Client-Centered Energy Savings for Concurrent HTTP Connections

Haijin Yan, Rupa Krishnan, Scott A. Watterson, David K. Lowenthal
Department of Computer Science
The University of Georgia

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 WNIC is to transition it to a lower-power sleep mode when data is not being received or transmitted.**

This paper investigates client-centered techniques for saving energy during web browsing. The basic idea is that the client predicts when packets will arrive, keeping the WNIC in high-power mode only when necessary. This is challenging because web browsing generally results in concurrent HTTP connections. To handle this, we maintain the state of each open connection on the client and then transition the WNIC to *sleep* mode when no connection is receiving data. Our technique is compatible with standard TCP and does not rely on any assistance from the server, a proxy, or IEEE 802.11b power-saving mode (PSM). Our technique combines the performance of regular TCP with nearly all the energy-saving of PSM during web downloads, and we save more energy than PSM during client think times. Results show that over an entire web browsing session (downloads and think times), our scheme saves up to 21% energy compared to PSM and incurs less than a 1% increase in transmission time compared to regular TCP.

I. Introduction

Reducing energy consumption on mobile devices is becoming increasingly important. The wireless network interface card (WNIC) is a significant source of consumed energy in mobile devices. One way 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.

One common use of mobile devices is browsing the Internet [1]. Typically, this involves periods of downloading relatively short files followed by user inactivity. Mechanisms like 802.11b power-saving mode (PSM) [2] can be used force data to be buffered at the access point and arrive in bursts, which allows for an increase in WNIC *sleep* mode time at the client. However, PSM buffering increases round-trip times when data is arriving. According to [3], latencies when using PSM increase round-trip times from 16% to 232%.

This paper describes the design and implementation of a client-centered technique for saving energy during web browsing, where the browsing results in multiple concurrent HTTP (and therefore TCP) connections. For example, browsing `msn.com` results in five concurrent connections for embedded images and popups. Note that in this paper, we use the terms HTTP connection and TCP connection interchangeably.

Our technique is implemented with no change to TCP and no assistance from the web server or 802.11b power-saving mode (PSM) [2]. Our basic strategy is to maintain the state of each client-initiated connection. When *all* connections are idle, i.e., not actively receiving or sending data, the client transitions the WNIC to a lower-power *sleep* mode. The client then transitions the WNIC to back to high-power mode before the next packet (on any connection) arrives.

To carry out the algorithm described above, there are two key challenges that must be met. First, 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 handle this by tracking each connection and use smoothed round-trip time and variance estimates to accurately predict packet arrival times. Second, as an optimization, we want the client to be able to transition the WNIC to *sleep* mode during connection setup. This is because during web browsing, connections are often concurrent. This means that there is a larger percentage of time when at least one connection is in setup. To handle this, we exploit the principle of locality, caching detailed information including connection setup characteristics about each web site the client has visited within a single top-level web request.

To test our system, we ran actual tests to real Internet servers. In all comparisons we compared the results of our system with both PSM and a related approach known as the Bounded Slowdown Protocol (BSD) [3]. Our technique combines the performance of regular TCP and BSD with nearly all the energy-saving of PSM during web downloads, 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% energy compared to PSM and incurs less than a 1% increase in transmission time compared to regular TCP. In addition, our technique saves up to 25% more energy than BSD.

The rest of the paper is organized as follows. Section II discusses related work. Section III describes the design and implementation of our client-centered technique, and Section IV describes our experiments and discusses the results. Finally, Section V summarizes and discusses possible future directions for this research.

II. Related Work

This section discusses related work. Due to space limitations, not all related research can be included; please see our accompanying technical report for full details [4].

One way to save energy is to use the energy-saving mechanisms defined by 802.11b (power-saving mode, or PSM) [2]. Using PSM increases round-trip times to the nearest multiple of 100 *ms*, causing a significant delay. One improvement to 802.11b is the Bounded Slowdown Protocol (BSD) [3]. BSD maintains the WNIC in high-power mode during almost all of a TCP transmission (see Section IV), while bounding the transmission slowdown based on a user-supplied parameter. It is aimed at situations where there are long periods of user inactivity. Our work, while providing no bound on the potential slowdown, saves energy during transmission while generally incurring little or no delay.

In previous work we studied the problem of conserving energy on large file downloads [5]. That work is different than this current paper in that we consider only a single (ftp) connection, and we focus there on converting a smooth stream to a bursty one. In other words, we change the client TCP implementation to cause bursts. This paper focuses on concurrent HTTP connections for small file downloads and does not in any way modify TCP.

There has been also been significant research in power-aware computing at the network, hardware, and operating system levels, including dynamic voltage scaling [6], disk spindown [7], memory bank power-down [8], and power-aware 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, without modifying communication protocols.

Finally, our connection table maintains information across all TCP connections. This is reminiscent of TCP multiplexing [10].

III. IMPLEMENTATION

In this section we describe our detailed algorithm and implementation. We assume that when the client wishes to save energy for web downloads, it informs the client OS of its intentions. The client OS then must restrict the kinds of network traffic that can be sent and received. This is further discussed in Section III-D; for the rest of this section, we consider only HTTP traffic.

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 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 we inform the client OS to transition the WNIC between modes based on our predictions.

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. 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 track every 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. This section first discusses each of these techniques in more detail. We then discuss our implementation within Netfilter [11], a Linux kernel module. Finally, we discuss the limitations of our approach.

A. Connection Tracking

The client tracks each of its initiated open connections. 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; each stage is the client's estimate of the server's TCP window. Stages are easier to distinguish during TCP slow start. Fortunately, web objects are generally small, so many HTTP connections spend a significant percentage of time in slow start. The client builds a connection table to hold detailed information about each connection, which allows accurate prediction of the next packet arrival time. We first discuss tracking a single connection and then discuss combining information from many connections to determine when to transition the WNIC.

A.1 Connection table

The key data structure used in our system is the *connection table* (shown in Figure 1). The connection *status* field reflects four possible connection 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 transmission stage. If the connection is *active*, we also keep track of when the current stage is expected to end (*current expiration*). We use a timer to detect when there are no packets received within a threshold amount of time (described below); this is a possible indicator of the end of a stage.

In the example connection table shown in Figure 1, there are 4 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*.

A.2 RTT estimation

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 [12]. 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 used in most TCP implementations [13].

By default, the TCP receiver does not measure *RTT* for pure acknowledgements. Nevertheless, 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.

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

Fig. 1. Sample connection table with 4 concurrent connections (two *idle*, one *finished*, and one *active*). *TBD* means that the value has not yet been determined.

Currently we assume that the timestamp option is enabled. In all the servers in our experiments, this was the case. If it were not, we believe that by using techniques such as those presented in [12] to associate packets with their acknowledgements, we could obtain accurate *RTT* estimates. We leave this for future work.

A.3 Next stage prediction

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.

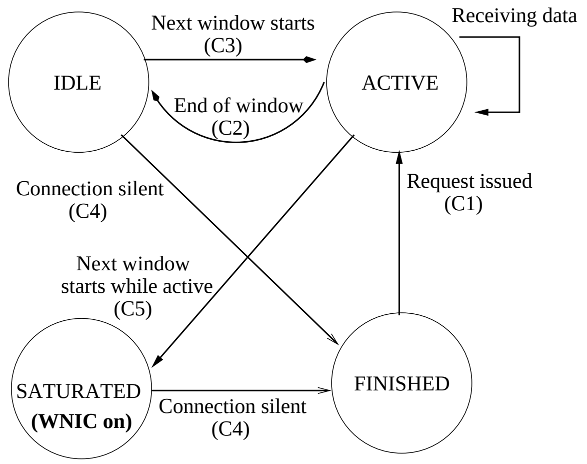
We record the interval between the first acknowledgement the client sent out in stage $i-1$ and the first packet the client received in stage i as our new round-trip measurement (*RTT*). The new variance measurement is then the difference between the measured *RTT* and the *SRTT*. To calculate the *SRTT* 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.)

A.4 Transitioning between states

Figure 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*.

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% of the time for the web sites in our experiments. If it is not, a connection is never 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*.



C1: A new request on a stagnant connections returns the state to active

C2: End of the window can be detected by count (slow start) or timer (congestion avoidance)

C3: Next window requires a transition to active state

C4: When no data arrives on a connection for a threshold, it is declared stagnant

C5: If the next window starts before the current one ends, there is no "dead time" and the WNIC should remain on

Fig. 2. State machine demonstrating our algorithm.

A.5 Extending to multiple connections

The technique described above identifies periods when no data is expected on a single connection. Web browsers like Internet Explorer and Netscape 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.

B. 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 percentage in which *at least* one connection is in ei-

Site	*Syn-RTT*	*Get-RTT*	*SRTT*
nytimes.com	28	49	33
popupads.com	45	60	49
akamai.com	32	45	38

Fig. 3. Example Round-Trip Time Cache for a top-level request to `nytimes.com`.

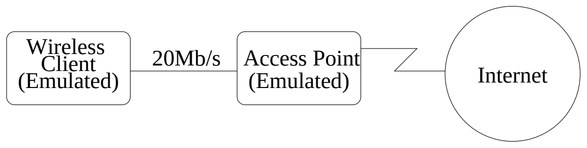


Fig. 4. Our experimental setup for the tests to actual Internet servers.

ther 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 non-scalable: for highly concurrent connections, little, if any, energy can be saved.

Fortunately, user web accesses exhibit high degrees of locality [14]. Therefore, we cache detailed round-trip time information for sites the client has visited within a "top-level" web request (see below), including SYN/SYN-ACK time, GET/GET-ACK time, and the last known *SRTT* (see Figure 3). 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

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 "top-level" web requests (i.e., requests in our trace file), we flush the RTT cache. Within one top-level web 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 IV).

C. Netfilter

We implemented our prediction-based algorithm in a Linux kernel module based on *Netfilter* [11]. 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 [15] 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.

D. Limitations

This section discusses two of the limitations of our system. First, in this paper we support HTTP traffic. This type of traffic is two way (request/reply) and predictable. 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. So, for example, a client cannot use our energy-saving techniques for web downloads while using an application such as voice-over-IP. In addition, web pages using *server push* cannot be used. Finally, we cannot respond to ARP requests. However, we believe our approach handles the most common cases.

Second, our technique successfully saves energy for web sites with a moderate number of concurrent connections. If the number of concurrent connections is large enough, there are not sufficient gaps to save energy, so our algorithm will keep the WNIC in high-power mode for the entire download. However, in such a case, no scheme, including PSM or BSD, is capable of saving energy.

IV. Performance

This section describes our experiments and presents our results. Section IV-A describes the experimental methodology. Then, Section IV-B gives our results on real Internet experiments. This includes a comparison with 802.11b power-saving mode (PSM) [2] and BSD [3], as well as a detailed analysis of some of the aspects of our system. It is important to note that we run actual experiments to real Internet servers. Due to space limitations, we report on a limited number of experiments; full results are reported in an accompanying technical report [4]. Note that those results also provide simulation results with an identical trace as was used by [3].

A. Experimental Methodology

This section describes our experimental methodology. In turn, we discuss how we chose the web sites, how we implemented the competing techniques (PSM and BSD), and how we set up the experiments.

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 [16]. Note that the *reach* is provided by *alexa*, which is the percentage of Internet users that visit that site per day. For each top-level domain, the *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 sub-sites that have a probability larger than 2%.

Our next step is to generate a sequence of web page requests, which is done by using the Alexa probabilities and reach—first

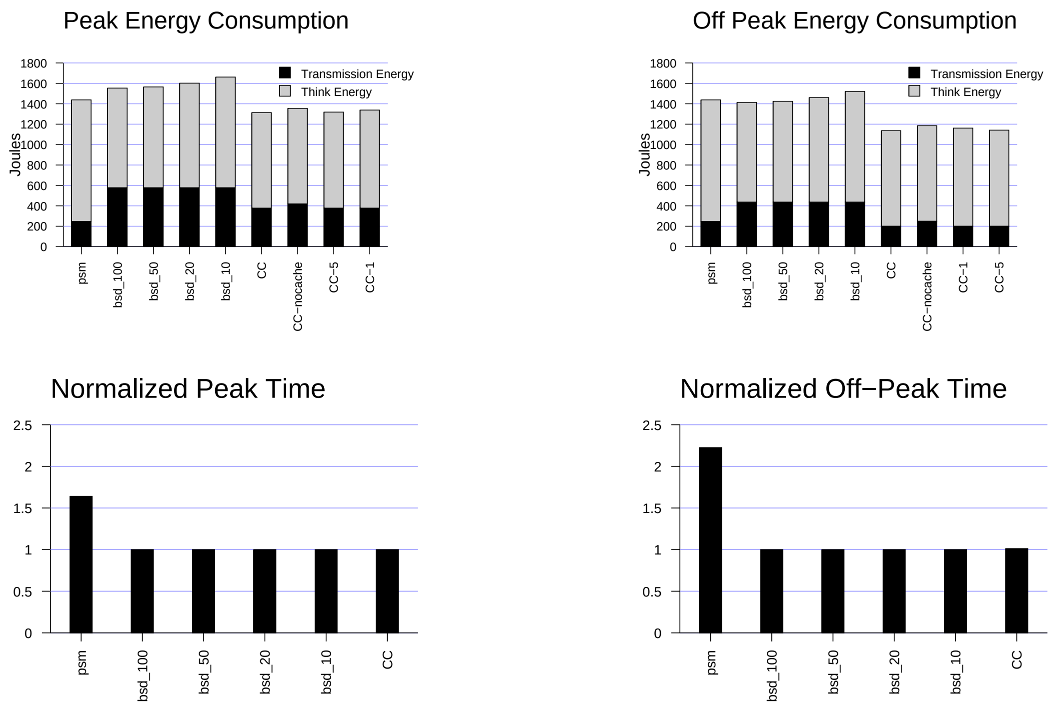


Fig. 5. Energy consumption and normalized transmission time for both peak and off-peak real Internet tests. CC, PSM, and several BSD variants are shown. Smaller bars are better.

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 $k = 1$). The exact set of experiments, along with further details, can be found on line at `http://www.cs.uga.edu/~yan`.

We compare our client-centered technique (CC) with PSM and BSD [3]. Our emulation of PSM and BSD for the actual Internet tests uses a transparent proxy that intercepts packets before they reach the access point and emulates the access point behavior. For PSM, the proxy buffers packets and sends them (after the beacon packet, which indicates if any data is buffered) to the client every 100 *ms*, our chosen beacon period. Note that our implementation of PSM is essentially optimal; it is unlikely to perform in practice as well as it performs in our emulation (see [4] for more details). For BSD, the proxy keeps track of the slowdown parameter (which was either 10%, 20%, 50%, or 100%—this means the slowdown is bounded by no more than that percentage) so that it knows when the client is expecting data.

Our experimental setup is shown in Figure 4 (previous page). 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* [17] could be used to provide a 20Mb/s bandwidth between access point and client. This value was selected by experimenting with 54 Mb/s access points with an actual wireless client and measuring the peak bandwidth attained. We use a 100Mb/s connection from the access point to the Internet. In all experiments, we performed each test three times and report the results from the test with the median transfer time.

As described above, we ran our script in both ns-2 and on the Internet. In the former, we use standard ns-2. In the latter, we divide our tests into *peak* tests (run between 12pm and 5pm EDT, which have significant RTT variations) and *off-peak* tests (run between 10pm and 6am EDT). In both, we calculate during execution the amount of time the WNIC is in each of its modes (*idle*, *sleep*, *transmit*, *receive*). Then, we compute energy based on a model of 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 [18]. Also, we model the energy cost of transitioning the WNIC from *sleep* to *idle* mode as 2 *ms* in *idle* time [3].

B. Results

Figure 5 shows that CC is superior to PSM in both energy consumption and transmission time. It is also superior to BSD in energy consumption. The figure also shows several CC variants, each with different behavior during think times. BSD is restricted to waking up more often than the CC variants (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). Figure 5 also shows the benefit of using the RTT cache, which is discussed further below.

This section gives an in-depth analysis of several aspects of our system. We first break down the different components of energy. Second, we show the improvement gained from using the RTT cache. Finally, we investigate the effect of different RTTs on energy consumption.

Figure 6 quantifies the sources of energy consumed in our experiments. Energy can be divided into five categories: *early*

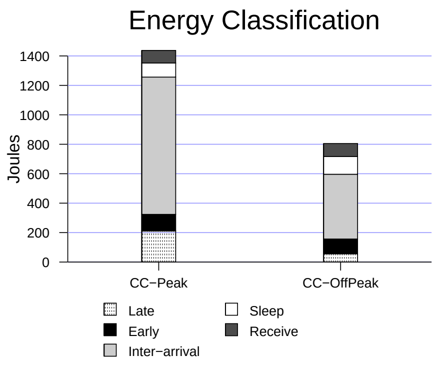


Fig. 6. Breakdown of energy consumed for CC, in joules, for both peak and off-peak browsing sessions.

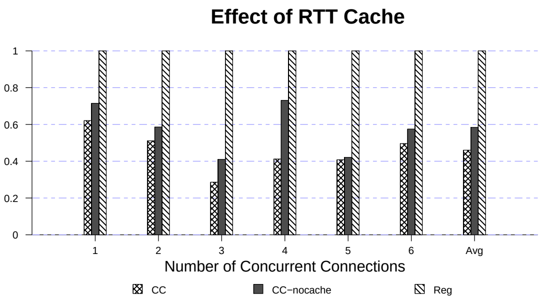


Fig. 7. Benefi t of RTT cache. All results are normalized to regular TCP, during downloads only.

wakeup, *late sleep*, *inter arrival*, *receive*, and *sleep*. Early wakeup energy accounts for energy consumed between the transition to *idle* mode and the receipt of the next packet. Late sleep energy accounts for the time spent between the transmission of the last acknowledgement and the transition to *sleep* mode. Inter-arrival energy accounts for the time spent in *idle* mode while actively receiving packets. Receive energy is the energy consumed when actually receiving data. Sleep energy is the energy consumed when the WNIC is in *sleep* mode. Note that any energy-saving technique will contain some of each of these categories (other than optimal, which has no *early wakeup* or *late sleep*).

Figure 7 shows that the RTT cache is effective in saving energy during downloads. In particular, the overall benefit (far right) is about 10% (compared to the regular TCP test). The rest of the figure breaks down the benefit of the RTT cache for each number of concurrent connections. In particular, the largest benefit of the RTT cache is at 3 and 4 concurrent connections, providing an improvement of 13% and 32%, respectively. With 1 and 2 concurrent connections, the benefit is limited because the RTT cache pays off only during connection setup. With 5 and 6, there is a high likelihood that connection setup overlaps with an active stage for a different connection.

V. Summary and Future Work

This paper has discussed a client-centered technique for saving energy during web downloads. Our algorithm does not require changes to TCP or web servers; furthermore, it requires no proxies or use of 802.11b power-saving mode. The client maintains the state of each initiated connection and transitions the WNIC to a lower-power *sleep* mode when all connections are idle. The client then transitions the WNIC to high-power mode before the next packet arrives. The key implementation mechanisms are (1) a connection table that tracks the state of each connection and (2) an RTT cache that allows the WNIC to be transitioned to *sleep* mode during connection setup.

Results show across 40 web sites, the median energy savings is over 20% and the median increase in transmission time is less than 5%. 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 feel that our energy-saving techniques will help prolong battery life on mobile devices without significantly affecting the user web browsing experience.

While our results are good in most cases, there is still further investigation required. In particular, some web sites will have jittery connections to the Internet or lossy routing paths, making predictions difficult. Though this did not occur in our experiments, it is possible that a significant number of packets could be missed on such connections due to mis-predicted WNIC transitions. This could cause transmission time to greatly increase and would also increase the amount of energy consumed. In addition, we intend to investigate handling multiple clients, which will have the effect of additional variance. Finally, we will look at issues arising from user roaming.

References

[1] David Kotz and Kobby Essien. Analysis of a campus-wide wireless network. In *Mobicom*, pages 107–118, September 2002.
[2] IEEE Computer Society LAN/MAN Standards Committee. IEEE Std 802.11: Wireless LAN medium access control and physical layer specifi cation. Technical report, August 1999.
[3] Ronny Krashinsky and Hari Balakrishnan. Minimizing energy for wireless web access with bounded slowdown. In *Mobicom*, September 2002.
[4] Haijin Yan, Rupa Krishnan, Scott A. Watterson, and David K. Lowenthal. Client-centered energy savings for concurrent HTTP connections. Technical report, University of Georgia.
[5] Haijin Yan, Rupa Krishnan, Scott A. Watterson, David K. Lowenthal, and Kang Li. Client-centered energy savings for TCP downloads. Technical report, University of Georgia.
[6] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *ISLPED 1998*, August 1998.
[7] 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.
[8] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In *ASPLOS*, pages 105–116, 2000.
[9] R. Kravets, K. Schwan, and K. Calvert. Power-aware communication for mobile computers. In *Proc. 6th International Workshop on Mobile Multimedia Communications*, Nov 1999.
[10] Jim Gettys and Henrik Frystyk Nielsen. The WebMUX protocol. Internet Engineering Task Force, 1998.
[11] Netfi lter. `http://www.netfilter.org`.
[12] Guohan Lu and Xing Li. On the correspondency between tcp acknowledgment packet and data packet. In *ACM Internet Measurement Conference 2003*, Oct 2003.
[13] Richard Wendland. "How prevalent is timestamp options and paws". Web survey result published in end-to-end interest list, 2003.
[14] Virgilio Almeida, A. Bestavros, M. Crovella, and A. Oliveira. Characterizing reference locality in the WWW. In *Proceedings of PDIS*, December 1996.
[15] Kansas Unversity Real-Time Linux. `http://www.ittc.ku.edu/kurt/`.
[16] Alexa Top Sites. `http://www.alexa.com/site/ds/top_500`.
[17] Luigi Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM Computer Communications Review*, 27(1), January 1997.
[18] Paul J. M. Havinga. *Mobile Multimedia Systems*. PhD thesis, Univ. of Twente, Feb 2000.