). At each hop, the broadcast Route Request ("RREQ") is delayed while the propagating station announces the broadcast address with an ATIM. Then, as the unicast Route Reply ("RREP") messages are returned, each station on the route may need to perform an ARP transaction to learn the MAC-layer address of the next hop. Both the broadcast ARP request and the directed Route Reply transmission in-

duce additional ATIM delays. If the beacon interval between ATIM windows is hundreds of milliseconds in duration, then this Route Discovery can require several seconds. This discovery latency is several orders of magnitude greater than that experienced *without* power management.

Fast wakeup is a method for reducing this latency by applying a DSR concept to the MAC layer. When the `fast_wakeup` options is set, a received broadcast ATIM triggers a priority transmission of a broadcast ATIM. In other words, broadcast ATIM frames propagate quickly through the network using a controlled flooding technique similar to Route Request propagation.

Algorithm 1 HANDLE-ATIM(a)

```
if address(a) = Broadcast then
  rcvd-bcast-atims ← rcvd-bcast-atims + 1
  if sent-bcast-atims = 0 then {controlled flooding}
    if rcvd-bcast-atims = 1 then
      SET-ATIM-HOLDOFF {random delay}
    ATIM-ENQUEUE(address(a))
```

Algorithm 1 shows the additional handling of a received ATIM frame a when fast wakeup is enabled. When a station receives its first broadcast ATIM frame of the current ATIM window, if it has not already transmitted a broadcast ATIM, it schedules a broadcast ATIM for transmission. The transmission will be delayed by a random time determined in `Mac802_11::set_atim_holdoff()`.

The new broadcast ATIM frame is scheduled by manually adding an address to the `ATIMQueue` using `ATIMQueue::announce()`. Such addresses are announced with higher priority than those obtained from the `PMQueue`. The idea is to allow the broadcast ATIMs to propagate as far across the network as possible during an ATIM window. Following the window, the DSR Route Request can propagate as far as the broadcast ATIMs were able to reach. This dramatically reduces Route Discovery latency under power management [3]. The method does

not require syntactic changes to any 802.11 frame, and is backwards-compatible with 802.11 implementations that do not support fast wakeup.

4.3 Beacon Generation and Dozing

Section 2.2.2 listed the conditions under which a station could doze at the end of an ATIM window. Stations are not permitted to doze if they have transmitted a beacon frame during the current beacon interval (§11.2.2.3). In the single-hop environment for which IBSS mode was designed, this condition might make sense: at most one station should "win" the beacon contention procedure. In a multi-hop network, the general case is that *many* stations generate beacons. This is affected by station density; the fewer neighbors a station can hear, the more likely it is to "win" the contention procedure and transmit a beacon.

The problem is that stations which transmit beacons often — for example, due to their position in a sparse network — will remain awake in relatively more beacon intervals. In the limiting case, a station which cannot hear any neighbors will generate a beacon in every beacon interval, and *never* doze. This behavior can lead to very high energy consumption *independent* of (data) traffic activity.

When the `no_beacon_keepawake` option is set, this requirement is ignored. Stations can enter doze even if they transmit a beacon, as long as they do not need to remain awake for some other reason (*e.g.*, they have transmitted ATIM frames). When this option is set, a neighboring station is *not* assumed to be awake based solely on its transmission of a beacon in the current beacon interval. (Normally, such a transmission is sufficient evidence of awakeness.) This option can substantially reduce station energy consumption, especially in sparse or idle networks.

4.4 Releasing ATIM Frames After Abandoned Beacons

Another aspect of the IBSS design which is reasonable in single-hop environments, but which breaks down in multi-hop networks, is the condition under which ATIM frame transmission may commence. ATIMs are to be transmitted "following the reception or transmission of the beacon, during the ATIM window" (§11.2.2.4). Consider the classic hidden terminal problem of Figure 8, in which 1, 2, and 3 are contending to transmit a beacon following a TBTT. Suppose that 1 and 3 transmit at the same time, and their beacons collide at 2, preventing 2 from receiving either beacon. If 2 attempts to transmit its beacon, but finds the medium to be busy, the specification defines no method for recovery (§11.1.2.2). In such a circumstance, a station must *abandon* its transmission attempt.

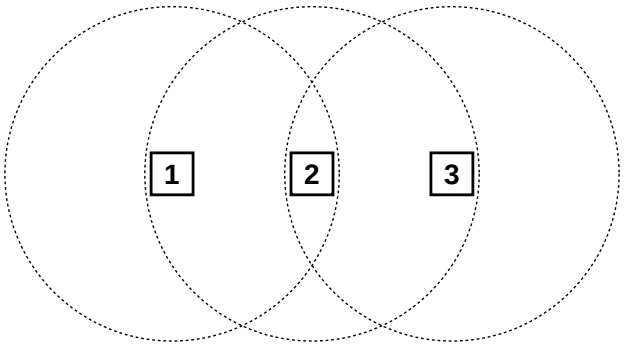


Figure 8: Hidden terminal problem.

The problem occurs when a station receives no beacons (as in the case of collision), and must abandon its own beacon transmission. The station therefore fails to meet the criteria for releasing ATIM frames, and cannot announce any of its pending destinations. This can lead to underutilization of the medium. When the `atims_after_abandoned_beacon` option is set, ATIM transmission commences after a beacon transmission is attempted, but abandoned. This option can result in some stations receiving an ATIM frame before they have sent or received a beacon frame.

4.5 Broadcast ATIM Frames and Directed Transmissions

Finally, an optimistic enhancement to the traffic announcement facility assumes that a broadcast ATIM transmission will wake all neighboring stations. When the `bcast_atim_implies_awake` option is set, a neighbor is taken to be awake if a broadcast ATIM was sent in the current beacon interval, and a directed ATIM to that neighbor did *not* fail in the current beacon interval. (The latter suggests that the neighbor has moved out of range; Section 2.2.4 describes handling of failed directed ATIMs.) This option is optimistic because congestion in the ATIM window may prevent some stations from receiving a broadcast ATIM, causing them to enter *doze* following the window. Directed transmissions to such stations will then fail, even if the stations are within range of the transmitting station.

5 Cross-Layer Power Management

The power management suspension technique (Section 4.1) requires stations to know the power management status of their neighbors. We provide some simple examples of how stations can use information at higher layers — such as the DSR layer — to infer the status of nearby stations. These examples implement the abstract class `PM`, which defines the interface used by `DSRAgent` to pass events to the power management logic.

Figure 9 illustrates the interaction between the existing protocol stack and the cross-layer power management examples. Generally, DSR stimulates the power management instance by signaling events such as packet reception. Power management then performs operations on the lower layers, such as suspending MAC power management. The actions shown in *(parentheses)* are not used by the examples described in this Section. They are explained in [3].

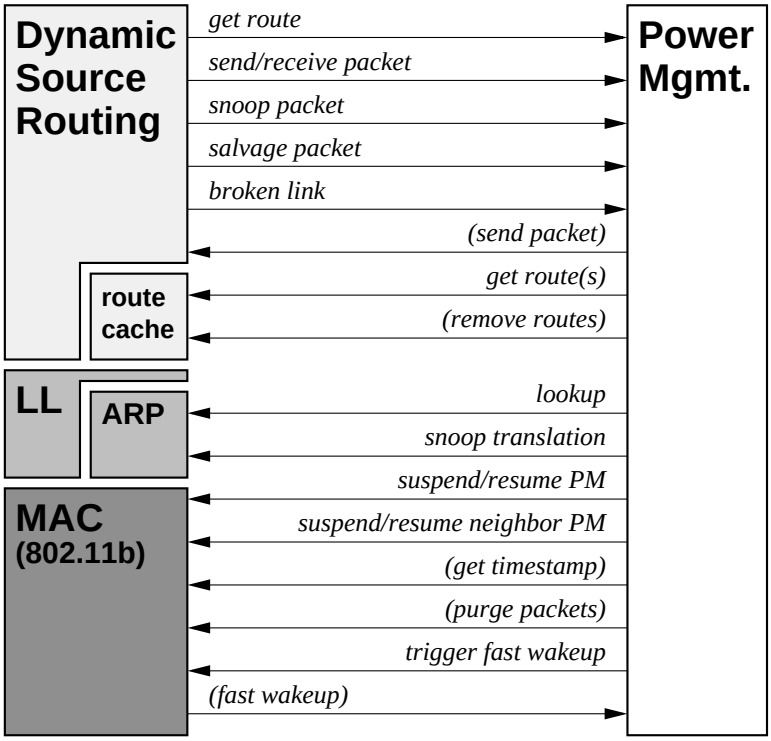


Figure 9: Power management layer interaction.

Sections 5.1 and 5.2 describe instances of the "Power Mgmt." interface shown in Figure 9. Both designs suspend node power management upon processing certain messages. For example, the receipt of a CBR message might be taken as a signal that a node is an active participant in some route, and should therefore remain awake. The previously-published approach of Section 5.1 uses MAC-layer snooping to infer the power management status of neighboring stations. The design of Section 5.2 uses higher-layer message events, and incorporates all of the optimizations from Section 4.

When one of these power management instances is configured, we modify the pruning rule used by DSR to control the flood of Route Request messages. The goal is to increase the *diversity* of routes discovered by DSR [3]. Normally, when Request $\langle P, n \rangle$ arrives on route $P = \left(p_1, p_2, \ldots, p_{|P|}\right)$ with sequence number n, it is only accepted if no previous Request from p_1 also had sequence n.

Our modification compares the route P against all previous Requests from p_1 with sequence n. Let $\langle Q, n \rangle$ be a previous Request with $Q = \left(p_1, q_2, \ldots, q_{|Q|}\right)$. The Request $\langle P, n \rangle$ is accepted if $\forall \langle Q, n \rangle : \frac{|Q \cap P|}{|Q|} \leq T$, where $0 < T \leq 1$ is a diversity threshold. T determines the maximum commonality that may exist between P and any previously-seen route with sequence n. In our implementation, $T = \frac{3}{4}$ seems to do a good job of pruning short Requests near the source (which reduces congestion), while encouraging diversity at more distant nodes.

5.1 On-demand Power Management

On-demand Power Management [14], or ODPM, is a cross-layer design which uses uncoordinated timers at nodes to configure MAC power management. We corresponded with the primary author of the 2003 ODPM paper to clarify some design points described therein. We also reviewed her implementation, originally developed for a different version of *ns*. We then implemented `OndemandPM` as an instance of the `PM` interface, since ODPM is an instance of the general architecture of Figure 9.

Additionally, `OndemandPM` implements the `FrameMonitor` interface, which allows the MAC layer to pass information about received frames to the power management instance. In particular, ODPM uses the Power Management bit (§7.1.3.17) in the Frame Control field of received frames to infer the power management status of neighboring stations. This bit, along with the address of the sending station, is passed to a `FrameMonitor` instance such as `OndemandPM`, which then updates its list of suspending neighbors.

Several concepts are described in the paper [14] which were not implemented in the code made available by the paper author. These include the use of peri-

odic "HELLO" messages, and neighbor state inference based on node degree and beacon frames. Our implementation matches the author's code, not the paper, in these instances.

Timer	Duration
Route Request keepalive	0s
Route Reply keepalive	5s
CBR message keepalive (source)	2s
CBR message keepalive (relay)	2s
CBR message keepalive (sink)	2s
Refresh interval	5s

Table 1: On-demand Power Management timer durations.

We adopted the timer values described in the paper, which are reproduced in Table 1. The refresh interval is not implemented as a timer, but rather is used when a transmission failure occurs. When a station sends an RTS frame but does not receive a CTS from its neighbor, it checks the suspending neighbor list. If the neighbor has not been heard from in longer than the refresh interval, then one of two things happens. If the neighbor was previously suspending power management, it is considered to have resumed power management. Otherwise, its list entry is removed.

The code uses a different value for the refresh interval: 900 seconds. This value means "forever" in many simulators based on the Monarch *ns* distribution. We used the value of 5 seconds from the paper in our implementation.

`OndemandPM` configures only one of the `mac_options` described in Section 4: power management suspension. As a result, it experiences lower average-case message delivery latency than if standard 802.11 IBSS power management were used. Route Discovery latency and energy profile are similar to IBSS power management, however [3].

5.2 Local Power Management

Local Power Management [3], or LPM, takes a different approach to uncoordinated, timer-based power management. Rather than having to observe *every* frame received from neighboring stations in order to update power status estimates (as in ODPM), LPM relies only on those messages normally processed by DSR. Additionally, LPM uses all five `mac_options` from Section 4, which improves Route Discovery latency and energy performance.

LPM uses a table of `ActiveRoute` instances to implement the concept of active destinations. When DSR requests a route to a new destination, `LocalPM` constructs a new `ActiveRoute` which will enforce consistency in the routes used for messages to that destination. DSR normally chooses the route for each outgoing message independently. We would like the same route to be used for as long as possible, so that the nodes *on* the route know to remain awake, and nodes *not on* the route can return to *doze*.

As soon as a route to the new destination is discovered, it is loaded into the `ActiveRoute` and is used consistently until the route is *finalized*. Finalization occurs `PM_FINALIZE` seconds after the new `ActiveRoute` is constructed. At this time, the DSR route cache is checked again for the cache-preferred route, which then becomes the new route used for subsequent messages. This two-stage process gives Route Discovery enough time to find several routes to the destination. By the time of finalization, enough routes should be known that a "good" one can be chosen for subsequent use. After an `ActiveRoute` has not been used for `PM_ACTIVE_TIMEOUT` seconds, then the active destination state is torn down.

Timer	Duration
Active route finalize	250ms
Active route timeout	500ms
CBR PM suspension	500ms
DSR PM suspension	500ms

Table 2: LPM timer durations.

Table 2 shows the `ActiveRoute` timer values used in our implementation. It also shows the duration for which LPM suspends power management upon processing CBR or DSR messages. Such messages result in updates to a `Suspend` table which tracks the power management suspension timers associated with a node and its neighbors. A separate timer is maintained for each message type; a node resumes power management when all of its timers have expired.

Because `LocalPM` deals only with messages processed at the DSR layer, it needs to know IP-to-MAC address translations so that it may configure neighbor status at the MAC layer. Our implementation exposes the `ARPTable` to `LocalPM` both for lookup *and* update.

Each DSR message bears the IP address of the most recent node to transmit the message. DSR messages are of course sent as the payload of a MAC-layer data frame. We assume that DSR has access to the MAC frame and its headers; this is reasonable for a kernel-mode network-layer implementation. Each DSR message either received or overheard by a node therefore contains both a source route header and a MAC header. These headers provide an IP-to-MAC translation for the address of the transmitting node.

Given the information provided in each DSR message, we implement an obvious Address Resolution Protocol (ARP) snooping technique. For every DSR message received or overheard by a node, we insert an address translation into the ARP cache for the sender of that message using `ARPTable::snoop()`.

ARP snooping eliminates the need for explicit ARP requests. A DSR node never transmits a directed data frame to a neighbor from which it has not previously *received* at least one frame. For example, during Route Discovery, nodes send broadcast Route Requests (which do not require address translation) before sending directed Route Replies. A node which transmits a Route Reply to a neighbor *necessarily* has received a Route Request from that neighbor, and therefore has already added an address translation for the neighbor to its ARP cache.

6 Power Model

To support experimentation with different radio power models, our implementation applies specific power rates to the states described in Section 2.2.2 in post-processing. The simulator itself only records the times each simulated wireless network interface spends in each state. Subsequent analysis can then explore the effects of different power profiles on overall energy consumption, without having to re-run the simulator.

We provide two tools to help study power state behavior. The first simply adds up all the time spent by an interface in each state, and outputs these totals at the end of a simulation. This approach minimizes filesystem activity and permits straightforward post-processing. The other tool outputs state transition times for each interface as the simulator runs. This is helpful for visualizing the power behavior of interfaces, but imposes higher demands on the filesystem.

6.1 Power State Total Occupation Times

The 802.11 implementation records power state transitions using the `Power` abstract class. By default, a `PowerTotal` instance adds up the times spent in

each of the seven states shown in Figure 3. At the end of the simulation, in `Mac802_11::reset()`, `PowerTotal::stop()` is called to write out the time totals to a log named by the `-powerlog` option described in Section 8.1.

```
1       0.000000        93.602326       0.149500        \
        0.149500        567.428837      219.259446      \
        19.410392       900.000000
2       1.000000        150.308347      0.240250        \
        0.240000        520.309069      217.589877      \
        10.312457       900.000000
3       1.000000        180.052388      0.287750        \
        0.287500        494.533904      216.422052      \
        7.416407        900.000000
```

Figure 10: `PowerTotal` log excerpt.

Figure 10 shows an excerpt from the log data produced by `PowerTotal`. The file is tab-delimited, and lists for each node (named in the first field) the times spent in the *off*, *doze*, *to-doze*, *from-doze*, *idle*, *receive*, and *transmit* states. The ninth field is the sum of state times, which as a sanity check should add up to the total simulation time.

A simple script to read in a `PowerTotal` log and compute total per-node energy consumption is shown in Figure 11. The `%powerlevels` hash contains measured power rates, in Watts, for the popular WaveLAN 802.11b PCMCIA device [7]. Previous work has cited power rates for the *to-doze* and *from-doze* states that are twice that of the *idle* state [10]. Our example rates reflect this relationship.

6.2 Power State Transition Traces

It is sometimes useful for illustrative purposes to know the timeline of power state transitions, rather than simply the total occupancy times. An optional `Power` implementation, `PowerTrace` records every transition by every simu-

```
#!/usr/bin/perl

while (@ARGV) {

    if ($ARGV[0] =~ /(.+)\.log/) {
        open(LOG, $ARGV[0]) or die "Can't open log: $!\n";
        $logname = $1;
        last;
    }

    shift;

}

open(DAT, ">$logname.dat");

%powerlevels = ( "o" => 0.000 * 4.74,           # off
                 "d" => 0.010 * 4.74,           # doze
                 "s" => 0.156 * 4.74 * 2,       # to-doze
                 "w" => 0.156 * 4.74 * 2,       # from-doze
                 "i" => 0.156 * 4.74,           # idle
                 "r" => 0.190 * 4.74,           # receive
                 "t" => 0.284 * 4.74 );         # transmit

$v = '\t([\d\.]+)';

while ($line = <LOG>) {

    if ($line =~ /^([\da-f]+)$v$v$v$v$v$v$v/) {

        printf DAT "%llx\t%f\n", hex($1),
                $2 * $powerlevels{"o"} +
                $3 * $powerlevels{"d"} +
                $4 * $powerlevels{"s"} +
                $5 * $powerlevels{"w"} +
                $6 * $powerlevels{"i"} +
                $7 * $powerlevels{"r"} +
                $8 * $powerlevels{"t"};

    }

}

close(DAT);
close(LOG);
```

Figure 11: `PowerTotal` post-processing example (*cmu/scripts/powertotal.pl*).

lated wireless interface in a single log file. Specific power rates can be applied in post-processing to produce power-*vs.*-time curves for each interface.[3] The use of `PowerTrace` is enabled in *cmu/mac-802_11.cc* by defining the preprocessor macro `USE_POWERTRACE`.

```
1.202791        1b      i
1.202804        4       i
1.204502        1       t
1.204502        1b      r
```

Figure 12: `PowerTrace` log excerpt.

Trace output is written continuously during the simulation to a log named by the `-powerlog` option described in Section 8.1. Figure 12 shows an excerpt from a `PowerTrace` log. The file is tab-delimited, and lists (simulator) time, node address, and power state. Addresses are expressed in hexadecimal, and power states are encoded with a single character as shown in Table 3.

Code	State
o	*off*
d	*doze*
s	*to-doze* (mnemonic: "**s**leep")
w	*from-doze* (mnemonic: "**w**ake")
i	*idle*
r	*receive*
t	*transmit*

Table 3: `PowerTrace` power state encodings.

The `PowerTrace` log output contains transition data for each node interleaved with that of all other nodes. Figure 13 shows a simple script to read in a trace and write out separate power curves for each node. The script is concise, but is wasteful of memory. This and similar post-processing tools should generally be used with short simulations involving small numbers of nodes, due to the potentially high volume of log output generated by `PowerTrace`.

[3] Total interface energy consumption can be derived by integrating these curves. This method is costly in terms of filesystem activity, and was the motivation for the more efficient `PowerTotal`.

```
#!/usr/bin/perl

while (@ARGV) {

    if ($ARGV[0] =~ /(.+)\.log/) {
        open(LOG, $ARGV[0]) or die "Can't open log: $!\n";
        last;
    }

    shift;

}

while ($line = <LOG>) {

    if ($line =~ /^([\d\.]+)\t([\da-f]+)\t(.)/) {

        push @{ $states{$2} }, [ $1, $3 ];

    }

}

close(LOG);

%powerlevels = ( "o" => 0.000 * 4.74,           # off
                 "d" => 0.010 * 4.74,           # doze
                 "s" => 0.156 * 4.74 * 2,       # to-doze
                 "w" => 0.156 * 4.74 * 2,       # from-doze
                 "i" => 0.156 * 4.74,           # idle
                 "r" => 0.190 * 4.74,           # receive
                 "t" => 0.284 * 4.74 );         # transmit

foreach $node ( sort {$a <=> $b} keys %states ) {

    open(DAT, ">power_$node.dat");

    foreach $i ( 0 .. $#{ $states{$node} } ) {

        print DAT "$states{$node}[$i][0]\t" .
            "$powerlevels{$states{$node}[$i][1]}\n";

    }

    close(DAT);

}
```

Figure 13: `PowerTrace` post-processing example (*cmu/scripts/powertrace.pl*).

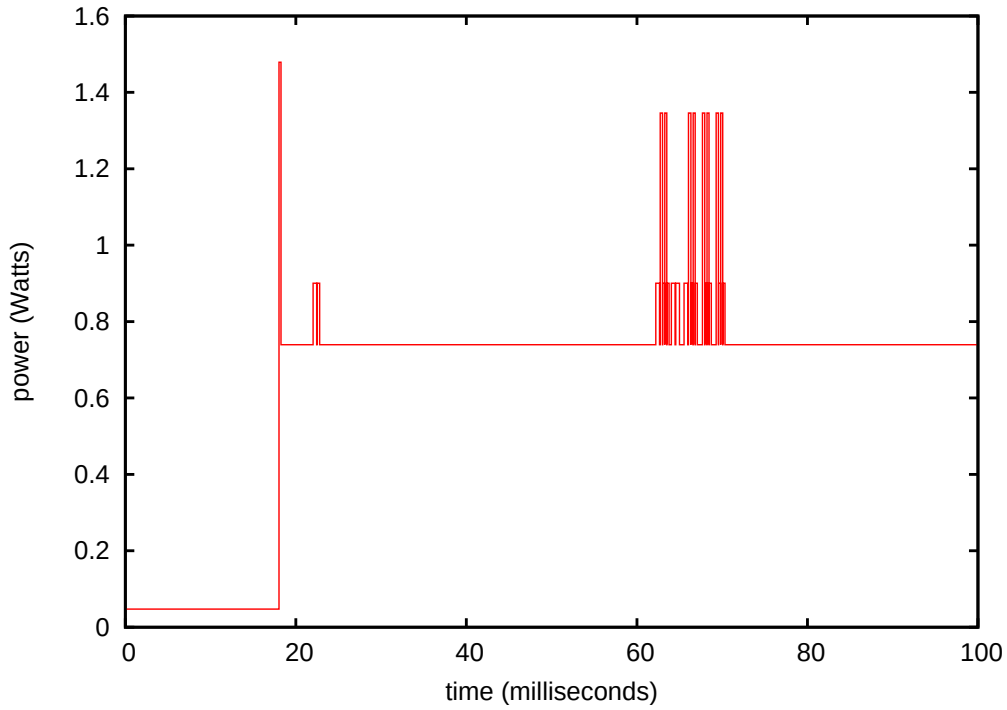


Figure 14: Power trace showing doze, wake, and frame exchanges.

Figure 14 shows an example power curve produced by post-processing a `PowerTrace` log. Just before 20 milliseconds, the interface wakes from doze, and then receives some beacon frames. Following a 40-millisecond ATIM window, the node remains awake to exchange several control and data frames.

7 Building the Simulator

Our improvements were implemented against version 1.1.2 of the Monarch *ns* extensions. We provide two patches which update the Monarch code base: one to provide compatibility with new platforms and tools, the other to provide functional improvements such as 802.11 IBSS power management. The `-jd8` patch updates the build scripts and implements syntactic corrections to support Mac OS X (Darwin) and the Intel C++ Compiler for Linux. The `-jd8.3` patch augments the first patch by implementing 802.11 IBSS mode, IBSS power management, cross-layer power management, and other features.

```
$ ls
cmu-extendedns-1.1.2.tar.gz
cmu-extendedns-1.1.2-jd8.patch.gz
cmu-extendedns-1.1.2-jd8.3.patch.gz
$ mkdir cmu-extendedns-1.1.2-jd8.3
$ cd cmu-extendedns-1.1.2-jd8.3
$ tar zxf ../cmu-extendedns-1.1.2.tar.gz
$ gunzip -c ../cmu-extendedns-1.1.2-jd8.patch.gz |  \
      patch -p1
(Successful patch output.)
$ gunzip -c ../cmu-extendedns-1.1.2-jd8.3.patch.gz |  \
      patch -p1
(Successful patch output.)
Make powertotal.pl and powertrace.pl scripts executable:
$ find ns-src -name "*.pl" -exec chmod +x
$ cd ns-src
Replace autoconf scripts with ones that know about Darwin:
$ cp /usr/share/libtool/config.* .
$ autoconf
Assumes Tcl/Tk 8.4:
$ ./configure --disable-static --with-tcl=/sw \
      --with-tcl-ver=8.4 --with-tk=/sw --with-tk-ver=8.4 \
      --with-otcl=/sw --with-tclcl=/sw
(Successful configure output.)
$ make depend
(Successful make output.)
$ make
(Successful make output.)
$ cd ../ad-hockey
$ make
(Successful make output.)
```

Figure 15: Unpacking, patching, configuring, and building *ns* on Darwin.

Figure 15 illustrates the build process on Mac OS X (Darwin). The patches and process are correct as of Mac OS X 10.3.5, using the *Fink* installation of Tcl/Tk 8.4 under */sw*. During the pre-compilation process, the GNU *autoconf* environment must be updated using vendor-supplied scripts that know about Darwin. The simulator is configured to use installations of *OTcl* and *TclCL* which are not typically found in vendor package systems. Installation of these components is beyond the scope of this report.

The build process on Linux is similar to that shown in Figure 15. One difference is that no *autoconf* scripts need be copied. Also, as of SuSE Linux 8.1, the *configure* script produced by *autoconf* is not executable by default. Finally, to use the Intel C++ Compiler (as of version 7.0), some environment variables must be configured. All of these steps are addressed by removing the *cp* command in Figure 15, and inserting the following commands after the invocation of *autoconf*:

```
$ chmod +x configure
$ export CC=icc CXX=icc LD=icc
```

8 Using the Simulator

The patched version of *ns* is used in the same manner as the original Monarch version. Some new features are always available. These include the use of the 802.11b high-rate PHY (Section 3.1), the improved PLCP implementation (Section 3.2), and the corrections to 802.11 retry counters and Contention Window usage (Section 3.3). Most other improvements are configurable, and are disabled by default. Running the patched simulator with unmodified (from the original simulator) command-line options should produce results that are similar to the original. Any variations should result from the aforementioned features, and the presence of IBSS beacons, which are enabled by default.

8.1 Power Management Options

Several new options are available in the patched simulator to configure IBSS mode, power management, and logging. Table 4 lists the options and summarizes their usage. We note that beacon frames can be disabled by passing "`-ibss 0`," which produces behavior more similar to the original simulator.

Also, when power management is enabled using "`-atim`," "`-llq NullQueue`" should also be passed to provide the queuing behavior described in Section 2.2.3.

Option	Usage
`-ibss` *beaconinterval*	Activates beacons and specifies the interval, in milliseconds, between their transmission. A beacon interval of `0` disables beacons. Default value: `200`.
`-atim` *window*	Enables IBSS power management and specifies the length of the ATIM window, in milliseconds. (Note: also specify `-llq NullQueue`.) A window length of `0` disables power management. Default value: `0`.
`-pm` *crosslayer*	Enables a cross-layer power management instance for all nodes, such as `PM/OndemandPM` or `PM/LocalPM`. Default value: disabled.
`-dsrlog` *file*	Enables `DSRLOG()` logging in `DSRAgent`, writing the output to *file*. Default value: disabled.
`-maclog` *file*	Enables `LOG()` logging in `Mac802_11`, writing the output to *file*. Default value: disabled.
`-pmlog` *file*	Enables `LPMLOG()` logging in `LocalPM`, or `ODPMLOG()` logging in `OndemandPM`, writing the output to *file*. Default value: disabled.
`-powerlog` *file*	Enables power state logging in `PowerTotal` or `PowerTrace`, writing the output to *file*. Default value: disabled.

Table 4: *ns* options for IBSS mode, power management, and logging.

The cross-layer power management schemes described in Section 5 are disabled by default, but can be configured using the "`-pm`" option. Only one such scheme can be used in a simulation run. When configured, all nodes participate in the scheme using the same parameters.

The original Monarch *ns* implementation logs significant events, such as frame transmissions or `DSRAgent` actions, to a trace file named by "`-tr`." For debugging purposes, it can be useful to have more comprehensive log data separated by functional module. The "`-dsrlog`," "`-pmlog`," and "`-maclog`" options enable logging for the DSR, cross-layer power management, and MAC code, respectively. Some of these, particularly the MAC logging facility, generate significant output, which may affect simulator performance.

Original example:

```
$ ./ns cmu/scripts/run.tcl -rp cmu/dsr/dsr.tcl \
      -x 1500 -y 300 -cp scen/cbr-50-20-4-512 \
      -sc scen/scen-1500x300-50-0-20-1 -stop 900 \
      -tr out.tr
```

With power state logging:

```
$ ./ns cmu/scripts/run.tcl -rp cmu/dsr/dsr.tcl \
      -x 1500 -y 300 -cp scen/cbr-50-20-4-512 \
      -sc scen/scen-1500x300-50-0-20-1 -stop 900 \
      -tr out.tr -powerlog power.log
```

Using IBSS power management with 40ms ATIM window:

```
$ ./ns cmu/scripts/run.tcl -rp cmu/dsr/dsr.tcl \
      -x 1500 -y 300 -cp scen/cbr-50-20-4-512 \
      -sc scen/scen-1500x300-50-0-20-1 -stop 900 \
      -tr out.tr -powerlog power.log \
      -llq NullQueue -atim 40
```

Using Local Power Management:

```
$ ./ns cmu/scripts/run.tcl -rp cmu/dsr/dsr.tcl \
      -x 1500 -y 300 -cp scen/cbr-50-20-4-512 \
      -sc scen/scen-1500x300-50-0-20-1 -stop 900 \
      -tr out.tr -powerlog power.log \
      -llq NullQueue -atim 40 -pm PM/LocalPM
```

Figure 16: Running *ns*.

Figure 16 shows some examples of running the patched *ns* simulator. The examples are based on the one given in [2], which is provided for reference. Note that in the final example using Local Power Management, the options for configuring 802.11 IBSS power management must still be used.

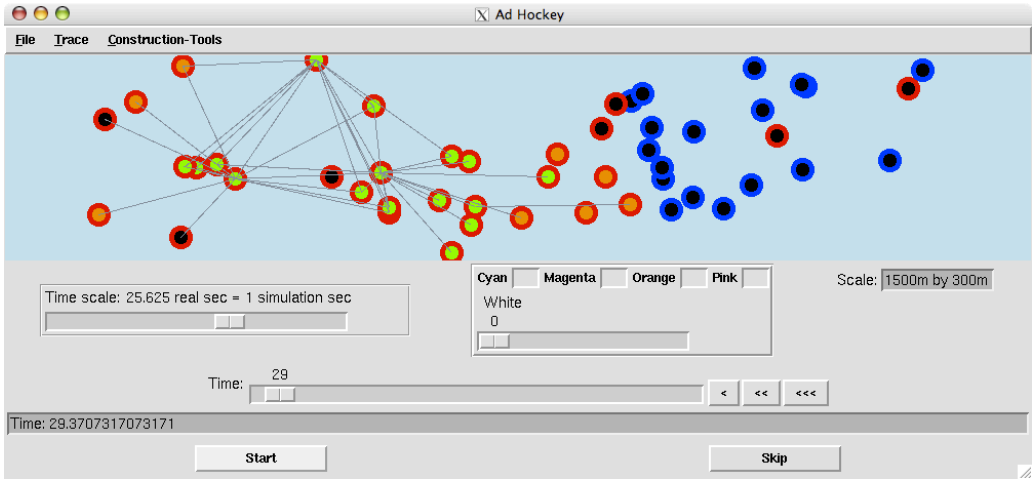
8.2 Ad-Hockey

Monarch *ns* includes a visualization tool, *ad-hockey*, which can be helpful in understanding the behavior of a mobile network. We have augmented *ad-hockey* with the ability to display node power states. This feature requires the use of `PowerTrace` (Section 6.2). Figure 17 shows how the power state data is incorporated into the *.viz* file produced by *viz-trace* using the "`-p`" option. The *ad-hockey* program is then run as usual.

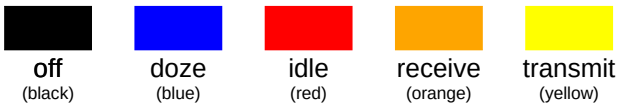
```
$ cd ../ad-hockey
```
Combine communications and power events in out.tr.viz:
```
$ ./viz-trace -p ../ns-src/power.log ../ns-src/out.tr
$ ./ad-hockey ../ns-src/scen/scen-1500x300-50-0-20-1 \
      ../ns-src/out.tr.viz
```

Figure 17: Using *ad-hockey* with power state display.

Figure 18 shows *ad-hockey* displaying power states for each node. The circular outline of each node changes color with its state. Nodes in the *off* state have a black outline, *doze*, *to-doze*, and *from-doze* nodes are blue, *idle* nodes are red, nodes in *receive* are orange, and nodes in *transmit* are yellow. Due to the way display events are scheduled in *ad-hockey*, it can be difficult to catch a particular node in the receive or transmit states. As Figure 18 shows, it is much easier to spot the difference between idle and dozing nodes.[4]



(a) The *ad-hockey* visualization tool.



(b) Power state outline color key.

Figure 18: *Ad-hockey* with interface power state indication.

[4]Note the small number of *idle* nodes distributed among the dozing nodes towards the "right" side of the network. These nodes won the beacon contention algorithm and must remain awake irrespective of traffic load. This behavior was discussed in Section 4.3.

Acknowledgments

We are very grateful to Dave Maltz for his guidance in developing these extensions. We recognize Dave Maltz and Josh Broch for their work on the original Monarch *ns*, without which this effort would not have been possible. David Johnson provided invaluable advice, and motivated the inclusion of On-demand Power Management in this release. Finally, Tom Martin's suggestions significantly shaped the power state measurement facility.

References

[1] CHEN, B., JAMIESON, K., BALAKRISHNAN, H., AND MORRIS, R. Span: An Energy-Efficient Coordination Algorithm for Topology Maintenance in Ad Hoc Wireless Networks. In *Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking (MobiCom '01)* (Rome, Italy, 2001), pp. 85–96.

[2] COMPUTER SCIENCE DEPARTMENT, CARNEGIE MELLON UNIVERSITY. *The CMU Monarch Project's Wireless and Mobility Extensions to* ns. Pittsburgh, Pennsylvania, Aug. 1999.

[3] DORSEY, J. G. *Game-Theoretic Power Management in Mobile Ad Hoc Networks*. PhD thesis, Carnegie Mellon University Department of Electrical and Computer Engineering, Pittsburgh, Pennsylvania, Aug. 2004.

[4] DORSEY, J. G., AND SIEWIOREK, D. P. Online Power Monitoring for Wearable Systems. In *Proceedings of the Sixth International Symposium on Wearable Computers* (Seattle, Washington, Oct. 2002), pp. 137–138.

[5] EBERT, J.-P., BURNS, B., AND WOLISZ, A. A trace-based approach for determining the energy consumption of a WLAN network interface. In *Proceedings of European Wireless 2002* (Florence, Italy, Feb. 2002).

[6] FALL, K., AND VARADHAN, K. ns *Notes and Documentation*. The VINT Project, Oct. 1998.

[7] FEENEY, L. M., AND NILSSON, M. Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)* (Anchorage, Alaska, 2001), pp. 1548–1557.

[8] HUANG, L., AND LAI, T.-H. On the Scalability of IEEE 802.11 Ad Hoc Networks. In *Proceedings of the Third ACM International Symposium on Mobile*

Ad Hoc Networking and Computing (MobiHoc 2002) (Lausanne, Switzerland, June 2002), pp. 173–182.

[9] JOHNSON, D. B., MALTZ, D. A., AND HU, Y.-C. *The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR)*. IETF MANET Working Group, Apr. 2003. Internet-Draft, `draft-ietf-manet-dsr-09.txt`.

[10] JUNG, E.-S., AND VAIDYA, N. H. An Energy Efficient MAC Protocol for Wireless LANs. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (Infocom 2002)* (New York, New York, June 2002), pp. 1756–1764.

[11] KAMERMAN, A., AND MONTEBAN, L. WaveLAN-II: A high-performance wireless LAN for the unlicensed band. *Bell Labs Technical Journal 2* (1997).

[12] LAN MAN STANDARDS COMMITTEE OF THE IEEE COMPUTER SOCIETY. *Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications (IEEE Std. 802.11-1997)*, 1997.

[13] LAN MAN STANDARDS COMMITTEE OF THE IEEE COMPUTER SOCIETY. *Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Higher-Speed Physical Layer Extension in the 2.4GHz Band (IEEE Std. 802.11b-1999)*, 1999.

[14] ZHENG, R., AND KRAVETS, R. On-demand Power Management for Ad Hoc Networks. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (Infocom 2003)* (San Francisco, California, Apr. 2003).