

Client-Centered, Energy-Efficient Wireless Communication on IEEE 802.11b Networks

Haijin Yan, Scott A. Watterson, David K. Lowenthal, Kang Li, Rupa Krishnan, and Larry L. Peterson

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 device is to transition its WNIC to a lower-power *sleep* mode when data is not being received or transmitted. In this paper, we investigate *client-centered* techniques for energy efficient communication, using IEEE 802.11b, within the network layer. The basic idea is to conserve energy by keeping the WNIC in high-power mode only when necessary. We track each connection, which allows us to determine inactive intervals during which to transition the WNIC to *sleep* mode. Whenever necessary, we also shape the traffic from the client side to maximize sleep intervals—convincing the server to send data in bursts. This trades lower WNIC energy consumption for an increase in transmission time. Our techniques are compatible with standard TCP and do not rely on any assistance from the server or network infrastructure. Results show that during Web browsing, our client-centered technique saved 21 percent energy compared to PSM and incurred less than a 1 percent increase in transmission time compared to regular TCP. For a large file download, our scheme saved 27 percent energy on average with a transmission time increase of only 20 percent.

Index Terms—Client-centered, energy, wireless networking, TCP.

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, even when all components of an IBM 560X laptop are active, the WNIC accounts for 15 percent of overall system energy [1].

To enable energy savings, WNICs are designed with multiple power modes. Generally, there are four power modes. The *idle*, *receive*, and *transmit* modes are all high energy modes that consume significant energy, while *sleep* is a low-power mode and consumes nearly an order of magnitude less energy.

The basic idea to reduce energy consumed by a WNIC is to transition it to a lower-power *sleep* mode when data is not being received or transmitted. We call this mechanism *energy-efficient wireless communication*. This normally requires packets to be sent in bursts at agreed-upon intervals in order to allow the wireless client to keep its WNIC in a lower power-consuming *sleep* state.

Link-layer mechanisms such as IEEE 802.11b power-saving mode (PSM) [2] reduce energy dissipation without any consideration of upper-layer applications and connections. However, PSM buffering increases connection round-trip times from 16 percent to 232 percent [3]. For applications such as Web browsing, the increase in effective round-trip time is unacceptable.

The focus of this paper is to investigate techniques for network-layer, energy-efficient wireless communication on 802.11b networks to support popular mobile applications. We propose a set of client-centered techniques for saving energy at the WNIC *during* transmission. The fundamental ideas are that the client actively tracks connections, predicts when packets will arrive, possibly shapes traffic, and keeps the WNIC in high-power mode only when necessary. Our technique allows mobile clients to save energy in a *client-centered* manner, i.e., *without* any assistance from servers, proxies, or IEEE 802.11b power saving mode (PSM) in the access point [2]. In this paper, we concentrate our efforts on handling TCP streams initiated by the client to a server via request/response. Our client-side implementation can be easily deployed at individual end hosts without any modifications to the servers.

Our implementation is within *Netfilter* [4], a kernel-level packet filter. Our techniques are performed strictly by the client at the network layer by exploiting TCP congestion control and flow control. Its client-centered manner ensures the technique can be easily deployed through individual end host upgrades. We apply our technique to support the two dominant wireless applications: Web browsing and large file downloads, where a significant amount of WNIC energy is expended.

To test our client-centered technique, we ran actual tests on the Internet as well as in emulated and simulated environments. When meaningful, we compared the results

- H. Yan is with Motorola, 1000 Technology Way, Libertyville, IL 60048. E-mail: yan@cs.uga.edu.
- S.A. Watterson is with the Research and Development Group, Scientific Games Corp., Alpharetta, GA 30004. E-mail: scottwatterson@gmail.com.
- D.K. Lowenthal and K. Li are with the Department of Computer Science, University of Georgia, Athens, GA 30602-7404. E-mail: {dkl, kangli}@cs.uga.edu.
- R. Krishnan is with the Department of Computer Science, SUNY-Stony Brook, Stony Brook, NY 11794. E-mail: krishnan@cs.sunysb.edu.
- L.L. Peterson is with the Department of Computer Science, Princeton University, Princeton, NJ 08544. E-mail: llp@cs.princeton.edu.

Manuscript received 14 Feb. 2005; revised 7 Oct. 2005; accepted 24 Nov. 2005; published online 15 Sept. 2006.

For information on obtaining reprints of this article, please send e-mail to: tmc@computer.org, and reference IEEECS Log Number TMC-0032-0205.

of our system with both PSM and a related approach known as the Bounded Slowdown Protocol (BSD) [3].

For Web browsing, our technique combines the performance of regular TCP and BSD with nearly all the energy-saving of PSM, and we save more energy than PSM during client think times. Our experimental results show that over an entire Web browsing session that includes downloads and think times, our scheme saves up to 21 percent energy compared to PSM and incurs less than a 1 percent increase in transmission time compared to regular TCP. In addition, our technique saves up to 25 percent more energy than BSD.

For large file downloads, in the best case, we save over 50 percent energy compared to baseline TCP with a transmission time increase under 8 percent. Over all seven Internet sites we tested, the average energy savings was 27 percent and the transmission time increase was 20 percent. In an emulated environment (using *DummyNet* [5]), the average energy savings was 51 percent when simulating off-peak hours.

The rest of the paper is organized as follows: Section 2 describes related work. Section 3 discusses the design and implementation of our client-centered techniques. Section 4 describes our experiments and discusses the results. Finally, Section 5 summarizes this paper.

2 RELATED WORK

To reduce WNIC energy consumption, one can use the energy-saving mechanisms defined by IEEE 802.11b (power-saving mode, or PSM) [2]. PSM is a general approach that works at the wireless link layer between the access point and the client. However, using PSM generally increases round-trip times to at least the nearest multiple of 100 ms^1 [3]. Furthermore, PSM is not a good match for applications that receive data at a steady and frequent rate, as an access point using PSM can in some cases cause additional (unexpected) packet delay [6].

One improvement to PSM is the Bounded Slowdown Protocol (BSD) [3], which uses minimal energy given a desired maximum increase in round-trip time. BSD maintains the WNIC in high-power mode during almost all of a TCP transmission (see Section 4), while bounding the transmission slowdown based on a user-supplied parameter. It is aimed at situations where there are long periods of user inactivity.

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 percent, T is 200 ms . Because BSD is implemented in the link 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 and no

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.

energy can be saved. For energy saving, the goal of BSD is reducing energy consumption during periods of user think time, not *during* a transmission.

Our approach, on the other hand, saves energy during transmissions. Our work improves upon this research by 1) saving energy while incurring little or no transmission time increase for short file downloads and 2) trading energy for transmission time for large file downloads. In addition, BSD requires modification to the access point. Our approach requires no new hardware and works solely based on changes to the client TCP implementation.

There has also been work done on reducing idle energy in the network interface [7]. This has potential to improve energy usage, but cannot be used on current hardware. The basic ideas of power-aware routing in wireless multihop networks are to perform routing in a power-aware manner or to integrate power awareness into the transport layer [8]. Additionally, there has been work in investigating power-aware mechanisms for end-to-end communication in wireless networks [9]. 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.

One body of work that is 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 [6], [10]. The disadvantage of these approaches is that they require a proxy, which will not exist on many wireless networks; our method is implemented on mobile clients, requiring no assistance from a proxy or 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, where the use of a proxy at the network layer requires a "double connection," which violates TCP's end-to-end semantics.

Prior work has substantially advanced power-aware computing at the hardware and operating system level. At the hardware level, dynamic voltage scaling (DVS) is a technique that allows processor speed to be decreased in order to run at a lower energy level (e.g., [11]). OS scheduling can then take advantage of DVS [12]. At the operating system level, there has also been work in creating burstiness to save energy consumed by disks [13] as well as work that uses program-counter-based techniques to determine when to spin down the disk [14]. Recent work has advocated managing energy explicitly as a resource in the operating system [15]. Another approach is to have the OS exploit application information to save energy [1]. Our work is similar in spirit to all of these techniques, but we focus on creating burstiness in network transmissions and predicting when packets will arrive.

Other researchers have had the idea of using RTT to transition the WNIC into *idle* mode [16]. However, this work only considers that the wireless device is the *sender* (which is a different problem). They do note the effect that smaller TCP window sizes result in more energy savings, a result which we also observed.

| Id | Status | Site | Stage | Num Packets | Next Stage | | Current Expiration | SRTT | Var. |
|----|-----------------|-----------------|-------|----------------|------------|-----|-----------------------|------|------|
| | | | | | Start | End | | | |
| 1 | <i>idle</i> | www.cnn.com | 7 | — | 140 | 160 | — | 50 | 5 |
| 2 | <i>active</i> | www.espn.com | 5 | 6 | 232 | TBD | 102 | 143 | 12 |
| 3 | <i>idle</i> | www.cnn.com | 12 | — | 147 | 174 | — | 50 | 5 |
| 4 | <i>finished</i> | www.cnn-ads.com | — | — | — | — | — | 20 | 5 |

Fig. 1. Sample connection table with four concurrent connections (two *idle*, one *finished*, and one *active*). *TBD* means that the value has not yet been determined.

Controlling the TCP sender's behavior via manipulation of acknowledgements by the receiver is not a new idea. In [17], Chan and Ramjee 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 [18], 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 advertise a zero-sized receiver window. Another idea is the Reception Control Protocol (RCP), which allows receiver control of the sender [19]. A protocol that is similar in spirit is TCP-Real [20], which uses a "wave" pattern in order to avoid congestion and detect and classify errors—similar to RCP, this protocol is receiver oriented. However, both RCP and TCP-Real are new protocols and require deployment on both connection ends, whereas our client-centered approach requires only client-side deployment.

Also, studies have been done of the energy consumed by different versions of TCP (Reno, Newreno, SACK) [21] as well as by the different actions within TCP (i.e., energy cost for copying data, radio, etc.) [22]. The latter work also motivates ways to conserve energy; for example, one way is to maintain the TCP send buffer on the NIC itself, and another is to transfer data between the kernel and the NIC in large chunks. In addition, there has been research to determine the energy cost of the wireless network interfaces in an ad hoc network [23].

Another energy-related research area is the study of transmission power control (TPC) in wireless RF channels [24]. Generally, TPC algorithms attempt to reduce energy consumption during the transmission phase. These algorithms affect the link layer, while our work aims to save energy via the network layer.

Our previous work [25] looked at saving energy during small (Web) file downloads, where concurrent HTTP connections are instantiated. We also viewed the problem from an energy-delay perspective [26]; in principle, our goal is to achieve energy efficient wireless communication using IEEE 802.11b in the network layer through application-aware energy saving methods.

3 IMPLEMENTATION

In this section, we describe the design and implementation of our client-centered techniques. We assume that the WNIC modes are *idle*, *receive*, *transmit*, and *sleep*. The first three are referred to as high-power modes, and *sleep* mode is the low-power mode. Any packets (sent from the server) arriving during *sleep* mode are dropped. We assume that

the client OS will transition the WNIC into high-power mode when data is sent from the client, and then we inform the client OS to transition the WNIC between modes based on our predictions.

We first give a detailed description of all the techniques we used for our energy efficient communications, including our connection table, client side round-trip time estimation, traffic shaping, and end-of-burst detection. We then describe their use in the context of two dominant upper-layer applications: Web browsing and large file downloads.

3.1 Connection Table

Central to our algorithm is a connection table that we use to track each TCP connection initiated from the client. This provides all the necessary information to predict 1) when to transition the WNIC from high to low power mode and 2) how long to keep the WNIC in low power mode. The client divides an individual connection into stages, between which transitioning the WNIC to *sleep* mode is possible. Stages naturally appear during TCP slow start and are also created (by us) through traffic shaping during TCP congestion avoidance.

The client builds a *connection table* (Fig. 1) to hold detailed information about each connection, which allows accurate prediction of the next packet arrival time. The connection *status* field reflects four possible states: *active*, *idle*, *finished*, and *saturated* (discussed below). The *site* field is used to index into the site table maintained on the client to locate the detailed information about the remote server site. (The actual table uses IP and port number. For presentation purposes, we present the logical name.) We partition each connection into *stages* and keep the stage number in the *stage* field. We also record the number of packets received in each stage. The next stage *start time* and *end time* are the predicted starting and ending times of the next transmission stage. We use the current estimate of the round-trip time (in the *SRTT* field) and variance (*var*) to compute the start and end times of the next stage. If the connection is *active*, we also keep track of when the current stage is expected to end (*current expiration*).

3.2 Round-Trip Time Estimation

To carry out our energy efficient algorithm, the client must make use of round-trip time information to accurately predict when the next packet will arrive on any connection, allowing for variance present in network transmission. We leverage the TCP timestamp option [27] in each packet to calculate round-trip times. We use smoothed round-trip time and variance estimates to predict first and last packet arrival times for each burst during slow start. We use the recorded minimal round-trip time to conservatively predict packet arrival time during congestion avoidance.

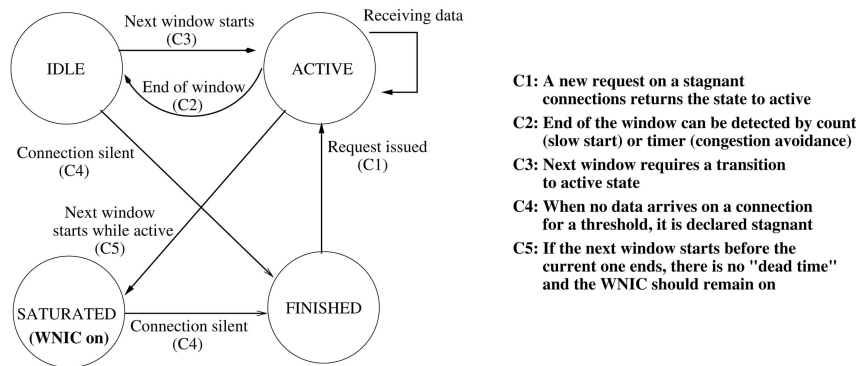


Fig. 2. State machine demonstrating our algorithm.

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 [28]. 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 turn improves TCP performance. As a result, presently, the TCP timestamp option is used in most TCP implementations [29].

Currently, we assume that the timestamp option is enabled. In fact, in all the servers in our experiments, the timestamp option was enabled. If it were not, we believe that by using techniques such as those presented in [28] to associate packets with their acknowledgements, we could obtain accurate RTT estimates. We leave this for future work.

3.3 Connection Tracking

In the following, we discuss our technique to predict the start and end time of each stage when tracking a connection during slow start (without traffic shaping). We will discuss techniques for shaping traffic in Section 3.4.

Whenever stage i ends, the client predicts the stage $i + 1$ start and end time as $T_{first} + SRTT - VAR$ and $T_{last} + SRTT + VAR$. Variables T_{first} and T_{last} are the times the first and last acknowledgements are sent during stage i . $SRTT$ is our smoothed round-trip time estimate and VAR is the estimated variance.

We use the method in Section 3.2 to estimate RTT. The new variance measurement is then the difference between the measured RTT and the $SRTT$. To calculate the $SRTT$ and its variance from each measurement, we employ the same algorithm which is used by many implementations of TCP: The new $SRTT$ is calculated using the formula $7/8 \times SRTT + 1/8 \times RTT$, where RTT is the observed value. (The new variance estimate is computed similarly.)

Transitioning between states: Fig. 2 shows the state transition diagram. A connection is in *active* state when it is receiving data within a stage. A connection is in *idle* state

if it is finished receiving packets from the previous stage and is waiting for the start of the next stage. In general, this is determined by two conditions: 1) the predicted ending time of the current stage has passed and 2) no packet has arrived for T_{thresh} ms (set dynamically based on the maximum observed interval between successive acknowledgements) since the last outgoing acknowledgement was sent. Conditions 1) and 2) combine to serve as safeguards against network delays, which would otherwise lead to prematurely marking a connection as *idle*.

The issue of when a connection is terminated requires a special case for HTTP (as opposed to TCP). As HTTP/1.1 uses a persistent connection for successive HTTP requests, connections are not terminated after transmission of a requested object. We mark the connection as *finished* when the requested page has been fully downloaded (and so the client will not receive data on this connection). This is detected using the Web page size, which was available over 95 percent of the time for the Web sites in our experiments. If not, a connection is not marked *finished*.

The final possibility is that sender's window size is larger than the bandwidth-delay product of a connection, and so there is no possibility for energy savings between stages. In this case, we mark a connection as *saturated*.

3.4 Traffic Shaping

An energy efficient communication mechanism saves the most energy when data is transmitted in bursts. However, when a TCP stream is in congestion avoidance, it does nothing to combine packets into bursts—instead, it attempts to smooth the packet stream—which makes saving energy difficult due to significant consumption when the WNIC is always in *idle* mode. This is because packet arrivals are unpredictable during the smoothed transmission period and so are not amenable to energy savings as compared to the inherent bursts that occur during slow start. To conserve energy during congestion avoidance, we introduce a traffic shaping mechanism on the client side to shape the TCP traffic implicitly into bursts, increasing the time that the WNIC can be placed in *sleep* mode.

TCP uses congestion control and flow control to limit the 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

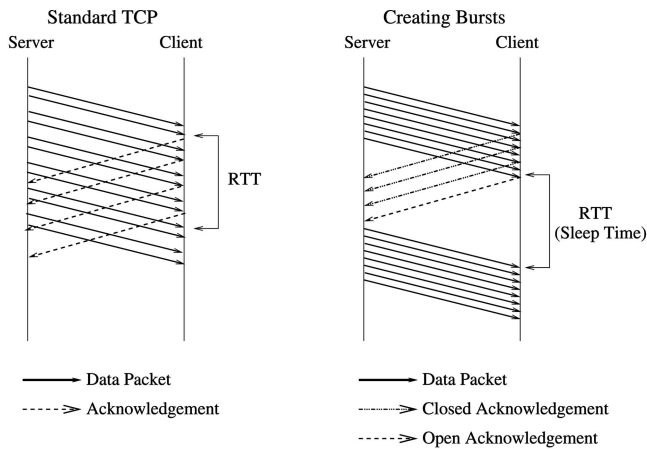


Fig. 3. Example of creating bursts for large file downloads. On the left is standard TCP, where the packet stream is smoothed, and on the right is our client-centered technique. To create bursts, the first three acknowledgements advertise a receiver window size of 0, and the fourth advertises a full buffer. This creates more potential time that the WNIC can remain in *sleep* mode, though the transmission time will increase.

window size² 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 within a window. Instead of sending all packets in a window at once, TCP only allows sending a new packet when another packet is acknowledged. This results in a smooth data stream when the acknowledgments are evenly paced.

Our goal is to shape a TCP stream into bursts to increase potential sleep intervals between packet receptions during congestion avoidance. The basic idea is that the receiver implicitly forces the sender to send each window of packets in a burst so that the potential sleep time between bursts is maximized. 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 the window size advertised by TCP (usually 64KB). 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. Fig. 3 shows the difference between standard TCP and our modified TCP.

2. In this paper, we will refer to congestion window size as simply window size. When we refer to the receiver's window size, we will use the full term.

Our client-centered 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. Then, we discuss two challenges that we face in our implementation.

3.4.1 End of Burst Detection

It is important for our technique to detect the end of a burst as quickly as possible. The detection of the end of a burst is nontrivial because of the variance in round-trip times inherent to the Internet. Conceptually, there is a trade-off between the client inferring the end of a burst aggressively and conservatively. If the client believes incorrectly that a burst is over, packets can be missed. On the other hand, any time the client spends waiting to declare a burst over is in *idle* mode and therefore costs in both time and energy.

Three broad approaches to end-of-burst detection are possible. We briefly discuss the first two, which passively predict the end of a burst through 1) a fixed threshold and 2) a dynamic threshold. Then, we discuss in detail a novel technique we call *active burst detection*, which can, in most cases, precisely determine the end of a burst.

Note that both the dynamic threshold and *active* assume that the client can maintain 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 (i.e., the *previous* value of W), to mimic what TCP does in its steady state.³

Passive burst detection. One general idea for determining the end of a burst is what we call *passive burst detection*, where the client infers the end of the burst passively. The basic idea is for the client to infer the end of a burst when no packets arrive for a threshold amount of time. The first possibility for this is to use a fixed threshold, which is straightforward: If no packet is received for T_f ms, we conclude that the current burst has ended and the client sends an *open ack* to the server. The advantage of such an approach is that it is simple to implement and has minimal overhead. The disadvantage is that it cannot adapt to different network conditions. In particular, when network transfers have low variance (e.g., late at night), it may be excessively conservative. When conditions are poor (e.g., 3:00 pm on a weekday), a fixed threshold may be too aggressive, transitioning the WNIC to *sleep* mode before a burst is complete. Similarly, it could mistakenly transition to *sleep* mode because of the time gap caused by packet loss.

We also studied 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 interarrival time as a sample and then use a scheme that is analogous to what TCP does when computing the timeout value. That is, given new samples for arrival and deviation, we keep a running average of interarrival times (I_A) as well as interarrival deviation (I_D) using a weighted average. Then, if the client has received $P < W$ packets, we use a dynamic threshold (T_d) for determining the end of the burst: $T_d(P) = (W - P)I_A + 4I_D$.

3. Slow start and packet losses are handled separately.

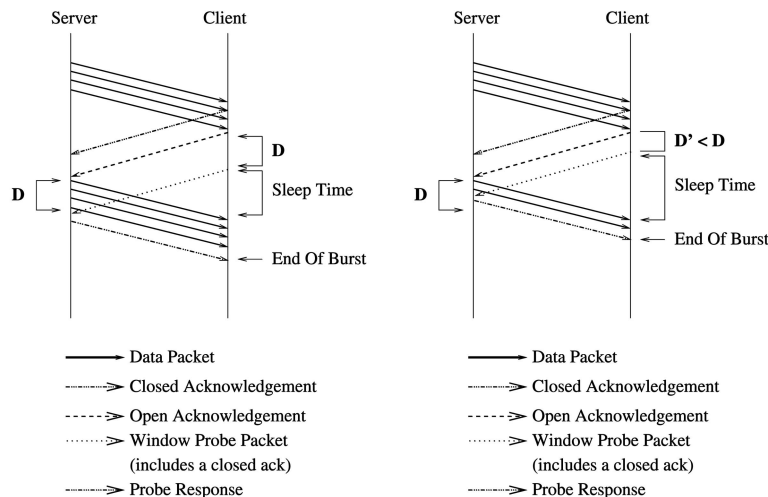


Fig. 4. Example of active burst detection. The left-hand side shows correct operation. The right-hand side shows a case where the window probe packet arrives in the middle of a burst, preventing part of the window from being sent by the server.

Active burst detection. While a dynamic threshold is typically a better idea than a fixed one, both will at times suffer from being either too conservative or too aggressive. As discussed above, this will lead to wasted energy, missed packets, or both. This is especially true when variation in round-trip times is significant, which can occur due to network variation or multiple clients sharing an access point.

We have developed a novel technique for precise end-of-window prediction that we call *active burst detection*. The basic idea is for the client to convince the server to inform it when a burst is over, i.e., the client actively determines the end of the burst. If this can be done, the client can transition the WNIC to *sleep* mode immediately upon this determination. *active* allows in practice for the client to make near-optimal end-of-burst predictions.

The basic idea is as follows (see the left-hand side of Fig. 4): When an *open ack* is sent to the server, the client delays for time D (see below). Then, the client sends a carefully crafted *window probe packet* to the server. In practice, this packet is essentially a zero-length, out-of-order packet. A TCP server responds to such a packet by sending an acknowledgement, which we call the *probe response*. As long as D is sufficiently large so that the probe response arrives after the last packet in the burst, the client can safely declare the burst over and transition the WNIC to *sleep* mode.

Three main issues arise with the use of a window probe packet. First, the server will respond immediately to the window probe packet, even if the entire burst has not been released due to server delay. This would cause the window probe packet to arrive before the end of the burst, leading to an incorrect transition of the WNIC to *sleep* mode. To prevent this problem, we send a *closed ack* immediately after the window probe packet. This way, if there is a server delay, the remaining part of the burst will not be sent until the next *open ack* is received. A related problem is that the window probe packet can be “piggy-backed” on the last data packet if the time between the *open ack* and window probe packet is not large enough.

Second, the choice of D is a trade-off between 1) decreased energy savings and increased time overhead and 2) a larger window size and accurate end-of-burst detection.

In other words, the larger D is, the longer the client waits in high-power mode before sending the *open ack* and the window probe packet. However, the smaller D is, the more likely it is that either 1) the window probe packet sent by the client arrives at the server too early (shown in the right-hand side of Fig. 4), which prevents the sender from sending all of the packets in a window, or 2) the probe response sent from server to client is piggybacked or reordered with at least one data packet somewhere on the network, leading to an incorrect end-of-burst detection.

Our algorithm for determining D is to initialize $D = W/B$. In this formula, W is the client’s estimate of the sender’s congestion window size and B is the estimate of server bandwidth. The idea is that W/B approximates how long it takes the server to release an entire burst. Ideally, if we know the server bandwidth and that there are no variations in round-trip times, we can simply use this D for the lifetime of the download.

However, in practice, neither of these is the case, so we allow D to adapt. To calculate the initial D , we currently set B to 100 Mb/s (because of the popularity of Fast Ethernet). Variable D is increased by 1 ms (the smallest amount the timer can support) in two cases: 1) whenever there is reordering of the probe response that causes it to arrive in the middle of a burst or 2) whenever the probe response is piggy-backed on the last data packet of a burst. We can detect the former situation because the WNIC is not transitioned to *sleep* mode until the window probe packet is sent, which occurs D ms after receiving the probe response. We can detect the latter case through the TCP timestamp or a sub-MSS sized packet.

We decrease D when we have seen P bursts in a row without either of the two cases above occurring (currently, P is set to 10). The decrease in D is half the average gap between the last packet in a burst and the window probe packet. We also provide a floor and ceiling for D ; currently this is 3 ms and the one-way trip time ($1/2$ RTT). If it would be necessary to increase D above the ceiling, we instead switch to the dynamic threshold algorithm, as this indicates that the server is heavily loaded and cannot promptly send packets.

Third, the window probe packet could be lost. When this occurs, the client keeps WNIC in high-power mode for at most a one-way trip time. If a loss occurs at the largest value of D that we consider, we switch to passive detection with a dynamic threshold. In practice, this is rare because of the small size of the window probe packet and low reverse data path loss (see below).

Next Window Prediction. 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. Note that we use the technique described in Section 3.2 to estimate the round-trip time of a connection.

Given accurate RTT estimates, we 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).

3.5 Applications

In this section, we describe how our client-centered energy-saving techniques can be used to support upper layer applications to conserve energy. Ideally, the client would only be in high-power mode while packets are actually arriving and in *sleep* mode at all other times. In practice, the client makes predictions about when packets will arrive, transitioning the WNIC to high-power mode if it expects packets and *sleep* mode if not.

3.5.1 Web Browsing

Modern Web browsers usually invoke multiple concurrent HTTP (and therefore TCP) connections to download embedded Web objects to reduce latency. Prior work observed that Internet Web traffic is highly volatile and bursty in nature [30], [31]. During Web browsing, our technique tracks the connections and predicts packet arrival times to exploit “silent intervals” for energy conservation.

The prediction of the arrival time of incoming packets is nontrivial even when the client has only one outstanding connection. This is particularly true when packets are delayed or lost in the network. Furthermore, the existing Internet introduces many factors that do not exist in simulated networks. These factors contribute to delay and loss in unpredictable ways.

In a client where multiple concurrent connections exist, it is even more challenging to predict the arrival time of the next packet because of accumulated error over many connections. We do two things to solve this problem. First, we use connection tracking technique described in Section 3.3 to track each concurrent connection in the client, collecting detailed information and making predictions. Second, we cache round-trip time information about each site visited by the client to allow us to save energy during connection setup. We discuss each of these techniques in more detail in the following sections.

Extending to Multiple Connections. The technique described above identifies periods when no data is expected on a single connection. Web browsers like Internet Explorer

generally issue multiple concurrent connections to a site to retrieve embedded objects. The extension to handling multiple concurrent connections on the client is straightforward. In particular, we track each active connection in the client and change its state individually. Each time a connection state changes to either *idle* or *finished*, we check all open connections. The client transitions WNIC to *sleep* mode only when *all* connections are *idle* or *finished*, and the client keeps the WNIC in *sleep* mode until the *nearest* predicted next stage starting time of all connections.

In the example connection table shown in Fig. 1, there are four concurrent connections. This is a sample of a connection table taken at time 100. The second connection (marked *active*) is receiving data, the first and fourth connections are between stages (marked *idle*), and the third connection is *finished*.

Round-Trip Time Cache. While the algorithm above is effective in saving energy in many cases, it can be improved. Without prior knowledge, the client cannot predict the SYN/SYN - ACK and GET/GET - ACK round-trips (the GET/GET - ACK is often longer than the SYN/SYN - ACK). In addition, the round-trip time for the actual data can differ from both. Unfortunately, with multiple connections, the chance that *at least* one connection is in either the SYN/SYN - ACK, GET/GET - ACK, or first slow start stage can be large. This makes the solution of keeping the WNIC in high-power mode during these phases nonscalable: For highly concurrent connections, little, if any, energy can be saved.

Because of embedded objects, user Web accesses exhibit high degrees of locality [32]. Therefore, we cache detailed round-trip time information for sites the client has visited within a Web page request (see below), including SYN/SYN - ACK time, GET/GET - ACK time, and the last known *SRTT*. For a given connection C , we start by performing a cache lookup. This information is used to transition the WNIC into low-power mode during the SYN/SYN - ACK and GET/GET - ACK stages as well as the first slow start stage. If there is no entry for C , we revert to the conservative algorithm (no sleep time in setup).

The key issue is how long entries in the cache should remain valid. Clearly, a cached value stored during off-peak hours is invalid during peak hours and will lead to the client to transition to *sleep* mode for too long, meaning that the SYN - ACK packet could be missed. To investigate variance in SYN/SYN - ACK and GET/GET - ACK times, we downloaded Web pages from several sites every half hour over several days. Investigation of the results show in fact that there is little correlation between SYN/SYN - ACK (and GET/GET - ACK) times, even when comparing just peak or just off-peak measurements. Hence, our approach to RTT cache consistency is as follows: Between two Web page requests in our trace file, we flush the RTT cache. Within one Web page request, we fill the RTT cache with entries for each unique Web site accessed. This simple solution has worked well in our experiments (see Section 4).

3.5.2 Large File Downloads

We use traffic shaping techniques and active burst detection described in Section 3.4 to save energy during large file downloads. We transition to our traffic shaping algorithm when slow start is over, which the client detects

when the number of packets received does not double. In two cases, we revert to standard TCP: 1) the connection is saturated or 2) an in-order window is not seen for three consecutive windows.

Next, we discuss the challenges and issues that arise when applying our technique for large TCP downloads: saturated connections, lost *open acks*, and the effect of bursts.

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. During downloads from some Internet sites, the wireless network is saturated. If the client executes the energy-saving algorithm described above, the result will be longer transmission time and *increased* energy usage. Note that when the wireless network is saturated, *no* technique can save energy.

To handle this, the client uses its estimate of window size, wireless bandwidth, and round-trip time to determine when executing the energy-saving algorithm is not profitable. If the window size is within a threshold (currently 90 percent) of the bandwidth-delay product (wireless bandwidth multiplied by the round-trip time) for three consecutive bursts, we revert to standard TCP. 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.

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, even under network conditions during peak times, 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*. This is not surprising, given prior research that indicates that more than 90 percent of loss is on the data path [33].

Effect of Bursts. Our client-centered 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, and blocking at the application layer [34]. In fact, any wireless device utilizing IEEE 802.11b power-saving mode (PSM) [2] will introduce burstiness into the network if the connection is not saturated—independent of our algorithm—because PSM results in ack compression at the client. Our approach,

while incurring slightly more burstiness than PSM, differs primarily in that it attempts to exploit the burstiness and, hence, controls it, rather than allowing it to happen in an arbitrary way. In any case, as effective wireless bandwidths are generally much less than 50 Mb/s, the burstiness caused by a single wireless client should not be too severe.

Second, because CC increases transmission time, the amount of data per unit 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 [35] simulation. Somewhat 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.

Finally, recent research has pointed out that there can be benefits to bursty transmission. For example, [36] explains how burstiness can make TCP more robust to the reordering of packets caused by a route change.

3.6 Implementation with Netfilter

We implemented our techniques in a Linux kernel module. It is based on *Netfilter* [4], which is a generalized framework of hooks in the network stack of Linux 2.4. It is a superset of a firewall subsystem and is the basis for the implementation of IP tables and IP chains. Among other things, it supports packet filtering 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 [37] 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.

Our implementation is emulation based; in other words, while we implemented our client-centered approach in the kernel, we emulate the wireless card as well as the transitioning of states. Further, as will be seen in the next section, we emulate a wireless network using a wired one along with *DummyNet*. To explore the effects of using a wired network to emulate an actual wireless network and wireless client, we used ns-2. Specifically, we performed both wireless and wired simulations with our client-centered protocol and compared them. The results are shown in Table 1, which gives details of CC-wired and CC-wireless for several different RTTs. In this test, the wired bandwidth was set to 6 Mb/s because that was what we measured the effective wireless bandwidth as. As can be seen, the wireless results closely track the wired ones in execution time and energy consumption. (Further analysis showed the WNIC sleep time was the same, also.) Based on these simulations, we believe that our emulations using a wired network and client are valid.

TABLE 1
Slowdown and Energy Savings for Wireless and Wired Versions of CC at a Variety of RTTs

| RTT | Time (s) | | Energy (J) | |
|--------|----------|----------|------------|----------|
| | Wired | Wireless | Wired | Wireless |
| 30 ms | 6.1 | 5.8 | 5.5 | 5.3 |
| 45 ms | 6.9 | 6.6 | 5.8 | 5.6 |
| 60 ms | 7.9 | 7.5 | 6.0 | 5.9 |
| 90 ms | 9.7 | 9.4 | 6.5 | 6.4 |
| 120 ms | 11.5 | 11.3 | 7.1 | 6.9 |

3.7 Limitations

This section discusses our limitations. First, in this paper, our techniques only support the client-initiated request/response type of TCP streams. This type of traffic is two way and predictable. Although there are potentially many other kinds of traffic that can arrive at a wireless client without client soliciting, we believe this type of traffic is the largest wireless traffic.

Two-way predictable traffic can be handled as we have in this paper, by transitioning the WNIC between *sleep* and *high-power* mode when necessary. DNS requests would fall into this category, though, because of the likely small round-trip time, the WNIC should probably remain in high-power mode exclusively until the reply. For two-way, unpredictable traffic, the WNIC can be left in high-power mode until the reply is received. Our approach cannot be used for one-way traffic. For example, ARP traffic as well as voice-over-IP are one way and unpredictable. 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). Clearly, though, a client 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.

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.

4 PERFORMANCE

In this section, we show the results of our techniques. In Sections 4.1, 4.2, and 4.3, we describe our experimental setup, PSM/BSD implementation, and terminology, respectively. Next, in Section 4.4, we discuss the friendliness and fairness of CC, TCP, and PSM. In Section 4.5, we give our experimental methodology and performance results for Web browsing. In Section 4.6, we present performance results for large file downloads.

4.1 Experimental Setup

Our experimental setup is shown in Fig. 5. The wireless client was emulated by a 1GHz Pentium desktop machine running Linux 2.4-18. 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 provide a 20Mb/s bandwidth between access point and

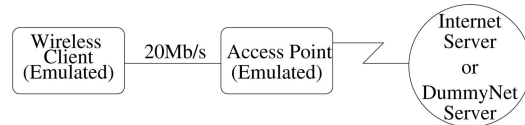


Fig. 5. Our experimental setup.

client. This value was selected by experimenting with 54 Mb/s access points with an actual wireless client and measuring the peak bandwidth attained. In practice, most wireless TCP flows do not achieve 20Mb/s because of the way that default socket buffer size is chosen in most systems (e.g., for a 64KB TCP socket buffer at a 40 ms RTT, the maximum speed is below 20 Mb/s). Hence, the wireless link is not the bottleneck, which allows energy saving.

We use a 100 Mb/s connection from the access point to the Internet. We compute energy based on a model of a 2.4 Ghz WaveLAN DSSS WNIC, which uses 1,319 mJ/s when idle, 1,425 mJ/s when receiving, 1,675 mJ/s when transmitting, and 177 mJ/s when in sleep mode [38]. We model the energy cost of transitioning the WNIC from *sleep* to *idle* mode as 2 ms in *idle* time [3].

We determine energy consumption by capturing a trace during execution via `tcpdump`. We evaluate this trace after the download has completed to compute energy savings and record the transmission time. A simulator reads the trace and computes energy based on the WaveLAN values. 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. This is compared to a 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*; this avoids 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 [3]. Note that we do not consider the effects on other system components, especially the CPU. Using our technique could conceivably cause extra energy to be consumed by the CPU because of the increased transfer time (a principle mentioned in [39]). However, in principle, we could scale down the processor while awaiting each burst—even possibly providing additional improvement—which regular TCP cannot as easily do.

In all experiments, we performed each test at least three times and report the results from the test with the median transmission time. For each experiment, we ran both our algorithm and TCP; the latter uses the standard TCP implementation in the Linux kernel. The 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. We also compare to PSM and BSD, which are described next.

4.2 PSM and BSD Implementation

We implemented software versions of 802.11b power-saving mode (PSM) [2] as well as BSD [3]. 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 after the download has completed 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 [6]. This means that PSM is unlikely to perform in practice as well as it performs in our emulation.⁴

For BSD, the proxy keeps track of the slowdown parameter (which was either 10 percent, 20 percent, 50 percent, or 100 percent—this means the slowdown is bounded by no more than that percentage) so that it knows when the client is expecting data. Note that, during downloads, the BSD protocol [3] keeps the WNIC in high-power mode exclusively.⁵

4.3 Terminology

Here, we briefly discuss the terminology used in this section. In all experiments, unless otherwise noted, we report transmission time and energy normalized to those of the TCP. Furthermore, we run our tests during either what we denote *peak* times (between 12:00 pm and 5:00 pm EDT, which have significant RTT variations) and *off-peak* times (run between 10:00 pm and 6:00 am EDT). For BSD, we denote *BSD-num* as the BSD version that bounds the percentage slowdown by *num*.

4.4 Friendliness and Fairness

With any new protocol, one must be concerned with how it affects standard TCP streams. In this section, we investigate the effect of CC on three protocols: CC (itself), TCP, and PSM. We do this using ns-2. All experiments use a bandwidth of 20 Mb/s and a 60 *ms* RTT.

4.4.1 Fairness

The first simulation is intended to measure how CC competes with itself; i.e., is CC fair? The results are shown in Fig. 6, with one to 10 concurrent CC flows. For means of comparison, we also show the corresponding number of TCP flows. Note that all throughput values (i.e., for each flow) are normalized to the *average* throughput for that number of flows.

Two things are evident from the figure. First, at smaller number of flows (e.g., four or five), CC has a disparity in throughput between flows. Here, it is likely due to initial loss, caused by a burst in one or two flows. Flows not incurring loss will in practice have less competition for bandwidth, so

4. In fact, our experience with PSM over a range of access points is that it performs much more poorly than expected.

5. BSD-100, where a slowdown of a factor of 2 is allowed, can save energy if the RTT is larger than 100 *ms*. In our experiments, only *inria.fr* has an RTT larger than 100 *ms*. BSD-10, BSD-20, and BSD-50 could never save energy during downloads.

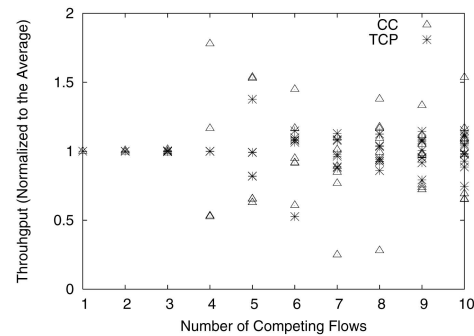


Fig. 6. Fairness of CC and TCP. Here, we show up to 10 concurrent flows of CC and 10 of TCP (separately). For each, we show the throughput of each flow, normalized to the average from that number of flows. Note that if points have a similar throughput, they may overlap.

there is a spread between CC flows—especially compared to TCP. Second, at nine and 10 flows, both TCP and CC flows look similar. This is because there are enough flows that losses happen more than once and most flows incur a similar number of losses. While CC is not as fair as TCP at a lower number of flows, we believe that the behavior at moderately large (nine to 10) flows is more important (because fairness is not as significant of an issue with few flows).

4.4.2 Friendliness

The next set of simulations is intended to determine whether or not CC is TCP friendly. We also run each protocol (CC, TCP, and PSM) against each of the other two (separately) to see how well each of the three protocols coexist with the other two. The results are shown in Fig. 7. With TCP and CC flows (Fig. 7a), the download times are relatively similar for a total of four, eight, and 16 flows (where half are TCP flows and half are CC flows). As expected, TCP is typically more aggressive than PSM and CC shows similar behavior when it competes against PSM.

Our measurements of flow throughput indicate that CC flows are friendly to TCP flows. We further investigated the effect of the protocols on the queue size—specifically, to ensure that CC does not adversely affect queue size. Fig. 8 shows four flows of each type on the left, and eight flows of each type on the right. In each figure, we look at the same combinations of flows as above. Importantly, it is clear from the figure that the queue fill levels are close to each other for all groups of flows.

4.5 Web Browsing

This section describes our Web browsing experiments and presents the results. It is important to note that we run actual experiments to real Internet servers.

4.5.1 Experimental Methodology

We carried out our experiments by running a script that retrieves 100 Web pages over a total browsing time of 5,400 seconds. These retrievals generated 558 requests for subobjects and a total of 2.79 MB was downloaded. The Web pages were chosen as follows: First, we selected the 20 most popular Web sites (denoted *top-level sites*) as determined by the *Alexa* Top Sites Web pages. Note that *Alexa* provides the *reach*, which is the percentage of Internet users that visit that site per day. For each top-level domain, the

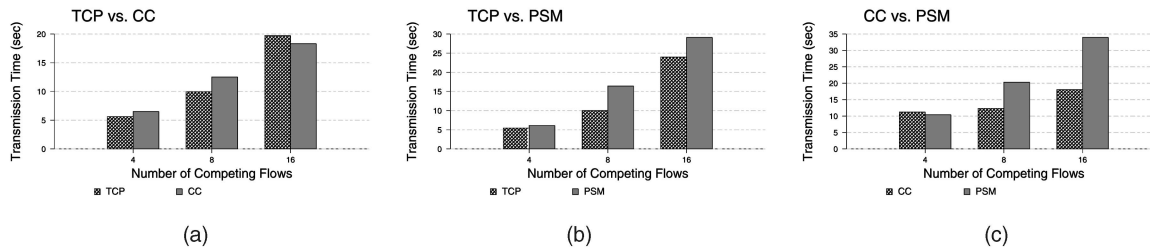


Fig. 7. Friendliness of all combinations of CC, TCP, and PSM for two, four, and eight flows (of each).

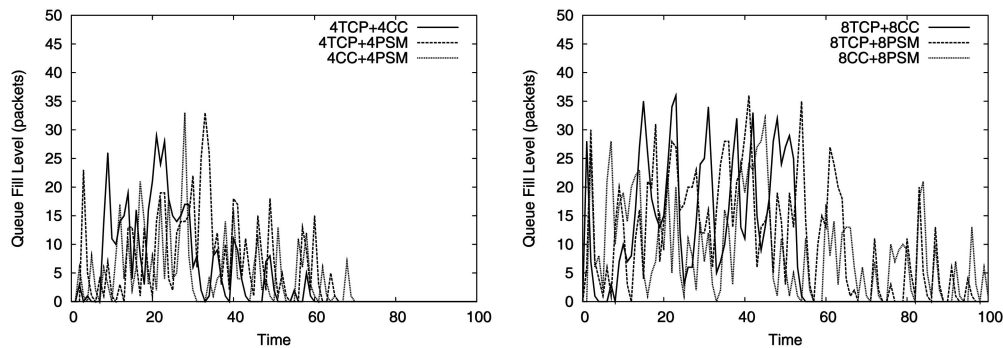


Fig. 8. Queue fill level for all three combinations of competing protocols (TCP versus PSM, TCP versus CC, and CC versus PSM). On the left are four of each flow and on the right eight of each flow.

Alexa list also provides the probability of visiting each of its subdomains, provided the user stays within that top-level domain. Note that *Alexa* does *not* provide the probability that the user stays within the top-level domain. For each site, we include all subsites that have a probability larger than 2 percent.

Our next step is to generate a sequence of Web page requests, which is done by using the Alexa probabilities and reach—First, we compute the probability of visiting a site given the reach; then, we use the conditional probability of each site in the domain if the next request is in that domain. For think times in between requests, we use the same method for determining think times as used in the BSD work [3] (a Pareto distribution with $\alpha = 1.5$ and $k = 1$).

4.5.2 Results

We ran our script on the Internet and divide our tests into peak and off-peak. Fig. 9 shows that CC is superior to PSM in both energy consumption and transmission time. It is also superior to BSD in energy consumption. BSD is restricted to waking up more often than CC (at least once every 0.9 seconds, to ensure its transmission bound). This means that CC can improve upon the best BSD variant in terms of think time behavior (BSD-100). The numbers presented for CC in Fig. 9 include the benefit of using the RTT cache; we measured this separately and found that its benefit was 10 percent overall.

4.6 Large File Downloads

This section describes our large file download experiments and presents the results. We begin with experimental methodology. Then, we present the overall experimental results on seven Internet servers, including a comparison to PSM. Finally, we give a detailed analysis of situations

where round-trip times vary as well as the effect of different access point bandwidths.

4.6.1 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*. We used *DummyNet* [5] to experiment with different wireless bandwidths, RTTs, and loss rates.⁶ Most of our tests used a 20Mb/s bandwidth between access point and client, while a few use other bandwidths.

Internet Experiments. The Internet experiments were carried out by running a script that downloaded a file at a time, in succession, from the seven Internet servers shown in Fig. 10. In each experiment, the client performs an *ftp* download from the server, requesting a file between 4MB and 5 MB. Experiments were performed during both peak and off-peak times.

DummyNet Experiments. In order to study our system in an environment where experiments are mostly repeatable, we used *DummyNet* to construct an emulated environment in which we could control loss rates and round-trip times. We model both peak and off-peak traffic patterns of the Internet. We observed that *DummyNet* is able to emulate well the lack of significant round-trip variation that occurs during off-peak traffic times. Hence, our off-peak simulations used the standard *DummyNet*, with a variety of round-trip times (30 ms, 60 ms, 90 ms, and 120 ms) and loss rates (several between 0 percent and 1 percent). However, peak traffic times show a much higher degree of variation in round-trip times. Unfortunately, *DummyNet* does not handle round-trip time variation; it uses only fixed delay

6. In our experiments, we do not differentiate between wireless and wired loss. We show experiments up to 1 percent loss.

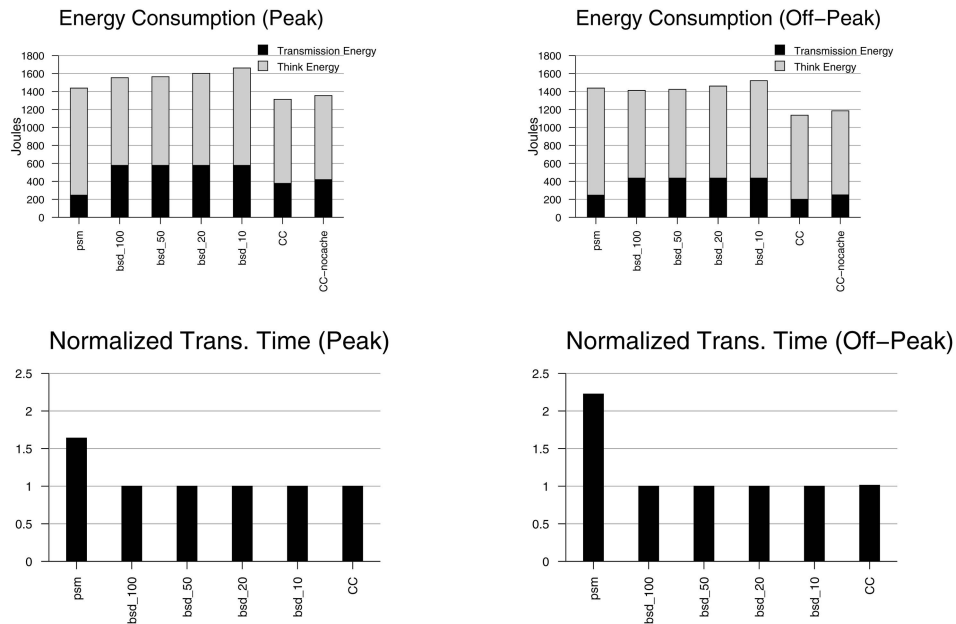


Fig. 9. Energy consumption and transmission time for both peak and off-peak real Internet tests, normalized to TCP. CC, PSM, and several BSD variants are shown. Smaller bars are better.

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 [17]. We model round-trip time variation using an uncorrelated gamma distribution [40]. 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 a correlated distribution and 2) an uncorrelated distribution is *more* difficult for CC to handle effectively than a correlated one due to the increased unpredictability of round-trip times. To emulate peak traffic on round-trip times other than 60 ms, we scaled this gamma distribution proportionally to the new round-trip time.

4.6.2 Results

In this section, we discuss the performance of CC on the Internet as well as in our emulated environment. CC uses active unless otherwise noted. All results are normalized to TCP. First, Fig. 11 shows energy consumed and transmission times for both peak and off-peak times when downloading

files from several Internet servers. Second, Fig. 12 shows the same metrics using emulated traffic with *DummyNet*.

In both environments, our system is effective at saving energy while maintaining reasonable download speeds. During peak times on the real Internet, our system on average uses 27 percent less energy with an increased normalized transfer time of 20 percent. During off-peak times, our energy savings is 32 percent with an increased normalized transfer time of 20 percent. In our emulated peak environment, our average energy savings (over all round-trip times and loss rates) is 51 percent, with an increased normalized transfer time of 12 percent. In the emulated off-peak experiments, energy savings is 47 percent and the normalized transfer time increases by 6 percent.

Internet experiments. The primary time overhead of CC is stream delay caused by *closed acks*. (The overhead D , described in Section 3, is typically much less than that of *closed acks*.) Therefore, the performance of CC is dependent on two factors: the round-trip time and the average window size. This is because the *closed ack* overhead decreases as 1) round-trip times increase and 2) average window size decreases.

This can be seen from Fig. 11, in which sites are shown ordered by increasing RTT. The time overhead does not strictly decrease as RTT increases because the average window size is not the same for all sites (see Fig. 10). A typical value for the window size is about 32 KB. The energy savings for all sites other than `cs.uiuc.edu` is at least 10 percent, and is often around or more than 50 percent.

There are a few sites for which the average window size varies significantly from 32 KB, which impacts performance considerably. One is with our best case, `jriver.net`, where the energy savings is more than 50 percent with less than 8 percent increase in transmission time. This is because this particular site has a small average window size (11 KB). On the other hand, both `cs.uiuc.edu` and

| Site | Base RTT (ms) | Average Window Size (KB) |
|--------------------------------|---------------|--------------------------|
| <code>cs.uiuc.edu</code> | 21 | 61 |
| <code>hp.com</code> | 58 | 31 |
| <code>cs.washington.edu</code> | 60 | 32 |
| <code>cs.stanford.edu</code> | 65 | 61 |
| <code>jriver.net</code> | 63 | 11 |
| <code>research.msft.com</code> | 84 | 17 |
| <code>inria.fr</code> | 114 | 61 |

Fig. 10. Sites used in large file Internet tests with base RTT (without variations) and average window size.

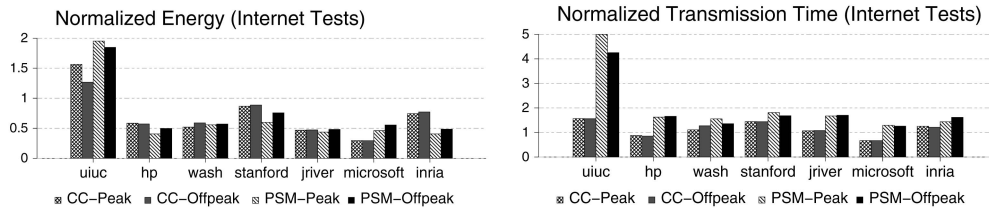


Fig. 11. Overall normalized energy savings and transfer time results for Internet tests using a 20 Mb/s access point and active end of burst detection during peak (2:00-5:00 PM EST) and off-peak (10:00 PM-1:00 AM EST) times. Results are shown for both CC and PSM.

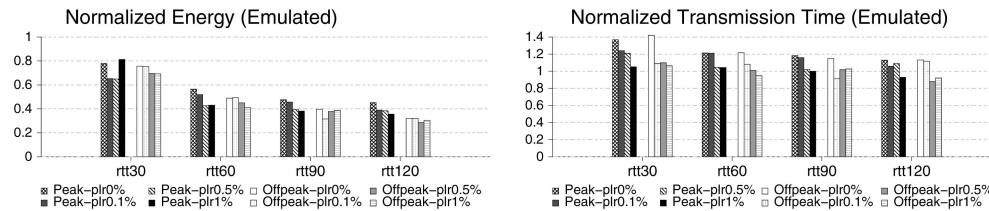


Fig. 12. Overall normalized energy savings and transfer time results for *DummyNet* tests using a 20 Mb/s access point and active end of burst detection during emulated peak and off-peak times. Here, only CC results are shown over different packet loss rates.

cs.stanford.edu have a window size of nearly the maximum (64 KB). Combined with the low RTT for cs.uiuc.edu, this causes CC to revert to TCP. The reason for the large overhead is 1) startup overhead when our client is trying to synchronize into energy-saving mode and 2) energy-saving 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. For cs.stanford.edu, CC was 44 percent slower than TCP, 6.7 seconds to 4.6 seconds. Analyzing the trace produced shows that the 64KB can be read by the client (which has a 20 Mb/s bandwidth) in about 26 ms. With an RTT of about 65 ms, we predict the stream delay as about 26/65, which is 40 percent. 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. To verify this, we ran a test to cs.stanford.edu with 40 Mb/s, and the slowdown was 16 percent compared to a predicted slowdown of 20 percent.

The time overhead of PSM, unlike CC, is solely dependent on the round-trip time. Essentially, PSM rounds the RTT to the nearest multiple of 100 ms (see Section 2). Therefore, PSM is slower than TCP by an average of 100 percent, which ranges from 511 percent for the lowest RTT (cs.uiuc.edu) to 29 percent for the largest RTT less than 100 ms (research.microsoft.com, at 84 ms); see Fig. 11. For inria.fr, which has a 120 ms RTT, the effective RTT becomes 200 ms, which adds 61 percent overhead.

Fig. 11 also shows that CC has lower time overhead than PSM whenever the RTT is less than 75 ms or greater than 100 ms. For a 20 Mb/s access point, the crossover point between CC and PSM is at approximately 75 ms if the average window size is 64KB (the maximum). This is because the stream delay due to closed acks will be about 25 ms, and PSM also increases the RTT 25 ms. (Under ideal conditions, CC would take slightly longer due to the delay, D , before the window probe packet.) Note that, as the average window size decreases, the crossover point will be a larger RTT.

Emulated experiments. We also experimented in an emulated environment so that we could precisely control round-trip time and loss rate. On the whole, the results are fairly similar to the Internet experiments. However, as shown in Fig. 12, the energy consumption much more clearly trends downward as both the RTT and loss rate increase. As described above, a large RTT means that the closed ack overhead is relatively small. Furthermore, the average window size decreases as loss increases, which also improves the closed ack overhead. Note also that, here, as we are using a single server, the advertised window size is similar for a given loss rate.

Fig. 13 compares CC to PSM and the Bounded Slowdown Protocol [3]. Comparing CC to PSM, we see that at low RTTs, CC is clearly superior to PSM in transmission time because PSM rounds effective RTTs up to 100 ms. 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 in both time and energy because the effective RTT increase is smaller.

At an RTT of 90 ms, PSM theoretically should perform better than CC (because the effective RTT increase is small). However, PSM introduces ack compression, which sometimes eventually results in TCP decreasing its congestion window—a reaction that is due to its belief that there is a problem on the network or at the receiver. This results in the sender returning to slow start. PSM increases the time without any ack appearing at the server (after a window of packets is sent by the sender) to at least 100 ms.

Note that BSD is identical to TCP. This is because we use BSD-50, which accepts a maximum slowdown of 50 percent. We chose BSD-50 because CC adds less than 50 percent overhead in transmission time. BSD-50 operates exactly as TCP at any RTT less than 200 ms, which was the case in all of our experiments.

Detailed analysis. This section discusses two aspects of our system in detail using *DummyNet*. First, we investigate the effects of RTT variation, both on the network as well as due to access point contention. Second, we investigate the effects of access point bandwidth.

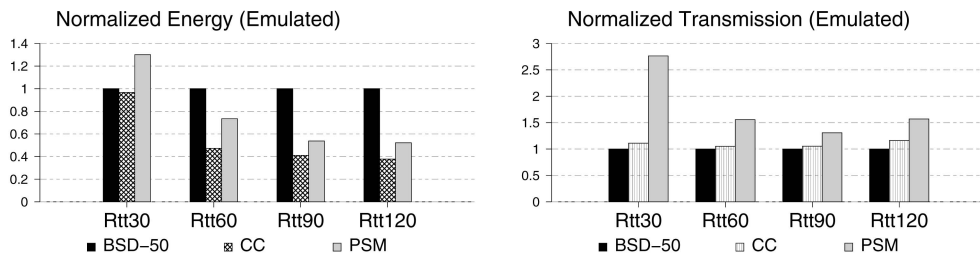


Fig. 13. Normalized energy savings and transfer time results for CC with active burst detection, PSM, and BSD-50 at varying RTTs. We emulated peak hours with a 0.1 percent loss rate and a 20 Mb/s access point.

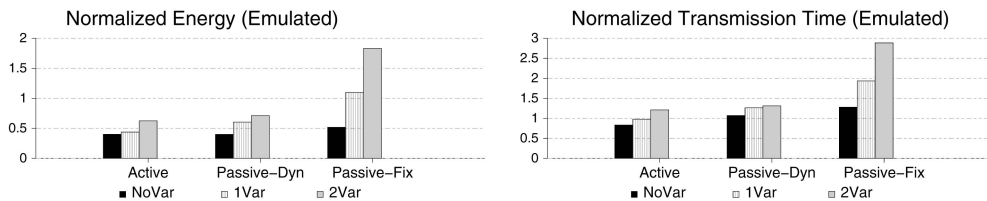


Fig. 14. Normalized energy savings and transfer time results ranging from no RTT variation to a maximum of 50 percent increase over the base RTT (called *2var*). We used a 20 Mb/s access point, a 60 ms RTT, and 0.1 percent loss.

We first discuss the effects of variations in round-trip times. This can be variation due to the network or variation due to other clients sharing the access point. The key to handling RTT variation lies in the end-of-burst detection algorithm, as the prediction of the next window is based on the minimum RTT and, so, is conservative (see Section 3).

Fig. 14 shows the effects of network-induced variation in RTTs. Specifically, we ran experiments with no variation, an uncorrelated gamma distribution where RTTs vary from 60 to 75 ms, and a distribution that was the same as the gamma distribution except the difference from the 60 ms base RTT is doubled (i.e., 60 to 90 ms). The last experiment is intended to see how our active burst detection performs under significant RTT variation.

While both active burst detection and the dynamic threshold handle RTT variation due to the network, active burst detection is superior in the case when there is RTT variation. This is because the window probe packet allows the client to transition the WNIC as soon as a burst has ended. When there is no variation, small overheads associated with active burst detection (e.g., energy incurred between the *open ack* and window probe packet), the dynamic threshold is slightly better.

We also experimented with multiple clients sharing an access point using CC with both active burst detection and a dynamic threshold. We also include a comparison to PSM. To do this, we first extended PSM to work with multiple clients. The PSM specification does not state anything about priority of packets between PSM clients and non-PSM clients. Clearly, it is not reasonable to give either type of client priority because that could lead to starvation of the other type. Therefore, we chose to use a weighted-fair-queueing algorithm at the access point when packets are being sent to both types of clients.

Fig. 15 shows the results of a single PSM or CC client along with a variable number of competing, regular TCP clients. Note that all results here are normalized to a *single*, regular TCP client; therefore, energy savings can occur even

if the y-value for a given bar is larger than one. This experiment has all clients download the same file at the same time. Saturation of the access point occurs at three competing clients. CC with active burst detection consumes less energy than PSM. This is because PSM must remain in high-power mode longer due to the competing flows, whereas with CC, the *sleep* time is constant (one RTT per burst).

Comparing CC with active burst detection to CC with a dynamic threshold indicates that active burst detection plays an important role in cases where multiple clients compete for access point bandwidth. The advantage of active burst detection is that it takes much less time than the dynamic threshold. Again, this is because active burst detection can precisely determine the end of each burst even with competing flows. Keep in mind that multiple flows can be viewed as single flows, each with RTT variation caused by the others. On the other hand, with the dynamic threshold, several packets are missed due to incorrect predictions. This also results, in turn, in less energy consumed with active burst detection. Finally, as expected, TCP is faster in transmission time but consumes more energy.

Finally, we measured the effects of different bandwidth access points: 4Mb/s, 10Mb/s, 20Mb/s, and 40 Mb/s. Fig. 16 shows the results of our experiments with these two bandwidths during peak and off peak hours. We observe that higher bandwidth access points will result in larger energy savings and smaller time overhead. The former is because we typically have a lower link utilization and, hence, more time to transition the WNIC to *sleep* mode. In particular, with a 4Mb/s bandwidth, it is much more likely that the connection is saturated, which means that 1) no energy can be saved and, so, 2) the client reverts to standard TCP. The latter is because the *closed ack* overhead is reduced as the bandwidth increases. It is important to realize that while CC gains a benefit in time *and* energy as bandwidth increases, PSM gains a benefit only in energy—it is a rigid scheme in which download time is independent of the bandwidth (assuming the connection is not saturated).

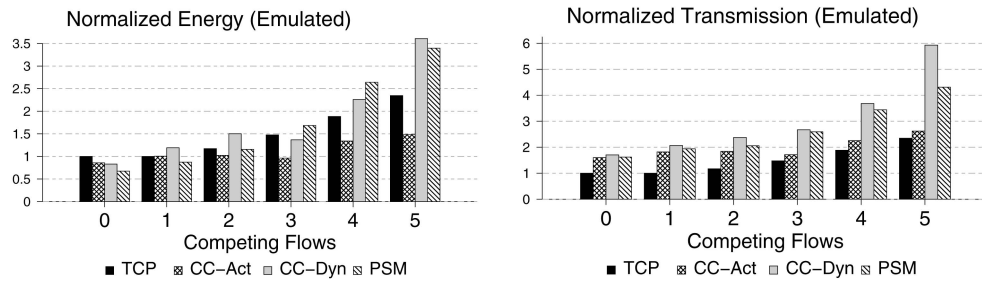


Fig. 15. Normalized energy savings and transfer time results for a single energy-aware client along with between 0 and 5 competing regular TCP clients. The results, which are shown for active burst detection, dynamic threshold, and PSM, are normalized to a *single* TCP client, so if there are n flows and the time (energy) is below n , there is a savings in time (energy) relative to using all TCP clients. We used a 20 Mb/s access point and a 60 m_s RTT.

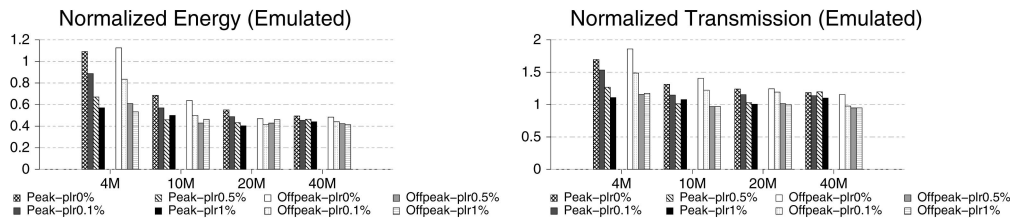


Fig. 16. Normalized energy savings and transfer time results for comparing different bandwidth access points (4 Mb/s, 10 Mb/s, 20 Mb/s, and 40 Mb/s) during peak and off-peak traffic. We emulated a 60 m_s round-trip time with various loss rates.

5 CONCLUSION

In this paper, we developed a novel method within the network layer to allow mobile clients to save energy for two popular mobile applications, Web browsing and TCP downloads. Our technique is *client-centered* and does not require changes to TCP or Web servers; furthermore, it requires no proxies or use of IEEE 802.11b power-saving mode.

Armed with the awareness of upper-layer applications and connection characteristics, our technique shows a better performance-energy trade-off than previous methods. For Web browsing, our results show that across 40 Web sites, the median energy savings is over 20 percent and the median increase in transmission time is less than 5 percent. In addition, our client-centered technique is better than PSM in terms of transmission time and better than BSD in terms of energy savings.

We also showed that, for large files, it is possible to increase WNIC energy savings if a client is willing to accept slightly longer transmission times. 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. Also, we use active to precisely determine the end of a burst. Results show that, compared to TCP, our scheme saves over 50 percent energy in the best case with a transmission increase under 8 percent and also does quite well on average over all our Internet sites, saving 27 percent energy and increasing the transmission by 22 percent.

Overall, we believe we have created a building block towards allowing multiple clients on a wireless network, some energy-conscious and some not, to selectively apply our client-centered techniques to save energy.

REFERENCES

- [1] J. Flinn and M. Satyanarayanan, "Energy-Aware Adaptation for Mobile Applications," *Proc. Symp. Operating Systems Principles*, pp. 48-63, Dec. 1999.
- [2] IEEE Computer Society LAN/MAN Standards Committee, "IEEE Std 802.11: Wireless LAN Medium Access Control and Physical Layer Specification," technical report, Aug. 1999.
- [3] R. Krashinsky and H. Balakrishnan, "Minimizing Energy for Wireless Web Access with Bounded Slowdown," *Proc. Int'l Conf. Mobile Computing and Networking*, Sept. 2002.
- [4] Netfilter, <http://www.netfilter.org>, 2006.
- [5] L. Rizzo, "Dummysnet: A Simple Approach to the Evaluation of Network Protocols," *ACM Computer Comm. Rev.*, vol. 27, no. 1, Jan. 1997.
- [6] S. Chandra and A. Vahdat, "Application-Specific Network Management for Energy-Aware Streaming of Popular Multimedia Formats," *Proc. USENIX Ann. Technical Conf.*, 2002.
- [7] E. Shih, P. Bahl, and M. Sinclair, "Wake On Wireless: An Event Driven Energy Saving Strategy for Battery Operated Devices," *Proc. Int'l Conf. Mobile Computing and Networking*, Sept. 2002.
- [8] S. Singh, M. Woo, and C.S. Raghavendra, "Power-Aware Routing in Mobile Ad Hoc Networks," *Proc. Int'l Conf. Mobile Computing and Networking*, pp. 181-190, 1998.
- [9] R. Kravets, K. Schwan, and K. Calvert, "Power-Aware Communication for Mobile Computers," *Proc. Int'l Workshop Mobile Multimedia Comm.*, Nov. 1999.
- [10] P. Shenoy and P. Radkov, "Proxy-Assisted Power-Friendly Streaming to Mobile Devices," *Proc. Multimedia Computing and Networking Conf.*, Jan. 2003.
- [11] T. Pering, T. Burd, and R. Brodersen, "The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms," *Proc. Int'l Symp. Low Power Electronics and Design*, Aug. 1998.
- [12] M. Weiser, B. Welch, A.J. Demers, and S. Shenker, "Scheduling for Reduced CPU Energy," *Operating Systems Design and Implementation*, pp. 13-23, 1994.
- [13] A.E. Papathanasiou and M.L. Scott, "Energy Efficient Prefetching and Caching," *Proc. USENIX Ann. Technical Conf.*, June 2004.
- [14] C. Gniady, Y.C. Hu, and Y.-H. Lu, "Program Counter Based Techniques for Dynamic Power Management," *Proc. Int'l Symp. High-Performance Computer Architecture*, Feb. 2004.
- [15] H. Zeng, C.S. Ellis, A.R. Lebeck, and A. Vahdat, "ECOSystem: Managing Energy as a First Class Operating System Resource," *Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

- [16] I. Batsiolas and I. Nikolaidis, "Selective Idling: Experiments in Transport Layer Energy Conservation," *J. Supercomputing*, vol. 20, no. 2, pp. 101-114, Sept. 2001.
- [17] M.C. Chan and R. Ramjee, "TCP/IP Performance over 3G Wireless Links with Rate and Delay Variation," *Proc. Int'l Conf. Mobile Computing and Networking*, Sept. 2002.
- [18] H. Balakrishnan, V.N. Padmanabhan, and R.H. Katz, "The Effects of Asymmetry on TCP Performance," *Proc. Int'l Conf. Mobile Computing and Networking*, Sept. 1997.
- [19] H.-Y. Hsieh, K.-H. Kim, Y. Zhu, and R. Sivakumar, "A Receiver-Centric Transport Protocol for Mobile Hosts with Heterogeneous Wireless Interfaces," *Proc. Int'l Conf. Mobile Computing and Networking*, Sept. 2003.
- [20] V. Tsaoussidis and C. Zhang, "TCP-Real: Receiver-Oriented Congestion Control," *Computer Networks*, vol. 40, no. 4, pp. 477-497, 2002.
- [21] H. Singh and S. Singh, "Energy Consumption of TCP Reno, Newreno, and SACK in Multi-Hop Wireless Networks," *ACM SIGMETRICS*, pp. 206-216, June 2002.
- [22] B. Wang and S. Singh, "Computational Energy Cost of TCP," *Proc. IEEE INFOCOM*, pp. 206-216, Mar. 2004.
- [23] L.M. Feeney and M. Nilsson, "Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment," *Proc. IEEE INFOCOM*, pp. 1548-1557, Apr. 2001.
- [24] J. Ebert, B. Stremmel, S. Eckhardt, and W. Adam, "An Energy Efficient Power Control Approach for WLANs," *J. Comm. and Networks*, vol. 2, no. 3, pp. 197-206, Sept. 2000.
- [25] H. Yan, R. Krishnan, S.A. Watterson, and D.K. Lowenthal, "Client-Centered Energy Savings for Concurrent HTTP Connections," *Proc. Workshop Network and Operating System Support for Digital Audio and Video*, June 2004.
- [26] H. Yan, R. Krishnan, S.A. Watterson, D.K. Lowenthal, K. Li, and L.L. Peterson, "Client-Centered Energy and Delay Analysis for TCP Downloads," *Proc. IEEE Int'l Workshop Quality of Service*, June 2004.
- [27] V. Jacobson, R. Braden, and D. Borman, "RFC 1323: TCP Extensions for High Performance," May 1992.
- [28] G. Lu and X. Li, "On the Correspondency between TCP Acknowledgment Packet and Data Packet," *Proc. ACM Internet Measurement Conf.*, Oct. 2003.
- [29] R. Wendland, "How Prevalent is Timestamp Options and Paws," *Web Survey Result, End-to-End Interest List*, 2003.
- [30] L. Le, J. Aikat, K. Jeffay, and F. Donelson Smith, "The Effects of Active Queue Management on Web Performance," *Proc. Conf. Applications, Technologies, Architectures, and Protocols for Computer Comm.*, pp. 265-276, 2003.
- [31] M. Crovella and A. Bestavros, "Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes," *Proc. Int'l Conf. Measurement and Modeling of Computer Systems*, May 1996.
- [32] V. Almeida, A. Bestavros, M. Crovella, and A. Oliveira, "Characterizing Reference Locality in the WWW," *Proc. Parallel and Distributed Intelligent Systems*, Dec. 1996.
- [33] Y.-C. Cheng, U. Hlzle, N. Cardwell, S. Savage, and G.M. Voelker, "Monkey See, Monkey Do: A Tool for TCP Tracing and Replaying," *Proc. USENIX Ann. Technical Conf.*, June 2004.
- [34] J. Hao and C. Dovrolis, "Source-Level IP Packet Bursts: Causes and Effects," *Proc. ACM Internet Measurement Conf.*, Oct. 2003.
- [35] Network Simulator-2, www.isi.edu/nsnam/ns/, 2006.
- [36] S. Sinha, S. Kandula, and D. Katabi, "Harnessing TCP's Burstiness with Flowlet Switching," *Proc. Workshop Hot Topics in Networks (HotNets-III)*, Nov. 2004.
- [37] Kansas University Real-Time Linux, <http://www.ittc.ku.edu/kurt/>, 2006.
- [38] P.J.M. Havinga, "Mobile Multimedia Systems," PhD thesis, Univ. of Twente, Feb. 2000.
- [39] A. Miyoshi, C. Lefurgy, E. Van Hensbergen, R. Rajamony, and R. Rajkumar, "Critical Power Slope: Understanding the Runtime Effects of Frequency Scaling," *Proc. Int'l Conf. Supercomputing*, 2002.
- [40] A. Mukherjee, "On the Dynamics and Significance of Low Frequency Components of Internet Load," *Internetworking: Research and Experience*, vol. 5, no. 4, pp. 163-205, Dec. 1994.



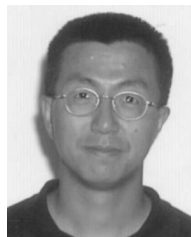
Haijin Yan received the PhD degree in computer science from the University of Georgia in January 2005. He received the Masters degree from the University of Electronic Science and Technology of China in 1998. His research interests are in computer networks, mobile computing, and distributed systems. He is now with Motorola for next generation mobile applications.



Scott A. Watterson received two BS degrees, one in computer science and one in mathematics, from the University of New Hampshire in 1994. He attended the University of Arizona computer science graduate program and was an assistant professor at the University of Georgia until 2003. He currently works in the Research and Development Group at Scientific Games Corporation.



David K. Lowenthal received the PhD degree in computer science at the University of Arizona in 1996. He is an associate professor of computer science at the University of Georgia. His research centers on parallel and distributed computing, operating systems, and networks. Current research projects include addressing scalability and energy for high-performance computing, as well as developing an infrastructure for flexible TCP-compatible protocols.



Kang Li received the BS degree in computer science from Tsinghua University, Beijing, China, in 1995 and the PhD degree in computer science and engineering from the Oregon Graduate Institute of Science and Technology in November 2002. He is currently an assistant professor at the University of Georgia.



Rupa Krishnan received the MS degree in computer science from the University of Georgia in 2004. She is currently a PhD student at SUNY-Stonybrook. Her research interests are in the field of wireless networks and systems.



Larry L. Peterson received the PhD degree from Purdue University in 1985. Currently, he is a professor and the chair of computer science at Princeton University, as well as the director of the Princeton-hosted PlanetLab Consortium. His research focuses on the design and implementation of networked systems. He is a coauthor of the networking textbook, *Computer Networks: A Systems Approach*. Professor Peterson recently served as editor-in-chief of the *ACM Transactions on Computer Systems*, he has been on the editorial board for the *IEEE/ACM Transactions on Networking* and the *IEEE Journal on Selected Areas in Communication*, and he has served as a program chair for SOSP and HotNets. He is a fellow of the ACM.