Dynamic, Power-Aware Scheduling for Mobile Clients Using a Transparent Proxy

Michael Gundlach, Sarah Doster, Haijin Yan, David K. Lowenthal, Scott A. Watterson
Dept. of Computer Science, The University of Georgia
Athens, GA, 30606
Email:{gundlach,sdoster,yan,dkl,saw}@cs.uga.edu

Surendar Chandra
Dept. of Computer Science and Engineering, The University of Notre Dame
Notre Dame, IN, 46556
Email: surendar@cse.nd.edu

Abstract

Mobile computers consume significant amounts of energy when receiving large files. The wireless network interface card (WNIC) is the primary source of this energy consumption. One way to reduce the energy consumed is to transmit the packets to clients in a predictable fashion. Specifically, the packets can be sent in bursts *to clients, who can then switch to a lower power* sleep *state between bursts. This technique is especially effective when the bandwidth of a stream is small.*

This paper investigates techniques for saving energy in a multiple-client scenario, where clients may be receiving either UDP or TCP data. Energy is saved by using a transparent proxy that is invisible to both clients and servers. The proxy implementation maintains separate connections to the client and server so that a large increase in transmission time is avoided. The proxy also buffers data and dynamically generates a global transmission schedule that includes all active clients. Results show that energy savings within 10-15% of optimal are common, with little packet loss.

1 Introduction

The wireless network interface card (WNIC) often causes the largest power drain in a mobile client. This is especially true in (1) streaming multimedia applications and (2) `ftp` or `HTTP` file downloads. In both situations, packets arrive frequently (though not always predictably) requiring the WNIC to remain in *idle* or *receive* mode, both of which use significant amounts of energy. Fortunately, a proxy may buffer packets for a client and then send them to the client in *bursts* at regular intervals. This allows the client to keep its WNIC in *sleep* mode while not receiving data; this mode consumes an order of magnitude less power. This is especially effective when the bandwidth of the stream is low, in which case a large fraction of the overall energy can be saved.

There are many challenges in the implementation of such a scheme. Ideally, the proxy should be transparent, so that clients can save energy with only minor modifications. It should also avoid parsing packet data, so that it can support any protocol. Furthermore, the proxy should be able to support multiple clients, as well as multiple connection types (e.g., multimedia, HTTP, ftp, etc.) simultaneously among those clients. This is important because it is common for multiple clients to share a single wireless access point. Currently, however, only single-client, proxy-based solutions exist[4, 3, 14]. Multiple clients present additional challenges. An example (shown in [3]) is that data can arrive at the access points for different clients at the same time—because the wireless medium is shared between clients, an energy-aware scheme must involve clients agreeing on a global schedule. Finally, a scheduling policy must be developed that can adapt to both regular and bursty behavior.

Our implementation addresses these concerns. We have designed a scheduling policy that bursts packets to clients; it is implemented in a transparent proxy that resides between clients and servers, which both believe connections to each other are direct. Our proxy uses *address spoofing* for transparency and separate connections to the client and server to implement this abstraction efficiently. We allow any protocol to be used, and we support multiple clients as well as different types of connections from those clients. The number of clients supported is dependent only on the the effective bandwidth of the network. All clients save energy

by receiving data in bursts from a wireless access point.

In this paper we investigate dynamic scheduling of multiple clients, using both UDP and TCP traffic. Our results show that energy savings within 10-15% of optimal are common. For example, when multiple clients viewing 56kbps UDP streams are connected to the proxy, they save over 75% energy compared to a naive client (one who keeps its WNIC in high-power mode exclusively). This is within 15% of the theoretical optimal. In addition, with mixed UDP and TCP traffic, clients downloading either or both were able to achieve good energy savings. Furthermore, this is usually done with few missed packets (typically less than 2%) on the clients.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the design and implementation of our proxy as well as its scheduling policy, and Section 4 describes our experiments and discusses the results. Finally, Section 5 summarizes the paper.

2 Related Work

The idea of a transparent proxy has been previously studied. These can be used to improve TCP throughput [13]. Another way they can be used is to partition the Internet into distinct regions [8]. Also, *connection splicing* can be used with a proxy, which can improve latency within the proxy [16]. The primary difference with using a transparent proxy in our setting is that we must buffer packets in the proxy. This means that a transparent proxy will increase round-trip times from the point of view of both sender and receiver, potentially decreasing the TCP window size and hence increasing the transmission time. Our proxy handles this problem by using transparent double connections along with source address spoofing (see Section 3). Our implementation is similar in spirit to that of Indirect TCP [1] and Snoop [2]. The former splits a TCP connection at the base station and the latter modifies the link layer; both are intended to optimize for wireless networks. The differences between these two protocols and our transparent proxy are that (1) they are optimizing for performance, whereas we are scheduling for reduced WNIC energy, and (2) our proxy handles both UDP *and* TCP traffic, as opposed to only TCP traffic (which causes some implementation difficulties), and (3) we are not modifying the access point or any other part of TCP.

A body of closely related work to ours provides energy savings for a single multimedia client. This has been done by [14] as well as [4, 3]. The former includes transcoding, conversion of a variable bit rate stream to a constant bit rate stream, and client side prediction. The latter provides energy savings for a single multimedia client for Quicktime, Real, and MS Media. However, while it effectively saves energy for a single client, the scheduling policy developed in that work caused high rates of collision and packet loss for multiple clients. Our work is distinct in that it supports multiple concurrent clients.

Our work uses a particular scheduling algorithm for both UDP and TCP data. It schedules each client on the same frequency, so all clients share the total bandwidth; this is similar to TDMA [7]. Others have also worked on wireless scheduling algorithms (e.g., [5]).

Other related approaches are to use the energy-saving mechanisms defined by 802.11b. However, it is not a good match for multimedia [4]. One improvement to 802.11b is the Bounded Slowdown Protocol [9], which uses minimal energy given a desired maximum increase in round trip time. However, like 802.11b, this protocol is aimed at long periods of inactivity followed by small amounts of data received (e.g., web pages). Our work is focused on multimedia streams, which by their nature have packets arriving for a long period of time. Finally, there has been work done on reducing idle energy in the network interface [15].

3 Implementation

This section describes our implementation. Section 3.1 gives an overview, and Section 3.2 describes our scheduling policy and its implementation, including *address spoofing* to achieve transparency without unduly increasing the transmission time. Section 3.3 describes *delay compensation* algorithms used on clients to adjust to routing delays in the wireless access point.

3.1 Overview

Our implementation is within a proxy that is interposed between servers and clients. It buffers data from the servers, and transmits it at regular intervals as a burst to the appropriate client. This way, the access point will not have to make scheduling decisions between clients, because only one client will be receiving at a time. Clients can request arbitrary types of data, though in our experiments we use UDP and TCP.

We collect a trace of the wireless-side activity using a packet sniffer running on a mobile computer known as the *monitoring station*. This trace is read by a simulator postmortem in order to determine energy used per client. This is compared to the total energy used by a naive client, which keeps its WNIC in high-power mode for the duration of the trace. The overall architecture is shown in Figure 1.

We assume that a wireless network interface card (WNIC) can be in *sleep*, *idle*, *receive*, or *transmit* mode. *Sleep* mode uses a very small amount of power, and during this time no data can be received or transmitted. The remaining modes use a relatively large amount of power, with *receive* and *transmit* modes somewhat larger than that used by *idle* mode [17, 6]. We therefore refer to *sleep*

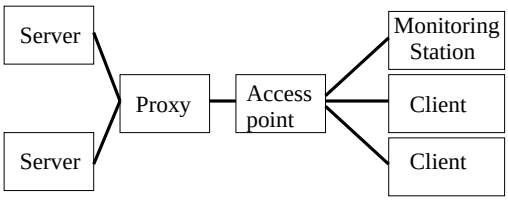

Figure 1. System architecture.

mode as *low-power mode* and all other modes as *high-power mode* for the remainder of the paper. A client saves energy by transitioning its WNIC between high- and low-power modes according to our scheduling policy, which is described next.

3.2 Scheduling Policy

This section describes the design and implementation of our energy-conserving scheduling policy. We use a proxy placed between the server and the wireless access point to carry out this policy. Section 3.2.1 discusses the design of our policy, and Section 3.2.2 details our implementation of the policy.

3.2.1 Design

Our primary design goal is to transform ordinary data streams into bursty streams, scheduling bursts so that each client receives a fair share of the bandwidth. We implement this policy through a transparent proxy, which is described below. The scheduling policy (see Figure 2) works as follows. The proxy broadcasts a schedule message as a UDP packet to all active clients at well-defined intervals. We define the *scheduler rendezvous point* (*SRP*) as a moment in time at which the proxy agrees to send the schedule. The schedule describes the length of each client's data burst and the order of the bursts, so that client i is assigned rendezvous point CRP_i. At point CRP_i, client i transitions its WNIC to high-power mode. At the same time, the proxy transmits data from the packet queue for client i to that client in a burst, marking the type-of-service bit in the IP header of the last packet so that the client knows when to transition its WNIC back to low-power mode. The time period between schedule broadcasts is known as a *burst interval*. Each schedule is valid for exactly one burst interval, and then a schedule for the next burst interval is sent at the next SRP[1]. In the example in Figure 2, in fact, there is another client that receives traffic during the second burst interval.

Schedules are determined immediately before being broadcast. We allow the size of a schedule to be variable or fixed. If schedules are of varying size, a new schedule is determined by examining a snapshot of the packet queues for all clients. The schedule is constructed such that each client can empty its packet queue during the next burst interval. The schedule will also contain the time at which the following schedule will be broadcast. If the schedule is a fixed size, however, the proxy gives each client a fraction of the available burst interval proportional to the amount of data in its packet queue. At present, we do not perform admission control at the proxy and so do not handle overload; to solve this problem we could leverage off of the significant amount of work in this area (e.g., [18]).

Our proxy is transparent, meaning that neither clients nor servers are aware of its involvement in any data transfer. A transparent proxy is desired for generality, as application protocols such as RTSP, HTTP and ftp can be supported without explicitly understanding the protocol semantics. To initiate a connection, the client contacts the server, which responds by opening a connection (as usual). After the connection is established, the server will simply send data to the client as normal. The client, on the other hand, sees data being transmitted from the server in a bursty manner. The client must also read the UDP broadcast packet from the proxy, which contains its rendezvous point as well as the arrival time of the next schedule. The client can turn off its WNIC until its rendezvous point is reached, at which point it transitions the WNIC to high-power mode. After the client receives its burst, it transitions the WNIC back to low-power mode until the next schedule packet is due. The modifications to the client to allow this are straightforward and could be implemented with a simple daemon. Note that the client may want to itself buffer multimedia packets and locally deliver them to the multimedia player at a regular pace. Such buffering introduces its own energy requirements, which we are not modeling.

3.2.2 Implementation

The proxy is implemented using the Linux 2.4.18 bridging mechanism [10]. We use the bridging code in the Linux 2.4.18 kernel, as well as the `brctl` utility, to do this. The proxy uses *IPQ*, a packet filter built into iptables, to catch all incoming or outgoing packets. The proxy is implemented using multiple threads: an *IPQ* thread, which catches and possibly modifies all incoming or outgoing packets; a bursting thread, which bursts data to clients; and a queuing thread, which moves packets between queues for the clients and servers.

Because the proxy is transparent, the clients and servers believe that they are connected to each other directly. However, as described above, to avoid a large increase in transmission time, there must be separate connections between (1) the client and the proxy and (2) the proxy and the server.

Figure 3 describes the steps taken when a client connects

[1] While the information sent to each client could be done individually at the end of its burst, complete information is available after *all* clients have received their data for a burst interval. This allows fairness between clients as well as schedules with variable burst intervals (see Section 4).

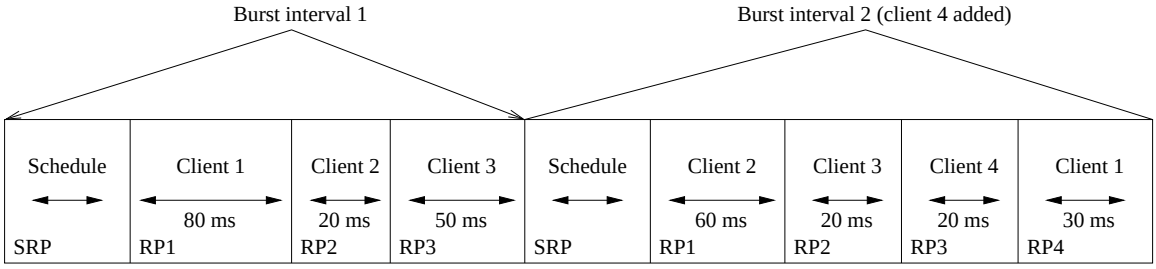

Figure 2. Example scheduling policy with four clients. Client 4 has traffic during the first burst interval and so joins the schedule for the second burst interval.

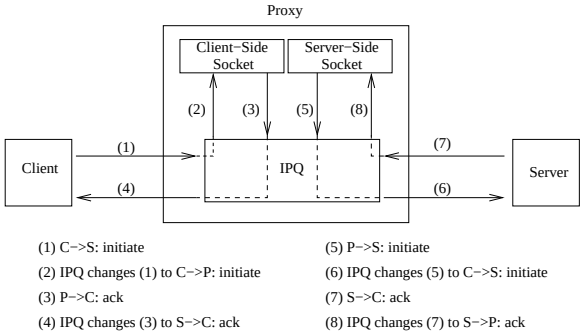

Figure 3. Picture of the steps taken on a TCP connection from client (C) to server (S) with a transparent proxy (P).

to the server. In step 1, the client sends an initiate request (SYN packet) to the server. This request is caught by *IPQ*, which creates a receiving socket (the "client-side" socket) and changes the header of the message so that the request is from the client to the proxy, then "re-injects" the packet to be routed (step 2). The proxy responds to this by first acknowledging the client (returning a SYN ACK packet) (step 3); this acknowledgement is caught by *IPQ*, which changes the header to indicate it is from the server (step 4). The proxy also creates a server-side socket and initiates a connection to the server (step 5); *IPQ* catches the message and changes the header to indicate that the message is from the client (step 6). Finally, the server sends back an acknowledgement to the the client (step 7), which *IPQ* changes so that the destination is the proxy (step 8). Similar steps are taken to handle the acknowledgement of the SYN ACK packet, as well as transmission of packets from the server to the client.

While the basic idea is given above, there are important low-level details required for its implementation. These include handling unexpected packet orderings, correctly marking the last packet of a client's burst, and handling bandwidth limitations. We describe each of these in turn.

Packet Ordering The proxy transmits both UDP and TCP data and sends out a new schedule after all clients have received their data for the previous schedule. Because UDP is unreliable, packets can arrive at the client in a different order than they were sent from the proxy. Specifically, the schedule message (which is UDP) for burst interval B can actually arrive at a client when the client is (1) still receiving data for burst interval $B-1$ or (2) already receiving data packets for burst interval B. In other words, in practice the schedule can arrive slightly before or slightly after the next burst. To handle (1), clients must ignore the new schedule until they receive a marked packet that indicates the end of the previous burst (or until another schedule is received, if the marked packet is dropped). To handle (2), clients must accept data that comes before a schedule.

Packet Marking Recall that a burst is terminated with a marked packet. While this idea is simple in principle, in practice it is somewhat more difficult for TCP packets. This is because we mark these packets in the *IPQ* thread, but we determine what data should be marked in the bursting thread. We accomplish this by having the bursting thread communicate via shared variables as to which is the last byte in a burst. Specifically, there are three shared variables per client-side socket C: B_c, I_c, and M_c. Variable B_c stores the number of bytes sent by the bursting thread, and I_c stores the number sent by the *IPQ* thread; the invariant is $B_c >= I_c$. Variable M_c, which represents the byte number to be marked, is initialized to -1. When the bursting thread sends the last packet in a burst, it copies B_c into M_c. The *IPQ* thread then marks a packet when $I_c = M_c$ and resets M_c to -1. This scheme is further complicated by potential retransmissions from the proxy to the client, which are handled by the proxy by comparing sequence numbers. For this case, B_c would not be incremented.

Bandwidth Constraints The schedule developed for a burst interval contains a set of client IP addresses and their associated transmission start and end times. However, the proxy must be careful to accurately estimate the amount of

data that can be sent to a client during its reception period. The proxy can transmit data to the access point faster than the access point can transmit to the client. If the proxy sends data to a client (through the bursting thread) during the whole period, some of that data may actually be forwarded by the *IPQ* thread *after* the end of the interval, due to the limited wireless network bandwidth. If this happens, the data for that client may take longer to transmit than the time allotted, meaning that subsequent clients will not receive their data as scheduled. This leads to wasted energy on those clients. Furthermore, if the proxy sends too few packets, bandwidth is wasted.

To address this problem, we executed a set of microbenchmarks to create a model of send overhead and latency on our wireless network. From these, we developed a linear cost function based on the message size. The proxy uses this to estimate how much data can be sent in a given time period.

Proxy memory requirements In our experiments, the memory buffer required on the proxy is not large. The proxy requires the maximum space when the entire wireless bandwidth is in use. With an effective bandwidth of 4Mbps, this means that even if one second of data (to all clients) had to be buffered, 512KB would be sufficient.

3.3 Delay Compensation Algorithms

A client's WNIC must be transitioned to high-power mode to receive the schedule and then again to receive its burst. If the client transitions early, some energy will be wasted while waiting for the packets to arrive; if the client transitions late, packets will be missed. The goal is to minimize the wasted energy, while avoiding missing packets. If the client could perfectly predict when a packet would arrive based on the information in the schedule, this would not be a problem. However, while the proxy usually sends the schedule as well as data bursts with better than millisecond accuracy, delay may at times be observed by the client. Even though the proxy is as close to the client as possible, all packets must pass through the access point. This, as well as the multithreaded nature of the proxy, can cause a packet to arrive earlier or later than expected. Another possibility is that the clocks on the proxy and a client may not be perfectly synchronized. For this reason, we use an adaptive *delay compensation algorithm* on each client to determine when to transition the WNIC out of low-power mode.

The intuition behind it is as follows. If a schedule packet arrives earlier or later than expected, it is likely a change in access point delay between the proxy and the client, and several subsequent schedule packets will arrive according to the same pattern. Adaptive delay compensation algorithms therefore set each transition point a fixed amount after the *arrival time of the previous schedule*. In order to reduce missed packets, the amount is slightly less than a burst interval; we refer to the difference as the *early transition amount*. While this synchronization strategy is simple, we found it to be effective in practice.

4 Performance

This section describes our experiments and presents our results. Section 4.1 describes the experimental setup. Section 4.2 describes the experiments performed and gives the results. Finally, Section 4.3 analyzes the results.

4.1 Experimental Setup

In each experiment, clients either (1) requested a video in Real format or (2) downloaded either `HTTP` or `ftp`. The results obtained from the Real format video should apply to any streaming data (e.g. Quicktime, MS Media); while there are some differences in the streaming patterns, all streaming data should be handled similarly. The monitoring station and clients were various laptops using 11Mbps Orinoco PCMCIA WNICs. The monitoring station ran `tcpdump` to capture data about each packet on the wireless network. We ran RealServer 8.01 as our video server, and each client laptop ran RealOne 2.0. The multimedia server, web server, and access point were connected over 100Mbps Fast Ethernet. The multimedia clients requested a 1:59s trailer for the movie *The Wall*, encoded by Adobe Premier 6.0 at 56kbps, 128kbps, 256kbps, or 512kbps. Because the encoder could not perfectly match the requested bitrates, the effective bitrates of these streams are 34kbps, 80kbps, 225kbps, and 450kbps respectively. For video experiments, requests were spaced roughly one second apart in order to spread traffic; transmitting identical multimedia streams simultaneously could cause large spikes of activity during high bitrate periods. The streams were unicast to the clients, so that the total bandwidth used was roughly equal to the sum of the individual streams (i.e., no streams were multicast).

We used a simulator to read the tcpdump trace postmortem, calculating (1) how much time a client's WNIC has spent in high- and low-power mode, (2) how many bytes its WNIC has transmitted and received, (3) how many packets are lost (UDP) or packets dropped (TCP) for each client, and (4) how much energy the client would use by transitioning its WNIC between modes according to a given delay compensation algorithm. This is compared to the naive client, which keeps its WNIC in high-power mode. The card simulated is 2.4Ghz WaveLAN DSSS, which uses 1319 mJ/s when idle, 1425 mJ/s when receiving, 1675 mJ/s when transmitting, and 177mJ/s when sleeping [17, 6]. Also, we model the energy cost of transitioning the WNIC from *sleep* to *idle* mode as 2 ms in *idle* time [9].

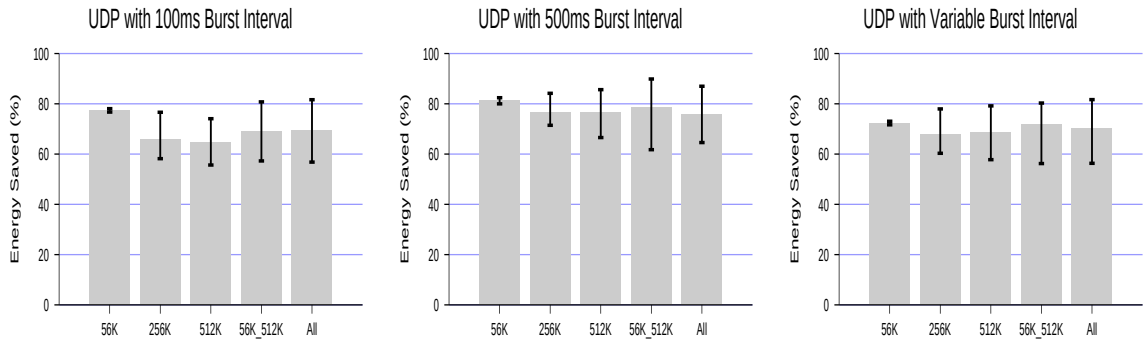

Figure 4. Results with ten clients viewing UDP (video) streams with 100ms, 500ms, and variable burst intervals. Average, minimum, and maximum energy usage for the clients are represented by the bar and error lines, respectively. The first three bars (56K, 256K, 512K) show the experiments where each client viewed an identical stream, while the last two bars represent cases where the clients viewed streams of different fidelity.

4.2 Experiments

We performed three different sets of experiments. The first set investigated a scenario where all clients view video streams. This shows how well our system generalizes from prior work done with a single video client [3, 14]. Next, we performed experiments with all clients downloading `HTTP` data. This is a more realistic scenario, as web access is the most common activity on wireless devices. Third, we performed experiments where a subset of the clients view video streams and the rest download TCP data (either `HTTP` or `ftp`). These experiments are intended to investigate potential interactions between the two types of data.

For each type of experiment, we tested three burst interval sizes: fixed burst intervals of 100ms and 500ms as well as a variable burst interval. This is intended to determine the tradeoffs involved in scheduling. We discuss the results of our experiments below. In Section 4.3 we analyze the results.

Multiple video clients The first set of experiments involved each client receiving only a video stream. Figure 4 shows the average, minimum, and maximum energy usage for each experiment (represented by the bar and error lines, respectively). There were five types of client access patterns. The first three involved each client viewing an identical bitrate stream; we tried 56Kbps, 256Kbps, and 512Kbps. The results show that the average energy saved is 77% for the 56Kbps streams, 66% for the 256Kbps streams, and 53% for the 512Kbps streams.

The fourth type involved five clients viewing a 56Kbps stream and the other five viewing a 512Kbps stream. The final type involved five clients viewing a 56Kbps stream and the remaining five clients viewing different fidelities ranging at 56Kbps, 128Kbps, 256Kbps, and 512Kbps. The dynamic nature of our scheduler allows the energy savings to average about 69% in both experiments. As expected, lower fidelity streams save more energy because they use less bandwidth, giving them more opportunity to transition the WNIC to *sleep* mode.

Multiple TCP clients The next set of experiments involves each client browsing the web, which generates multiple concurrent TCP streams per client. For each experiment, we used a script (which was generated prior to the experiments) to ensure that the traffic pattern remained identical across different experiments. We used burst intervals of 100ms and 500ms as well as a variable interval. Due to space limitations a graph is not shown, but the results showed that as with the video experiments, the clients save significant amounts of energy compared to a naive client (between 70 and 80%).

Video and TCP traffic We next examined our system when handling a combination of video and TCP traffic. Four experiments were run; each one has seven clients viewing video and three clients browsing the web. Figure 5 shows the results of our tests. The first three bars show all video clients viewing a specific quality stream, 56Kbps, 256Kbps, and 512Kbps respectively. The fourth test (labeled *All/TCP*) had the video clients view a variety of fidelities. The energy savings varies from just over 50% to just under 90%. One interesting point is that the best-case energy savings among the video clients is similar among different fidelities. This is discussed further in Section 4.3.

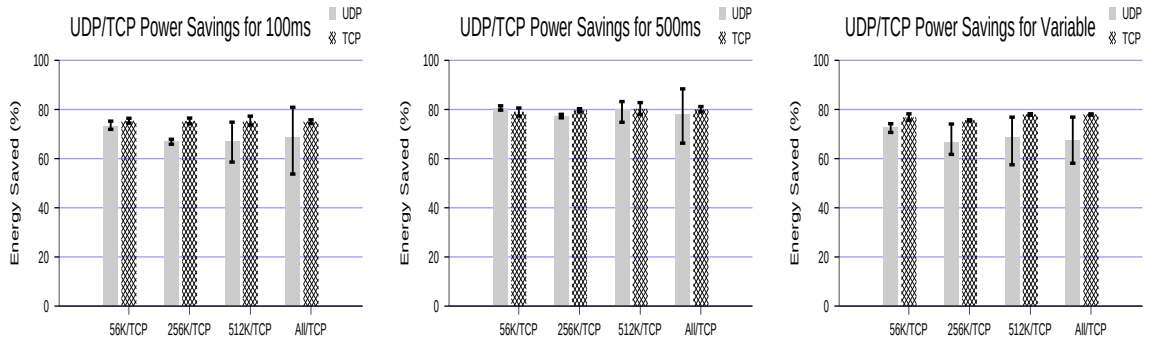

Figure 5. Results with ten clients with some viewing UDP (video) streams and others downloading TCP (HTTP) data with 100ms, 500ms, and variable burst intervals. The left bar is for the UDP clients and the right one for TCP clients.

4.3 Analysis of Results

This section analyzes the results. We first compare our results to the theoretical optimal. Next, we discuss packet loss. We then compare our results to those that could be obtained using a static schedule. Finally, we discuss the worst-case client.

Comparison to optimal One important metric to consider is how close the energy saving for a client is to the theoretical optimal. The optimal energy saved for a stream can be computed by the formula:

$$E_{opt} = \frac{(T_r \times P_r) + ((T_p - T_r) \times P_s)}{(T_n \times P_i) + P_b}.$$

Here, T_r is the time to receive the entire stream if it were sent all at once, P_r is the energy cost per second to receive, T_p is the time for the download using the proxy, P_s is the energy cost per second to sleep, T_n is the time for the download without the proxy, P_i is the energy cost per second in idle mode, and P_b is the energy cost per byte. This is based on the idea that optimally, the WNIC is in high-power mode only to receive the data and in *sleep* mode at all other times, while a naive client is idle when not receiving data. Using this formula, the optimal energy saved for the 56Kbps, 256Kbps, and 512Kbps streams from the video-only experiments is 90%, 83%, and 77% respectively, compared to our results (from Section 4.2) of 77%, 66%, and 53% for the same streams.

In the mixed video/web experiments, generally, the median client energy savings is within 15% of optimal. However, there are some outlying cases; for example, with 512Kbps video files, the best case client saves over 80% of the WNIC energy compared to a naive client. As stated above, the optimal energy savings is 77% for a 512Kbps stream. The reason for this anomaly is that the peak bandwidth required to transfer 10 512Kbps streams exceeds the effective wireless network bandwidth. This causes RealServer to believe that the connection is lossy, and the stream is adapted to a lower-quality, lower-bandwidth one. This is a problem inherent to exceeding bandwidth limitations rather than a problem introduced by the proxy. The TCP clients have a lower variance in energy savings, which is because adaptation does not occur.

The reason the 100ms burst interval performs worse than the 500ms burst interval in all experiments is because the WNIC is transitioned five times more often. As there is an early transition amount (see Section 3.3) each time the WNIC is transitioned to high-power mode (to avoid missing packets), this penalty is significant. For example, in our experiments, we used 6ms as the early transition amount for the 100ms experiments. In general, our experiments showed a factor of four increase in this penalty, on average from 3 seconds to 11 seconds of time the WNIC must be in high-power mode, between using a 500ms and 100ms burst interval. To investigate this further, we tried different early transition amounts for a 100ms burst interval, transitioning 0, 2, 4, 6, 8, and 10ms early. The ideal value is the one that minimizes wasted energy due to transitioning the WNIC early while avoiding missed packets. The latter degrades performance in two ways: first, it makes video stream quality poor and causes TCP data retransmission, and second, if a schedule is missed, a client must keep the WNIC in high-power mode until the next schedule arrives. Figure 6 shows the amount of wasted energy on a single client caused by early transitioning of the WNIC. As the early transition amount decreases, the overhead for transitioning to high-power mode early decreases, but the number of missed schedules increases. In this case, 6ms is the best value to choose. The other dimension is missed packets;

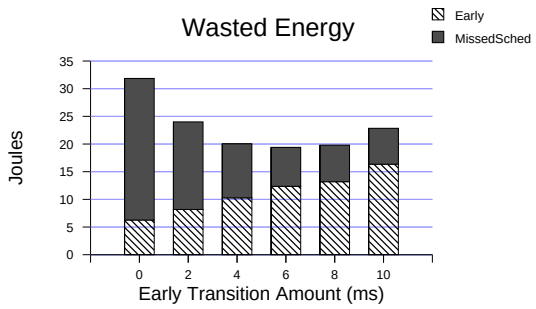

Figure 6. The effect of different early transition amounts on wasted energy.

this ranged from 0.97% (10ms early) to 1.83% (0ms early). Variable burst intervals appear to have energy savings in between the 100ms and 500ms interval. This is because the minimum burst interval is 100ms, and the maximum is less than 500ms unless several streams have high bandwidth.

Packets lost or dropped Packets lost (UDP) or packets dropped (TCP) are usually less than 2% with a few outliers, indicating that generally, the data is sent according to the schedule issued by the proxy.

It is difficult to analyze the effect on UDP streams of dropping up to 5% of the packets. Obviously, in the case of a video, the video will appear worse to the user. However, while we believe that missing a few packets is acceptable, this is fundamentally an human perception issue. We leave this to researchers in that area.

On the other hand, dropping TCP packets in reality causes retransmissions. In our tests, we actually receive packets that would be dropped and compute what would be dropped postmortem. (This is primarily because most of our clients are running Windows, where we do not know how to drop packets when necessary.) To estimate the effect of drops, we ran separate experiments with one client and Netfilter [11], which we configured so that if the client was in *sleep* mode, packets are actually dropped. We found that the effect of dropping packets was small (no more than a 10% increase in transmission time, which would have the corresponding effect of an expected increase in in energy consumed of no more than 5%[2]). We repeated an experiment using DummyNet [12], configuring a 4Mb/s network with a 2ms round-trip time and 5% drop rate. Similar results were observed. Essentially, the low round-trip time between proxy and client means that dropping packets is not severe.

[2]The extra energy is consumed because the transmission time is longer; however, most of the extra time would be time the WNIC is in *sleep* mode.

Comparison to static schedules For the subset of experiments above where all clients view streams of equal fidelity, a static schedule (rather than our dynamic one) can be used. In other words, the proxy can simply broadcast a single (permanent) burst interval for each client. This should save energy compared to our dynamic approach in that there is no early transition necessary to receive the schedule. If all clients are receiving approximately the same amount of data, it is also sufficient (i.e. no bandwidth is wasted by assigning a client a fixed-size amount of time for receiving) because on average, each client is receiving data at approximately the same rate. Hence, we implemented a static schedule for comparison, using a 100ms burst interval and ten clients viewing (identical) video streams of 56Kbps, 256Kbps, and 512Kbps. We found that both average energy usage and variance is lowered by using a static schedule.

However, a static schedule is insufficient in the case that the fidelity of the videos vary per client or TCP traffic is involved. An example is shown in more detail in Figure 7, which contains the results from using fixed-sized slots for both TCP and UDP data. During TCP slots, all TCP data is sent to the appropriate clients, and during UDP slots, UDP data is sent. However, the burst interval was 500ms, and the TCP and UDP slots were varied so that for each traffic pattern (light, medium, and heavy), we ran experiments where the TCP slot size was larger than, smaller than, and roughly equal to the size necessary to support the TCP traffic. (We show only the medium traffic pattern.) In these experiments, during the TCP slot, all clients must have their WNIC in high-power mode, so that the latency increase is restrained. It is not possible to simply minimize the TCP slot size, as TCP data will not be able to travel through the proxy: as the slot size decreases, the end-to-end latency of background traffic increases. The larger the slot size, the more energy that is wasted. For example, while the energy use for the UDP clients is lowest with a small TCP slot, the added latency of the TCP stream is significant.

Instead, our dynamic schedule handles this seamlessly; recall from Figure 4 the following experiments: (1) five clients viewing a 56Kbps stream and the other five viewing a 512Kbps stream, and (2) different clients viewing different fidelities ranging at 56Kbps, 128k, 256Kbps, and 512Kbps. The dynamic nature of our scheduler allows the energy savings to average about 69% in both experiments.

Worst-case client In the case of the worst-performing client (as shown by the error bars in Figure 4), examination of the traces shows that missed schedules are the primary cause. In particular, this client generally happens to miss the broadcast *and* at the same time misses its data. This means that it does not receive a marked packet (and the WNIC remains in high-power mode); on the other hand, other clients that miss the broadcast may receive some or all of their burst and then transition the WNIC to *sleep* mode.

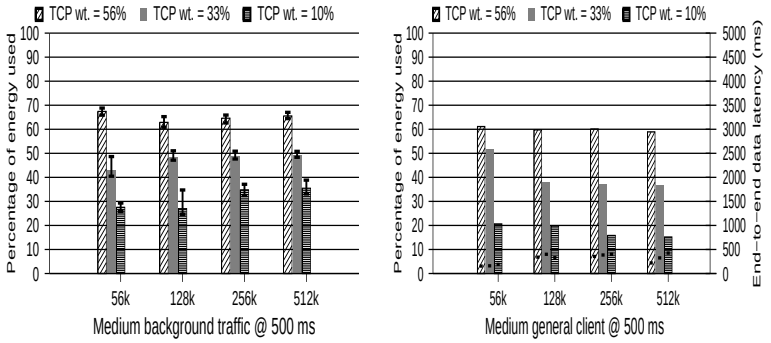

Figure 7. On the left, the average, minimum, and maximum energy usage for ten multimedia clients with TCP traffic on the network, using a static schedule. On the right, the TCP client is analyzed; both energy usage (bars, left y-axis) and transmission latency (dots, right y-axis).

5 Summary and Future Work

This paper has described a novel scheme to save energy for multiple mobile clients. We have designed a scheduling policy, capable of adapting to steady and bursty traffic. Our policy is implemented in a transparent proxy that bursts packets to clients. This allows clients to transition to low-power mode between bursts, saving energy. Our proxy uses *address spoofing* for transparency, which allows clients and servers to appear to communicate directly. It also maintains separate transparent TCP connections to the client and server in order to reduce transmission times

Results showed that energy savings within 10-15% of optimal were common with a low missed packet rate—typically less than 2%. For low-bandwidth streams, clients saved over 75% energy compared to a naive client. Our proxy was able to support multiple concurrent clients downloading both UDP and TCP data.

One avenue for future research is reducing the energy wasted by requiring the client to transition to high-power mode at both the schedule and burst rendezvous points. If the schedule does not change from one burst interval to the next, the proxy may inform the client that it will use the same schedule again for the next burst interval. The client can then transition to high-power mode only at its burst rendezvous point, avoiding the energy cost of receiving the schedule from the proxy.

References

[1] A. Bakre and B. R. Badrinath. I-TCP: Indirect TCP for mobile hosts. *15th International Conference on Distributed Computing Systems*, 1995.

[2] H. Balakrishnan, S. Seshan, and R. H. Katz. Improving reliable transport and handoff performance in cellular wireless networks. *ACM Wireless Networks*, 1(4), 1995.

[3] S. Chandra. Wireless network interface energy consumption implications of popular streaming formats. In *Multimedia Computing and Networking (MMCN '02)*, Jan 2002.

[4] S. Chandra and A. Vahdat. Application-specifi c network management for energy-aware streaming of popular multimedia formats. In *USENIX Annual Technical Conference*, 2002.

[5] S. Damodaran and K. M. Sivalingam. Scheduling algorithms for multiple channel wireless local area networks. *Computer Communications*, 2002.

[6] P. J. M. Havinga. *Mobile Multimedia Systems*. PhD thesis, Univ. of Twente, Feb 2000.

[7] IEC. `www.iec.org/online/tutorials/tdma/`.

[8] B. Knutsson and L. Peterson. Transparent proxy signalling. *Journal of Communications and Networks*, 2001.

[9] R. Krashinsky and H. Balakrishnan. Minimizing energy for wireless web access with bounded slowdown. In *Mobicom 2002*, Atlanta, GA, September 2002.

[10] Linux Bridging. `http://bridge.sourceforge.net/`.

[11] Netfi lter. `http://www.netfilter.org`.

[12] L. Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM Computer Communications Review*, 27(1), Jan. 1997.

[13] P. Rodriguez, S. Sibal, and O. Spatscheck. TPOT: translucent proxying of TCP. *Computer Communications*, 24(2):249–255, 2001.

[14] P. Shenoy and P. Radkov. Proxy-assisted power-friendly streaming to mobile devices. In *Proceedings of the 2003 Multimedia Computing and Networking Conference*, Santa Clara, CA, January 2003.

[15] E. Shih, P. Bahl, and M. Sinclair. Wake on wireless: An event driven energy saving strategy for battery operated devices. In *Mobicom 2002*, Atlanta, GA, September 2002.

[16] O. Spatscheck, J. S. Hansen, J. H. Hartman, and L. L. Peterson. Optimizing TCP forwarder performance. *IEEE/ACM Transactions on Networking*, 8(2):146–157, 2000.

[17] M. Stemm, P. Gauthier, D. Harada, and R. H. Katz. Reducing power consumption of network interfaces in hand-held devices. In *Proc. 3rd International Workshop on Mobile Multimedia Communications*, Sept. 1996.

[18] H. M. Vin, A. Goyal, and P. Goyal. Algorithms for designing multimedia servers. *Computer Communications*, 18(3):192–203, 1995.