Client-Centered Energy and Delay Analysis for TCP Downloads

Haijin Yan, Rupa Krishnan, Scott A. Watterson, David K. Lowenthal, Kang Li
University of Georgia
{yan,krishnan,saw,dkl,kangli}@cs.uga.edu

Larry L. Peterson
Princeton University
llp@cs.princeton.edu

Abstract

In mobile devices, the wireless network interface card (WNIC) consumes a significant portion of overall system energy. One way to reduce energy consumed by a mobile device is to transition its WNIC to a lower-power *sleep* mode when data is not being received or transmitted.

This paper investigates *client-centered* techniques for trading download time for energy savings during TCP downloads, in an attempt to reduce the energy*delay product. Effectively saving WNIC energy during a TCP download is difficult because TCP streams tend to be smooth, leaving little potential sleep time. The basic idea behind our technique is that the client increases the amount of time that can be spent in *sleep* mode by shaping the traffic. In particular, the client convinces the server to send data in predictable bursts, trading lower WNIC energy cost for increased transmission time. Our technique does not rely on any assistance from the server, a proxy, or IEEE 802.11b power-saving mode. Results show that in Internet experiments our scheme outperforms baseline TCP by 64% in the best case, with an average improvement of 19%.

1 Introduction

With the explosion of battery-constrained mobile devices, conserving energy has become increasingly important. One significant source of consumed energy on such devices is the wireless network interface card (WNIC); in fact, it can in some cases be the single largest power drain in a mobile client. For example, when all components of an IBM 560X laptop are active, the WNIC accounts for 15% of overall system energy [1]. The energy consumed by a WNIC is particularly significant in situations where data is received for a long period of time, e.g., file downloads and streaming multimedia applications. These applications require the WNIC to remain in *idle*, *receive*, or *transmit* mode, all of which use significant amounts of energy. Remaining in high power mode is the most effective way to minimize transfer time, but in a mobile environment, users may be willing to increase transfer time in exchange for energy savings. In fact, IEEE 802.11b power saving mode (PSM) [2] is designed specifically for this purpose. Similarly, techniques such as dynamic voltage scaling [3] and disk spindown [4] trade reduced energy for potentially increased execution time.

The focus of this paper is to allow mobile clients to trade download speed for energy savings during TCP downloads in a *client-centered* manner, i.e., *without* any assistance from servers, proxies or PSM. For a mobile device, low access delay and long battery life are both good indications of high service quality. However, these two quality metrics are often in conflict, and typically saving energy results in additional access delay. Our goal is to balance the energy and delay costs, as measured by energy*delay.

We focus our efforts in this work on handling TCP streams initiated by the client to a server via request/response, using round-trip time estimates to determine when to transition the WNIC between *idle* and *sleep* mode. For large files (e.g. Kazaa downloads), the client saves the most energy when data is transmitted in bursts, because there is more time the client can transition its WNIC to *sleep* state between bursts. However, TCP does nothing to combine packets into bursts—instead, it attempts to smooth the packet stream—which makes saving energy during long transfers difficult. When bursts do appear naturally in a TCP stream due to external factors, they are unpredictable and so not amenable to energy savings.

We have designed and implemented a technique for TCP downloads in mobile clients that trades energy savings for increased delay. In the best case, we outperform baseline TCP by 64% when comparing energy*delay, with an average improvement of 19% over seven Internet sites. We also ran experiments in a emulated environment (using *DummyNet* [5]), where we observe as much as a 73% improvement.

Our implementation is within *Netfilter* [6], a kernel-level packet filter, and uses three basic techniques. First, the client shapes the traffic of files sent by the server to create gaps between packet arrivals, increasing the time that the WNIC can be placed in *sleep* mode. As the bursts generated are relatively small in practice, the shaping, which is similar to what PSM itself causes, should not adversely affect Internet routers. This is verified by simulations we performed using ns2 [7]. Second, a client predicts the end of each burst using estimates of packet interarrival times and deviations. Finally, a client estimates arrival times for packets using predictions of round-trip times obtained by leveraging TCP timestamp information. The traffic shaping is performed strictly by the client, using TCP information to manipulate how the data is sent by the server.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the design and implementation of our client-centered technique, and Section 4 describes our experiments and discusses the results. Next, Section 5 provides discussion and describes future directions for this research. Finally, 6 summarizes the paper.

2 Related Work

One body of work that is closely related to ours is in providing energy savings to multimedia clients at the application level through the use of a proxy that is interposed in between servers and mobile clients. The proxy shapes the traffic into bursts and coordinates transmissions directly with the clients, allowing transition of the WNIC to *sleep* mode in between bursts [8, 9]. The disadvantage of these approaches is they require a proxy, which will not exist on many wireless networks; our method is completely implemented on mobile clients, requiring no assistance from a proxy or a server. Also, the proxy-based approaches above are focused on handling multimedia, where network traffic is primarily UDP. Our work is instead focused on TCP traffic.

Other related approaches are to use the energy-saving mechanisms defined by 802.11b (power-saving mode, or PSM) [2]. PSM, however, is not a good match for applications that receive data at a steady and frequent rate[8]. Furthermore, using PSM generally increases round-trip times to at least the nearest multiple of 100 *ms* [10][1].

One improvement to PSM is the Bounded Slowdown Protocol (BSD) [10], which uses minimal energy given a desired maximum increase in round trip time. BSD is effective in saving energy for web sessions compared to PSM and also avoids the significant delay that PSM incurs. However, fundamental to this approach is that after every request sent by the client, the WNIC must remain in high-power mode for a certain amount of time T. For example, if the desired maximum transmission time increase is 50%, T is 200 *ms*. Because BSD is implemented in the MAC layer, every acknowledgement sent by the client must be considered a request. Hence, in most situations (like the one above, assuming the round-trip time is less than 200 *ms*), BSD will have to leave the WNIC in high-power mode for each entire download. For energy saving, the goal of BSD is reducing energy consumption during periods of idle time, not saving energy *during* a transmission. Hence, during transmission its energy*delay is identical to regular TCP. Our approach, on the other hand saves energy during transmissions.

Controlling the TCP sender's behavior via manipulation of acknowledgements by the receiver is not a new idea. In [11], Chan et al. proposed the *Ack Regulator* to improve TCP performance on a 3G wireless link by regulating the flow of acks back to the sender. Similar work has been done in [12], where acknowledgement congestion control and acknowledgement filtering were proposed to smooth the flow of traffic from the sender. Our approach is to force predictable bursts by at times sending acknowledgements that have a zero-

[1] Round-trip times can be increased more than that if they approach (but are less than) a multiple of 100 *ms*, and the wireless link is not the bottleneck.

sized receiver window. This general technique is also used by M-TCP[13] for a different purpose—to put the sender into persistent mode during disconnection periods when mobile clients are roaming.

There has also been work done on reducing idle energy in the network interface [14]. This has the potential to improve energy usage, but cannot be used on current hardware. There has also been work in creating burstiness to save energy consumed by disks [15]. Our work is similar in spirit, but we focus on creating burstiness in network transmissions.

There has been significant research in power-aware computing at the network, hardware, and operating system levels, with respect to the energy*delay tradeoff. At the hardware level, dynamic voltage scaling (DVS) is a technique that allows processor speed to be decreased in order to run at at a lower energy level (e.g., [3]). OS scheduling can then take advantage of DVS[16]. At the operating system level, many have studied disk spindown to save energy [4]. Additionally, in some architectures individual memory banks can be powered down [17]. Recent work has advocated managing energy explicitly as a resource in the operating system [18]. Another approach is to have the OS exploit application information to save energy [1, 19]. At the network level, the basic ideas are to perform routing in a power-aware manner or to integrate power awareness into the transport layer [20]. Additionally, there has been work in investigating power-aware mechanisms for end-to-end communication in wireless networks [21]. Our work, on the other hand, saves energy through transitioning the WNIC to *sleep* mode. It exploits energy savings at the network level and does so in a client-centric manner.

3 Implementation

TCP uses congestion control and flow control to limit its data sending rate without overloading the network and the receiver. This rate limitation is achieved by a sliding window scheme that controls the number of in-flight packets (sent but not yet acknowledged) over each round-trip time. The window size in TCP is dynamically adjusted according to network conditions. For example, the window size shrinks when congestion occurs, while it is usually increased if all packets in a window are acknowledged. In addition to the window-based congestion control, TCP uses a *self-clocking* mechanism to pace outgoing packets. Instead of sending all the packets in a window at once, TCP only sends a new packet when another packet is acknowledged. This results in a smoothed data stream when the acknowledgments are evenly paced. Our implementation exploits TCP for energy savings. The basic idea is that the receiver forces the sender to send each window of

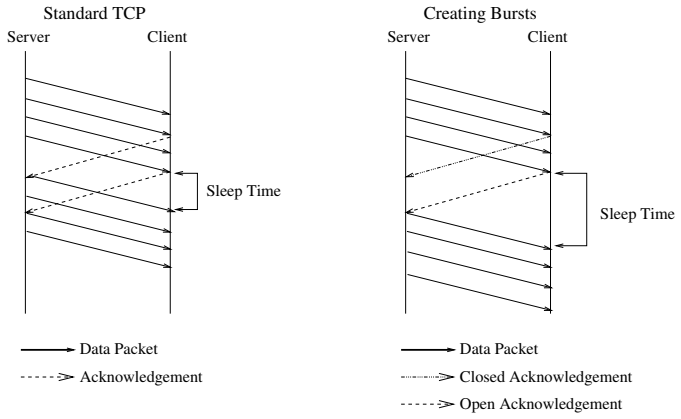

Figure 1: Example of creating bursts with a window size of four. On the left is standard TCP, and on the right is our technique. To create bursts, the first acknowledgement advertises a receiver window size of 0, and the second advertises a full buffer. This creates more potential time the WNIC can remain in *sleep* mode.

packets in a burst so that potential sleep time between bursts is maximized.

This section describes our implementation. Section 3.1 discusses the basic idea behind our client-centered technique. Next, Section 3.2 covers the challenges we encountered in designing and implementing our algorithm, and Section 3.3 discusses low-level implementation details in *Netfilter*.

3.1 Client-Centered Traffic Shaping

Our goal is to shape a smoothed TCP data stream into bursts to increase potential sleep intervals between packet receptions. To do this, we exploit TCP's flow control mechanism at the receiver (client) to manipulate the sender (server). In TCP, each acknowledgement from client to server contains the client's advertised receiver window size, which is the number of new packets it is able to hold in its buffer. In our modified TCP implementation, the client first announces zero buffer space to the sender to delay outgoing data packets at the server, and later announces appropriate buffer space to allow the server to release packets in a burst. Specifically, in each but the *last* acknowledgement in a window, the client advertises its receiver window size as zero (denoted as a *closed ack*). When the server receives a *closed ack*, it cannot send any further packets, as it believes that the receiver has no available buffer space to store them. When the receiver believes the window has completed, it triggers the next window of packets by sending an acknowledgement with a full window size (default 64KB) advertised. We denote this kind of acknowledgement as an *open ack*. Because the *open ack* (implicitly) acknowledges the (entire) previous window, the server will immediately send the (entire) next window. Figure 1 shows the difference between standard TCP and our client-centered technique.

Our technique converts a smooth TCP stream to a bursty TCP stream. This creates large gaps during which it is possible to transition the WNIC to *sleep* mode. In order to decide when to transition into and out of *sleep* mode, the client must infer two things: when a burst of packets ends, and when the first packet of the next burst arrives. We next discuss these in turn.

3.1.1 End of Burst Detection

The detection of the end of a burst is nontrivial because of the variance in round-trip times inherent to the Internet. The basic idea is for the client to infer the end of a burst when no packets arrive for a threshold amount of time. Clearly, there is a tradeoff between inferring the end of a burst aggressively (i.e., using a smaller threshold) and conservatively (using a larger threshold). The former increases the risk of missing packets, because the packets could merely be delayed (due to round-trip time variation) and arrive after the client transitions the WNIC to *sleep* mode. Such a situation directly leads to missed packets and can potentially cause an *increase* in energy consumption (as well as unacceptably long transmission times). The latter always increases both energy and transmission time overhead, as there is extra time spent attempting to correctly infer the end of the window, and it is spent in *idle* mode. Two broad approaches to end-of-burst detection are possible. One approach is to implement a fixed threshold, while the other is to use a dynamic threshold. Fixed thresholds are unable to adapt to network conditions and are therefore susceptible to packet loss. We study end of window detection in more detail in [22].

To avoid the pitfalls of fixed thresholds, we implemented a dynamic approach. Because each window is sent in a burst, we can use the interarrival times between packets and their variance to infer the end of a burst. We use each interrarival time as a sample and then use a weighted average scheme that is similar to the TCP timeout value computation. That is, given new samples for arrival (S_A) and deviation (S_D), we keep a running average of interarrival times (I_A) as well as interarrival deviation (I_D) using:

$$I_A = \frac{7I_A}{8} + \frac{S_A}{8} \qquad\qquad I_D = \frac{3I_D}{4} + \frac{S_D}{4}$$

Given these estimates, we use the following intuition for detecting the end of a burst: the more packets that are received, the more likely the burst is over. Assuming that the client knows the sender's congestion window size W (this inference is discussed below) and that the client has received $P < W$ packets, then any of the rest of the (expected) $W - P$ packets in the burst could arrive in $(W - P)I_A$ time. To allow

for variance, we increase the time by $4I_D$, as used in the TCP RTO estimation. Specifically, the dynamic threshold (T_d), which is a function of P, for determining the end of the burst is:

$$T_d(P) = (W - P)I_A + 4I_D$$

This means that the threshold value decreases as more packets arrive during a burst.

The above algorithm assumes that we have an accurate estimate of window size (W) in each round. Our current algorithm to estimate W sets the new window prediction as one more than the number of packets seen in the last window, to mimic what TCP does in its steady state[2].

One problem with our samples of I_A and I_D is that if not all packets arrive (due to packet loss), the estimate of I_A and I_D may not be correct. Our current approach in this case is to double I_D and increase I_A by that amount. Furthermore, in practice the upper bound of T_d is set to the one-way trip time, and the lower bound is set to 3 *ms*, which in our experience is the minimum reliable resolution for our timer.

3.1.2 Round-Trip Time Estimation

Once the end of the window is detected, an *open ack* can be sent and the WNIC transitioned to *sleep* mode. To avoid missing packets, the WNIC needs to be transitioned back to *idle* mode before the first packet of the next burst. That packet will arrive approximately RTT *ms* after the *open ack* is sent. This means that we must keep an accurate estimate of the RTT on the client.

Traditionally, TCP only measures RTTs at the sender side for data packets; this is used for setting retransmission timers. On the receiver side, it is difficult to estimate RTTs because it is hard to associate incoming packets with the outgoing acknowledgement that triggered them [23]. Even though in our particular case a single *open ack* triggers an entire burst of packets, variation can make the association difficult. Fortunately, the TCP timestamp option provides accurate RTT measurements when both the sender and the receiver agree to use it on a connection. Once enabled, up-to-date timestamps are always sent and echoed in the TCP header of each packet. Upon receiving a packet, either endpoint can calculate a new RTT sample as the time difference between the current timestamp value and the echoed value. TCP uses these accurate RTT samples to improve the quality of the TCP RTO estimate, which in turns improves TCP performance. As a result, presently the TCP timestamp option is an extension used in most TCP implementations [24]. In fact, in all

[2]Slow start and packet losses are handled separately in a different state; the states and their transitions are described in Section 3.2.5.

the servers in our experiments, the timestamp option was enabled.

By default, the TCP receiver does not measure RTT for acknowledgements. According to RFC 1323 [25], TCP is not required to echo back timestamps for anything other than data packets. In practice, however, the timestamps of the acknowledgement are echoed back in the data packets triggered by the acknowledgement. We added measurements at the TCP receiver to produce RTT samples for acknowledgement packets (as well as data packets) sent back to the server.

Given accurate RTT estimates, our approach is to use the the minimum RTT over all samples. This is a conservative approach, resulting in some wasted energy but also almost always avoiding missed packets on the client. In practice, this scheme performed quite well (see Section 4).

If the timestamp option is not enabled, currently we revert to standard TCP. However, we believe that by using techniques such as those presented in [23] to associate packets with their acknowledgements, we could obtain accurate RTT estimates. We leave this for future work.

3.2 Challenges

In this section, we discuss the challenges that arise when implementing our client-centered technique. These challenges include lost packets, saturated connections, lost *open acks*, and excessive delay due to delayed acknowledgement. We conclude with our overall algorithm.

3.2.1 Packet Loss

The client cannot distinguish between loss caused by the WNIC residing in *sleep* mode and network loss. Either way, loss is an indication that there may be significant variation on the Internet during a download. Hence, when the client detects at the end of a burst that at least one packet is missing, it switches to what we call *recovery state*. This mode keeps the WNIC in *idle* mode while still forcing bursts. This avoids missing packets until the variation ceases. We switch back to energy-saving state as soon as an entire burst is received intact. If three consecutive bursts have missing packets, we switch instead to standard TCP. In our experiments, entering recovery state was rare; Section 4 breaks down the energy consumed, including that incurred by recovery state.

3.2.2 Saturated Connection

Energy saving is only possible when the bandwidth of the wireless network is not fully used. This is because there must be time that the WNIC can be transitioned to *sleep* mode. Although infrequent in our experience, downloads from certain Internet sites at some times during the day saturate the wireless network. If the client executes the energy-saving algorithm described above, the result will be longer transmission time and *increased* energy usage.

To handle this, the client uses its estimate of window size, wireless bandwidth, and the round-trip time to determine when executing the energy-saving algorithm is not profitable. If the window size is within a threshold (currently 90%) of the bandwidth-delay product (wireless bandwidth multiplied by the round trip time) for three consecutive bursts, we revert to standard TCP. Currently, we do not attempt to resume saving energy if the bandwidth-delay product decreases, because our experience to date is that a saturated connection almost always remains saturated. We leave this subject for future research (see Section 5).

3.2.3 Lost *open acks*

One problem with our technique is that TCP is now vulnerable to the loss of *open ack* packets. Whenever an *open ack* is lost, if the client takes no action, the server will time out and probe the client. This is because in the absence of reception of an *open ack*, the server believes the client has no buffer space to store packets—it has previously received a series of only *closed acks* during the current burst. A timeout causes a large overhead in both energy and time, because (1) the client spends significant time waiting for packets in *idle* mode, and (2) the sender cuts the congestion window size to one packet.

Our current approach to this problem is to have the client wait twice the estimated RTT after the original *open ack*. If no data has arrived, it then retransmits the *open ack*. An *open ack* is retransmitted each RTT *ms* if the next burst does not arrive. In practice, this technique was sufficient in our experiments, because loss of an *open ack* was rare. In particular, our experiments showed that no Internet site we used in our test suite incurred more than one lost *open ack*.

3.2.4 Excessive Acknowledgement Delay

To increase the chance of piggybacking acknowledgments, most TCP implementations by default use delayed acknowledgment. This means that other than during slow start, TCP receivers usually send acknowledge-

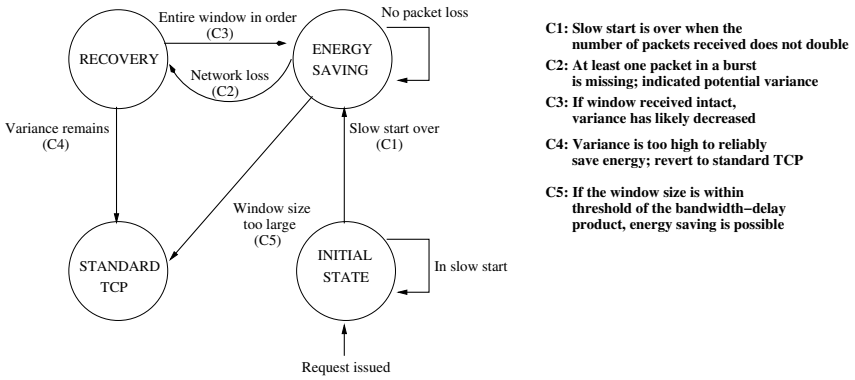

Figure 2: States in our algorithm.

ments every other packet. This is problematic if an odd-sized window occurs. For example, assume that an odd-numbered packet in a given window is received but no even-numbered packet arrives. The TCP receiver delays for some amount of time (40 *ms* in Linux 2.4, for example) before assuming that the even-numbered packet will not arrive. At that point the receiver sends an acknowledgement.

This TCP behavior is not energy efficient, because the time spent waiting for an even-numbered packet is spent in *idle* mode, which wastes energy. Hence, we use what we call *eager acknowledgement*, which means that the *open ack* is sent based on when the client detects the end of a burst (see above) rather than when TCP itself would send the acknowledgement. It should be noted that this speeds up a TCP transmission. To ensure a fair comparison, we use an implementation of TCP that also uses eager acknowledgement (called "baseline TCP" in our experiments).

3.2.5 Overall Algorithm

Our overall algorithm is shown pictorially in Figure 2. The WNIC is kept in high-power mode in all but the *energy-saving* state, in which our client-centered technique is being applied. From our initial state, we transition to our energy-saving algorithm when slow start is over, which the client detects when the number of packets received does not double. We move to recovery state if a burst is received out of order, and move back to saving energy as soon as a window is received in order. In two cases, we revert to standard TCP: (1) the connection is saturated, or (2) packets are received out of order for three consecutive windows.

3.3 Netfilter

We implemented our prediction-based algorithm in a Linux kernel module. It is based on *Netfilter* [6], which is a general framework of hooks in the network stack of Linux 2.4. *Netfilter* is the basis for the implementation of IP tables and IP chains. It supports packet filtering, packet mangling and network address translation (NAT) inside the Linux kernel.

Our implementation filters each incoming and outgoing packet on the client, applying the techniques discussed above. It also maintains the current state of the WNIC. For each incoming packet, we either pass the packet on to the client (if the state of the WNIC is high-power mode) or drop the packet (if the state of the WNIC is *sleep* mode). We patched our kernel using the KURT microsecond resolution timer [26] for our Linux client, because we need to be able to sleep at the granularity of a millisecond.

Because we use a kernel module, we can run actual Internet experiments as opposed to just simulations. Using this implementation we are able to observe the real-world (detrimental) effects of a missed packet, as well as gain insight into the effectiveness of our solution using real (unmodified) servers.

4 Performance

This section describes our experiments and presents our results. Section 4.1 describes our experimental methodology. Next, Section 4.2 presents our overall experimental results on 7 Internet servers as well as a variety of *DummyNet* experiments. Finally, Section 4.3 compares our system to traditional 802.11b power saving mode (PSM).

4.1 Experimental Methodology

In our experiments we examined the performance of our system on both real Internet traffic using actual servers and emulated traffic using *DummyNet*. The experimental setup is shown in Figure 3. In both sets of experiments, the wireless client was emulated by a 1GHz Pentium desktop machine running Linux 2.4-18 with *Netfilter*. The access point is emulated using another 1GHz desktop running FreeBSD 4.5 -stable; we used tunneling so that *DummyNet* [5] could be used to experiment with different wireless bandwidths. Our tests used a 20Mb/s bandwidth between access point and client. These values were selected by experimenting with a 54 Mb/s access point and measuring the peak bandwidth attained on a wireless client. In practice, most TCP connections cannot achieve a 20Mb/s speed, because of the way that default socket buffer size is chosen

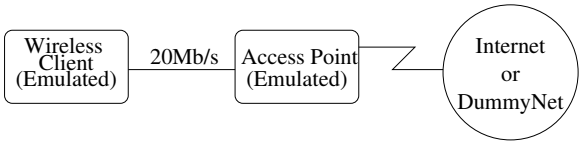

Figure 3: Our experimental setup.

Site	*Base RTT* (*ms*)
uiuc.edu	21
sun.com	50
freebsd.org	57
cs.stanford.edu	58
cs.washington.edu	61
jriver.net	63
intel.com	91

Figure 4: Sites used in Internet tests with base RTT (without variations).

in most systems (e.g., for a 64KB TCP socket buffer, at a 40 *ms* RTT, the maximum transfer speed is below 20 Mb/s). Hence, the wireless link is not the bottleneck, which allows energy saving.

Form of each experiment In each experiment the client performs an `ftp` download from the server, requesting a file between 4 and 5 MB (depending on the particular site). For each experiment, we ran both our algorithm and a baseline test. The baseline test (denoted "baseline TCP") used the standard TCP implementation in the Linux kernel with the addition of eager acknowledgements. The baseline TCP version keeps the WNIC in high-power mode for the duration of the download. We compute normalized energy savings and transfer times by comparing to the baseline. In all experiments, we performed each test three times and report the results from the test with the median transfer time.

Energy*Delay Metric We determine the energy consumption by capturing a trace during execution via `tcpdump`. We evaluate this trace postmortem to compute energy consumption and record the transmission time. A simulator reads the trace and models a 2.4Ghz WaveLAN DSSS WNIC, which uses 1319 mJ/s when idle, 1425 mJ/s when receiving, 1675 mJ/s when transmitting, and 177 mJ/s when in sleep mode [27, 28]. Note that this particular card does not handle a 20Mb/s rate, but we believe the energy consumption of such cards will show similar ratios. The simulator calculates how much time a client's WNIC has spent in high- and low-power mode so that total WNIC energy can be computed. We normalize the energy*delay product to a baseline TCP stream, where the WNIC remains in a high-power mode for the duration of the experiment. Note that the baseline experiment does *not* use *Netfilter*, avoiding any overhead that might be added. Also, we model the energy cost of transitioning the WNIC from *sleep* to *idle* mode as 2 *ms* in *idle* time [10]. The simulator also computes several additional quantities (like a breakdown of wasted energy) for analysis

purposes.

It is important to note that in this paper we consider only the energy consumed during actual TCP transmissions. We do not measure energy consumption during user inactivity.

Experiments The Internet experiments were carried out by running a script that downloaded a file at a time, in succession, from seven Internet servers shown in Figure 4 (previous page). The client machine accessed the Internet via the tunnel. Experiments were performed between 2-5PM EST (peak Internet usage times) during the week of October 26, 2003. Additionally, in order to study our system in an environment where experiments are mostly repeatable, we used *DummyNet* to construct a emulated environment in which we could control loss rates and round trip times. Our emulations used a variety of round trip times (30 *ms*, 60 *ms*, 90 *ms*, and 120 *ms*) and loss rates (several between 0% and 1%). For these emulations, we wanted to model as closely as possible the round-trip time variation that occurs in our Internet experiments. Unfortunately, *DummyNet* does not handle round-trip time variation; it uses only fixed delay values for a particular path. To address this, we modified *DummyNet* to add RTT variation without causing out-of-order packet arrivals, as was done in [11]. We model round trip time variation using an uncorrelated gamma distribution [29]. We obtained this distribution by performing a ping test during peak time, gathering several thousand samples from the `ftp.cs.washington.edu` server (which has a 61 *ms* base RTT), and then using these samples to determine the parameters to the gamma distribution. Note that while a realistic distribution is likely correlated, (1) *DummyNet* itself cannot handle such a distribution, and (2) an uncorrelated distribution is *more* difficult for our system to handle effectively, due to the increased unpredictability of round-trip times. To simulate peak traffic on round-trip times other than 60 *ms*, we scaled this gamma distribution proportionally to the new round-trip time.

4.2 Overall Results

In this section, we discuss the performance of our system in comparison to baseline TCP. We normalize the energy*delay product to 1 for baseline TCP. We analyze the performance of our client-centered technique in two environments. First, Figure 5 shows the energy*delay product when downloading files from several Internet servers. Second, Figure 6 shows the same metric using emulated traffic with *DummyNet* (tests described above). In both environments, our system is effective at saving energy while maintaining reasonable download speeds. In the Internet experiments, our system on average performs 19% better than baseline

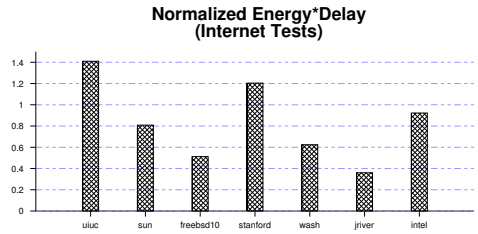

Figure 5: Normalized energy*delay product for Internet tests using a 20 Mb/s access point.

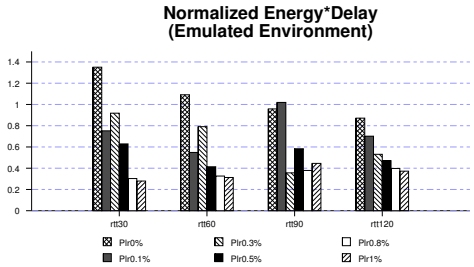

Figure 6: Normalized energy*delay product for *DummyNet* tests using a 20 Mb/s access point.

TCP. In our emulated environment our system outperforms baseline TCP by 54%.

We draw two main conclusions from these performance results. First, as round-trip time increases, the performance of our client-centered technique (in energy and time) improves. This is because two overheads inherent to our technique are small: (1) the relative overhead due to stream delay caused by *closed acks* and (2) the relative overhead due to inferring the end of a burst. Second, as the loss rate increases, the performance of our client-centered technique also improves. One reason for this is that an increase in loss results in a decrease in average window size, which generally reduces the overhead due to stream delay caused by *closed acks*. Furthermore, even under loss, our end-of-burst detection and RTT estimation works well, as the client does not miss many packets. In particular, over all seven Internet sites, the client missed less than one packet on average due to incorrect predictions. Finally, the specific FreeBSD implementation of TCP is more amenable to our technique when there is high packet loss (see below). Figure 6 clearly shows the superiority of our client-centered technique as both RTTs and loss increase.

The Internet test cases show similar behavior. Specifically, the best Internet test case, `jriver.net`, has a normalized energy*delay product 64% better then baseline TCP. This is because this particular site has a small average window size. In total, five of the seven sites had an energy*delay less than 1. However, there are two poorly performing sites that significantly lower our average improvement: stanford and uiuc. For `cs.stanford.edu`, our client-centered technique was 45% slower than baseline TCP, 7.48 seconds to 5.15 seconds. Analyzing the trace produced shows that the predicted estimated slowdown is about 42% (1.95 seconds for stream delay caused by *closed acks* and 0.2 seconds for inferring the end of a burst). The large delay from *closed acks* is due to the large average window size for this transmission. It should be noted that as wireless network bandwidth increases (soon to potentially 108 Mb/s peak speed), the overhead from *closed acks* will drop considerably. For example, in the `cs.stanford.edu` case, it would drop from 1.95

seconds to less than 0.4 seconds. This would decrease the energy*delay product from 1.2 to 0.89. The other poorly performing site with our technique is `cs.uiuc.edu`. This site has a round-trip time of 21 *ms*, and the bandwidth-delay product is large enough that our system reverts to standard TCP. The reason for the poor performance is (1) startup overhead, when our client is trying to synchronize into energy-saving mode, and (2) *closed ack* overhead, which occurs in the three round trips before reverting. The entire 5MB file takes only 2 seconds to download, so these overheads are not well amortized.

One last observation of the emulated experiments is that our technique actually results in *faster* transmission time than baseline TCP at high loss rates. This counterintuitive result is due to the particular behavior of the TCP implementation of BSD. When the server receives a *closed ack*, it enters persistent state. Then, it will begin sending packets starting from the following *open ack*. On the other hand, baseline TCP incurs fast retransmissions or timeouts, which reduces the window size and adds overhead.

4.3 Comparison to 802.11b Power-Saving Mode

We implemented a software version of 802.11b power-saving mode (PSM) [2] so that we could compare it with our client-centered technique. Our implementation of PSM uses a transparent proxy that intercepts packets before they reach the access point and emulates access point PSM behavior. It buffers packets and, every 100 *ms* (the beacon period used by an Orinoco access point), it sends a beacon packet that indicates if any data is buffered. The client responds with an ICMP packet (which emulates the "poll" frame), and then the proxy sends all buffered data to the client. The last packet in a burst is marked so the client can immediately transition the WNIC to *sleep* mode.

It is important to note that this implementation of PSM is essentially optimal. We compute the energy postmortem via a client trace and assume no early wakeup whatsoever. Hence, the client is awake waiting for packets only for the time it takes (1) the ICMP packet to travel from the client to the proxy and (2) the first data packet to travel from the proxy to the client. This time is less than 1 *ms*. In practice, PSM does not work this efficiently; in particular, Chandra and Vahdat found via direct measurement of access points that clients were often kept waiting (in high-power mode) for data after sending the poll frame [8]. Finally, PSM requires a poll for each packet, while our emulation uses one poll per burst. Taken together, PSM is unlikely to perform in practice as well as it performs in our emulation.

Figure 7 shows the energy*delay product for baseline TCP, PSM and our technique, normalized to baseline TCP. At low RTTs, our technique significantly outperforms PSM, primarily because PSM increases the

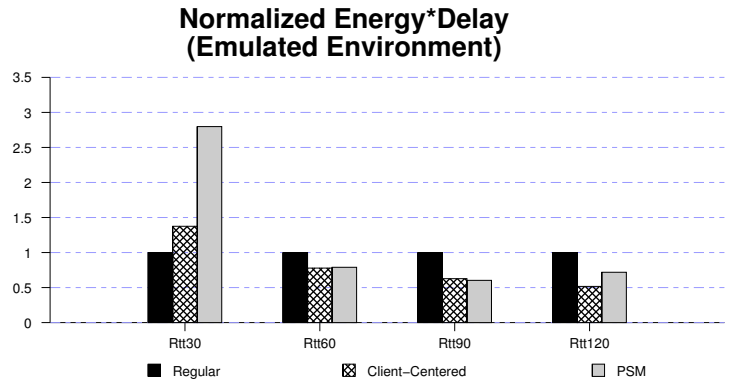

Figure 7: Analysis of energy consumption using our technique, baseline TCP, and PSM. We emulated a 60*ms* round trip time, with no loss, and a 20 Mb/s access point.

transmission time by more than a factor of 3. A similar effect is seen at 120 *ms*—PSM also performs poorly because the RTT is effectively 200 *ms*. As the RTT increases (but is less than 100 *ms*), PSM improves in a relative sense, because the effective RTT increase is smaller. At an RTT of 90 *ms*, PSM performs better than our technique because that RTT is close to optimal for PSM performance.

5 Discussion and Future Work

The previous section shows that our client-centered technique is effective in practice. This section first discusses the effect of bursts. Next, we describe the limitations of our technique. Finally, we discuss future work.

Effect of Bursts Our technique has the potential to affect the queuing behavior of routers, since it purposely introduces burstiness into the packet stream. However, we do not expect the effect to be a significant problem. This is for two reasons. First, individual packet streams already experience some amount of burstiness due to slow start, missing acknowledgements, or blocking at the application layer [30]. In fact, any wireless device utilizing IEEE 802.11b power-saving mode (PSM) [2] will introduce burstiness into the network—independent of our algorithm—because PSM results in ack compression at the client. Our approach differs only in that it attempts to exploit the burstiness, and hence *controls* it rather than allows it to happen in an arbitrary way.

Second, because our client-centered technique increases transmission time, the amount of data per unit

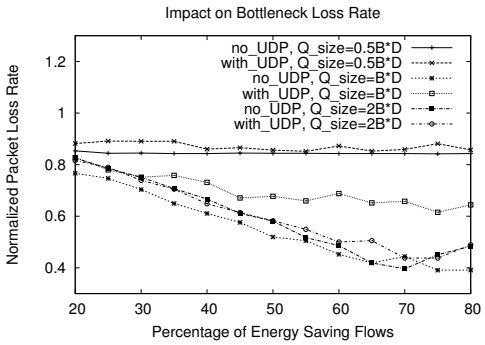

Figure 8: Number of packet losses seen at the bottleneck for a mix of energy-saving TCP flows and standard TCP flows. The number of losses is normalized based on the number of losses when all 100 flows are standard TCP.

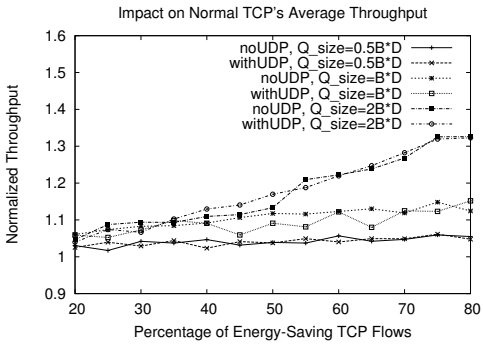

Figure 9: Average throughput of standard TCP flows when competing with a different number of energy-saving TCP flows. The throughput is normalized based on the average throughput of 100 standard TCP flows.

time that passes through routers *decreases*, thereby having a net positive effect on queue length. We studied the interaction of our energy-saving TCP flows and standard TCP flows using an ns2 [7] simulation. Specifically, we studied the impact of our energy-saving TCP flows on the bottleneck queue and standard TCP flows. Counterintuitively, we found that replacing standard TCP flows with energy-saving TCP flows *reduces* packet loss at the routers and increases the throughput of standard TCP flows.

The ns2 simulation is performed on a dumb-bell topology with a total of 100 competing TCP flows. Each flow is either a standard TCP flow or an energy-saving TCP flow. To study the impact on standard TCP flows, we measure the number of packet losses that occurred at the bottleneck, as well as the throughput of standard TCP flows. We increase the number of energy-saving TCP flows to see their impact. We also adjust the queue size of the bottleneck link to be to be 0.5, 1, and 2 times the bandwidth-delay product to simulate different buffering conditions.

The measurement result is counterintuitive—the loss rate of the bottleneck does not increase as the number of energy saving TCP flows increase, although these flows shape traffic into bursts. In most of the cases the loss rate actually decreases, and the average throughput of standard TCP flows increases. The loss rate result is presented in Figure 8, and the average throughput of standard TCP flows is presented in Figure 9.

We also performed experiments with additional UDP traffic that introduces random transient congestion. Standard TCP flows still perform better as the number of energy-saving flows increases. By inspecting packet-level traces, we discovered that the burst size of the energy-saving TCP flows is not arbitrarily large but rather is limited by congestion control. In addition, the energy-saving TCP flows actually compete for

bandwidth less aggressively than standard TCP flows. As a result, as the number of energy-saving TCP flows increases, the performance of standard TCP flows improves.

Limitations This section discusses our limitations. First, in this paper we support large file downloads. This type of traffic is two way (request/reply) and predictable. However, there are potentially many kinds of traffic that can arrive at a wireless client. For example, `ARP` traffic as well as voice-over-IP are one way and unpredictable. Our approach cannot be used for such traffic. For many of these kinds of traffic, if a packet is missed because the WNIC is in *sleep* mode, it is not critical. For example, a client can temporarily ignore `ARP` packets (e.g., from peers on the wireless network). A client clearly cannot use our energy-saving techniques for downloads while using an application such as voice-over-IP. However, we believe our approach handles the most common cases.

Second, our implementation uses KURT millisecond timers and extra computation to maintain state. This will increase the energy consumed compared to using baseline TCP. While a complete analysis would include all of this energy, the effect is likely small and is certainly difficult to quantify. Hence, we have not included these effects in our performance measurements. (Note that newer Linux versions are moving toward making millisecond-accurate timers standard.)

Finally, we do not consider the effect of the mobile client moving between access points. Clearly, if such a scenario were considered, the WNIC would need to remain in high-power mode during periods of roaming.

Future Work This paper has focused on reducing energy*delay during TCP downloads for a single mobile client in a client-centered manner. We view this work as a first step towards our goal of providing proxy-free energy savings to multiple clients sharing an access point on TCP downloads. Supporting multiple clients adds extra complexity because there is variation on the wireless link. (Initial experiments reveal that the access point schedule for sending packets to multiple clients is quite arbitrary.)

Also, we plan to investigate several aspects of when to revert to baseline TCP (because a connection is saturated) and when to resume our energy-saving technique. In general, to determine whether or not to revert, we need to consider the bandwidth-delay product, the ratio of *idle* energy to *sleep* energy, and the overhead to infer the end of the burst. We intend to consider all factors in improving our current algorithm. In addition, we could also revert to baseline TCP when a user-specified maximum time increase would be otherwise exceeded. Resumption of energy-saving mode would involve taking the above factors into account

after the connection is deemed saturated, monitoring the bandwidth-delay product, and changing state when appropriate.

6 Conclusion

In this paper we developed a novel method to allow mobile clients to exchange transmission speed for energy savings during TCP downloads in a *client-centered* manner. The fundamental idea for long streams is that a client shapes traffic from the server into bursts, which allows the WNIC to be placed in *sleep* mode for longer periods of time. Results show that our system performs up to 64% better than baseline TCP on the Internet, as measured by energy*delay. Our technique also does quite well on average over all our Internet sites, outperforming TCP by 19% on average. Furthermore, we did not rely only on simulations; we tested our technique on existing Internet servers.

Our technique relies only on leveraging the design of the TCP protocol. Therefore it is in a sense hardware independent, in contrast to techniques that use proxies or IEEE 802.11b power-saving mode. This means that our technique is flexible, in that it can be applied on any wireless network infrastructure. We believe this is a building block towards allowing multiple clients on a wireless network to save energy.

References

[1] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Symposium on Operating Systems Principles*, pages 48–63, December 1999.

[2] IEEE Computer Society LAN/MAN Standards Committee. IEEE Std 802.11: Wireless LAN medium access control and physical layer specification. Technical report, August 1999.

[3] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *ISLPED 1998*, August 1998.

[4] F. Douglis, P. Krishnan, and B. Bershad. Adaptive disk spin-down policies for mobile computers. In *Proc. 2nd USENIX Symp. on Mobile and Location-Independent Computing*, 1995.

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

[6] Netfilter. `http://www.netfilter.org`.

[7] Network Simulator-2. `www.isi.edu/nsnam/ns/`.

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

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

[10] Ronny Krashinsky and Hari Balakrishnan. Minimizing energy for wireless web access with bounded slowdown. In *Mobicom 2002*, Atlanta, GA, September 2002.

[11] M.C.Chan and R.Ramjee. TCP/IP performance over 3G wireless links with rate and delay variation. In *Mobicom 2002*, Atlanta, GA, Sep 2002.

[12] Hari Balakrishnan, V.N. Padmanabhan, and R.H. Katz. The effects of asymmetry on TCP performance. In *Mobicom 1997*, Atlanta, GA, Sep 1997.

[13] K.Brown and S.Singh. M-TCP: TCP for mobile cellular networks. In *ACM Computer Communications Review*, Atlanta, GA, Sep 1997.

[14] Eugene Shih, Paramvir Bahl, and Michael Sinclair. Wake on wireless: An event driven energy saving strategy for battery operated devices. In *Mobicom 2002*, Atlanta, GA, September 2002.

[15] Athanasios E. Papathanasiou and Michael L. Scott. Energy efficiency through burstiness. In *WMCSA*, October 2003.

[16] M. Weiser, B. Welch, A. J. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Operating Systems Design and Implementation (OSDI '94)*, pages 13–23, 1994.

[17] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In *Architectural Support for Programming Languages and Operating Systems*, pages 105–116, 2000.

[18] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *ASPLOS X*, October 2002.

[19] Manish Anand, Edmund Nightingale, and Jason Flinn. Self-tuning wireless network power management. In *Mobicom*, September 2003.

[20] S. Singh, M. Woo, and C. S. Raghavendra. Power-aware routing in mobile ad hoc networks. In *Mobile Computing and Networking*, pages 181–190, 1998.

[21] R. Kravets, K. Schwan, and K. Calvert. Power-aware communication for mobile computers. In *Proc. 6th International Workshop on Mobile Multimedia Communications*, Nov 1999.

[22] Haijin Yan, Rupa Krishnan, Scott A. Watterson, David K. Lowenthal, Kang Li, and Larry L. Peterson. Client-centered energy savings for TCP downloads. Technical report, University of Georgia, February 2004.

[23] Guohan Lu and Xing Li. On the correspondency between tcp acknowledgment packet and data packet. In *ACM Internet Measurement Conference 2003*, Oct 2003.

[24] Richard Wendland. "How prevalent is timestamp options and paws". Web survey result published in end-to-end interest list, 2003.

[25] V. Jacobson, R. Braden, and D. Borman. Rfc 1323: Tcp extensions for high performance, May 1992.

[26] Kansas Unversity Real-Time Linux. `http://www.ittc.ku.edu/kurt/`.

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

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

[29] A. Mukherjee. On the dynamics and signicance of low frequency components of internet load. *Internetworking: Research and Experience*, 5(4):163–205, December 1994.

[30] Jiang Hao and Constantinos Dovrolis. Source-level IP packet bursts: Causes and effects. In *ACM Internet Measurement Conference 2003*, Oct 2003.