An Experimental Study of TCP's Energy Consumption over a Wireless Link

Sandeep Agrawal and Suresh Singh

Abstract—**In this paper we examine TCP's energy consumption behavior over wireless links and suggest improvements that result in significant energy savings. The various modifications and fine-tunings to TCP code suggested here help in conserving battery power at nodes by saving on software overhead and reducing protocol processing. These modifications were tested with experiments done on three laptops equipped with 11Mbps Lucent WaveLAN wireless cards. The results obtained from these experiments indicate that, with certain modifications made in the implementation of TCP code, we can attain as much as a 25% improvement in TCP's efficiency for the same amount of energy consumed. This work is ongoing and as a next step we will be examining the energy behavior of TCP in more complex wireless environments.**

I. Introduction

Nodes in wireless networks need to remain on battery power for extended periods of time and thus these nodes need to be energy conserving. For instance, significant amount of power is consumed by the display, by spinning disks, by the CPU, by I/O devices and also by the transceiver radio. Recently some researchers have begun studying the problem of reducing power consumption at the wireless interface. Some examples of this work include power-efficient MAC protocols [7], [2] and energy-efficient routing [8], [3]. Little work has been done to address the energy consumption issues at the transport layer, however. In this paper we examine, in detail, how TCP's energy consumption can be reduced while remaining within the *standards-imposed constraints*.

The results of the experiments reported in this paper document TCP's energy consumption behavior in wireless environments. In our experiments we use laptops equipped with 11Mbps 802.11 cards. For each experiment the laptops are charged to capacity after which data is sent over a TCP connection until the battery drains out. We measure the time as well as the amount of data transmitted for each modification made to TCP code. Some modifications do yield improvements in the energy-efficiency of TCP (e.g., larger MTU sizes are more efficient) while others make little difference (e.g., the window scale option). In this paper we report all of our attempts (successful as well as unsuccessful) as a guide to other researchers interested in working on the same problem. The remainder of the paper is organized as follows:

- Section II presents the specific modifications we made to TCP in an effort to improve its energy-efficiency.
- Section III presents experimental results along with a discussion of why some techniques work while others do not.

This research was funded by DARPA under contract F33615-C-00-1633 and the NSF under grant No. NCR-9706080

Department of Computer Science, Portland State University, Portland, OR 97201 ,*singh@cs.pdx.edu*

II. Approaches for Improving TCP's Energy Consumption Behavior

The current implementations of TCP in Linux incorporate several optimizations including reduced copy operations, header prediction and efficient checksum computation that do result in reduced energy consumption as compared with older TCP implementations. Unfortunately, however, along with these modifications current TCP implementations include certain options and extensions which are only useful for long fat networks (LFN) where the delay-bandwidth product is large. These options cause extra protocol processing both at the receiver and sender which makes them problematic for wireless networks. Wireless ad-hoc networks have a relatively small bandwidth (11Mbit/s for the new WaveLAN cards) and the propagation delay of the medium is low since the distance between the source and destination is generally not very large. The upshot of this is that the delay-bandwidth product for ad hoc networks is much smaller than for LFNs and thus many of the options implemented to support LFNs will not be suited to ad hoc networks. These various options, which are suitable only for LFNs and need to be modified for slow wireless networks, are discussed below. We describe how each of these options work and give reasons why they may need to be turned off in wireless networks.

A. Timestamp Option

TCP implements reliable data delivery by re-transmitting segments that are not acknowledged within some retransmission timeout (RTO) interval. Accurate dynamic determination of an appropriate RTO is therefore essential to TCP performance. RTO is determined by estimating the mean and variance of the measured round-trip time (RTT), i.e., the time interval between sending a segment and receiving an acknowledgment for it [5]. Many TCP implementations base their RTT measurements upon a sample of only one packet per window. While this yields an adequate approximation to the RTT for small windows (used in mobile ad-hoc systems), it results in an unacceptably poor RTT estimate for LFNs which have very large sized windows.

The timestamp option is one in which the sender uses 12 bytes of the TCP options field to place a timestamp in every segment sent (including retransmissions) to the receiver. The receiver echoes this timestamp value in the ACK packet sent to the sender. By using this timestamp option in every packet the sender is able to get a better value of the round trip time (RTT). Clearly, this option is very useful for high-speed networks because the RTO can be adjusted quickly resulting in better usage of the large bandwidth. In wireless networks, on the other hand, the overhead of obtaining frequent RTTs is not justified for at least two reasons – first, the window size of TCP tends to be small (small delay*bandwidth product) unlike LFNs and second, using this option results in additional computation and communication overhead corresponding to the additional 12 bytes of RTT. Thus by not using this option we would hope to gain a small amount of energy efficiency.

PAWS (Protection Against Wrapped Sequence Numbers) uses the same TCP Timestamps option as the RTT mechanism described above, and assumes that every received TCP segment (including data and ACK segments) contains a timestamp whose values are monotone non-decreasing in time. The basic idea is that a segment can be discarded as an old duplicate if it is received with a timestamp which has a value that is less than the last previous timestamp received on this connection. This is again a very useful option for high-speed networks because of the high bandwidth. However, this feature is not required for slower wireless networks since the wrap around of sequence numbers will not occur within the MSL (Maximum Segment Lifetime) due to the much lower transmission speeds of wireless links. It should be mentioned here that having the PAWS checking is not much of a load on the nodes by itself but still it would be better not to have it because PAWS checking is the first step in the *frequent fast path* of the TCP code (section II-D) and, as we will discuss, it is best to get rid of it for efficiency reasons.

B. Window Scale Option

The TCP header uses a 16-bit field to report the receive-window size to the sender. Therefore, the largest window that can be used is $2^{16} = 65K$ bytes. The window scale extension expands the definition of the TCP window to 32 bits and then uses a scale factor to carry this 32-bit value in the 16-bit Window field of the TCP header. The scale factor is carried in a new TCP option called Window Scale. This option is sent only in a SYN segment (a segment with the SYN bit on), hence the window scale is fixed in each direction when a connection is opened. Again it should be noted that very large windows are not necessary for slower wireless connections and hence this option is not required by wireless TCP. However it does have a purpose in high-speed connections as the window size is a limitation in those connections and larger window sizes might have to be negotiated in the SYN segment if allowed by both the receiver and the sender.

C. SACK Option

Any packet loss in a LFN can have a catastrophic effect on TCP's throughput because of two reasons – the sender only retransmits the lost packet when a timeout occurs resulting in an underutilization of the network resources, and second, because of TCP's use of cumulative ACKs, several packets following the lost packet get retransmitted unnecessarily. This happens because the sending TCP has no information about segments that may have reached the receiver and been queued since they were not at the left window edge. So it may be forced to retransmit these segments (since they will time-out soon after the lost packet times out - unless a new updated ACK is received from the receiver). A Selective Acknowledgment (SACK) mechanism, combined with a selective repeat retransmission policy, can help to overcome these problems for LFNs. Here, the receiving TCP sends back SACK packets to the sender informing the sender of data that has been received. The sender can then retransmit only the missing data segments.

Unlike LFNs, however, in slower wireless connections TCPs reliance on timeouts and use of cumulative ACKs will probably not have the same impact on performance. This is because not many segments after the lost segment would have been transmitted to the receiver anyway (smaller windows) and thus the number of needless retransmissions would be very small (if any). TCP may however experience poor performance when multiple packets are lost from one window of data. With the limited information available from cumulative acknowledgments, a TCP sender can only learn about a single lost packet per round trip time. An aggressive sender could choose to retransmit packets early (fast retransmit and fast recovery), but such retransmitted segments may have already been successfully received.

If SACK is implemented in wireless networks, there is a benefit of not retransmiting needlessly (thus conserving energy) but the protocol processing involved in implementing SACK might overcome this small benefit gained. There is thus a tradeoff between the amount of power required by the sender to retransmit the packets and the power required to do the SACK protocol processing. In the non-LFN regime therefore, while selective acknowledgements reduce the number of packets retransmitted (not a whole lot) they may not otherwise improve performance, making their complexity (and extra power consumed due to this complexity) of questionable value. Finally, it must be kept in mind that SACKs are only useful if the physical medium is highly error-prone and there is a general tendency of getting more than a single packet error in a particular window.

D. Header Prediction

Header prediction [4] is a high-performance transport protocol implementation technique that is most important for high-speed. This technique optimizes the code for the most common case, receiving a segment correctly and in order. Using header prediction, the receiver asks the question, "Is this segment the next in sequence?" This question can be answered in fewer machine instructions than the question, "Is this segment within the window?" Adding header prediction to the timestamp procedure leads to the following sequence for processing an arriving TCP segment - this is also the implementation in Linux:

H1) Check timestamp - this means check to see if the packet is not an older delayed packet by checking the timestamp value with the most recent timestamp value received earlier.

H2) Do header prediction - if the segment is next in sequence (checked by using the frequent path code which basically has about five comparisons) and if there are no special conditions requiring additional processing, accept the segment, record its timestamp, and skip H3.

H3) Process the segment normally - (this is the slow path which would only be taken if there are errors etc). This includes dropping segments that are outside the window and possibly sending acknowledgments, and queuing in-window, out-of-sequence segments.

In the above algorithm the modification that we can make for wireless networks would be to interchange steps H1 and H2, i.e., to perform the header prediction step H2 first, and perform H1 and H3 only when header prediction fails. This can be done because H2 basically checks for H1 also except for the case when the packet received is exactly the same packet (i.e.

next in sequence with the same headers) but from the previous window of 2^{32} bytes. This could be a performance improvement, since the timestamp check in step H1 is very unlikely to fail and timestamp checking requires interval arithmetic on a finite field, which is a relatively expensive operation. To perform this timestamp check on every single segment is contrary to the philosophy of header prediction and speeding up the frequent path. However, putting H2 first would create a hazard: a segment from 2^{32} bytes in the past might arrive at exactly the wrong time and be accepted mistakenly by the header-prediction step. The following reasoning has been introduced in [6] to show that the probability of this failure is negligible. If all segments are equally likely to show up as old duplicates, then the probability of an old duplicate exactly matching the left window edge is the maximum segment size (MSS) divided by the size of the sequence space. For IEEE802.11b, the maximum MSS size is 2296 bytes that gives us a probability of $2^{11}/2^{32} = 2^{-21}$. This probability of error is *smaller* than the unreliability of TCP's checksum (with a 16-bit checksum, this unreliability is 2^{-16})! Thus, a protocol mechanism whose reliability exceeds the reliability of the TCP checksum should be considered "good enough", i.e., it won't contribute significantly to the overall error rate. From the above reasoning it can be concluded that we can ignore the problem of an old duplicate being accepted by doing header prediction (step H2) before checking for the timestamp (step H1). Hence H1 can be done after H2 in the algorithm and this would obviously cause some saving in the power consumed without causing any effect on the reliability of TCP. The saving would come from the fact that most of the packets received would satisfy H2 (which would now be before H1) and would not have to go through H1 at all.

E. MTU Size

As we know, the Ethernet supports a maximum MTU (Message Transfer Unit) size of 1500 bytes. However, for wireless transmissions the MAC protocol used is the IEEE802.11b standard that allows a larger MTU size of 2296 bytes and it allows all protocols running above it to use this size as their MTU. Hence the IP layer above the 802.11 MAC protocol will not fragment datagrams that it receives from the transport layer which are within 2296 bytes. So the TCP layer can negotiate a MSS (maximum segment size) of 2296 bytes and send datagrams with a maximum size of 2296 bytes without the fear of having the IP layer fragment them and thus increase processing costs. This support for large MTU sizes enables us to select "correct" MTU size for a specific application. There are benefits as well as drawbacks to using large versus small MTU sizes. With large MTU sizes, it takes fewer packets to send a file and saves on energy because fewer header bits are used and fewer packets need to be processed. On the other hand using small MTU sizes has a lower probability of error in a transmitted packet implying a lower retransmission cost (retransmitting a large packet is more expensive than retransmitting a small packet).

F. Cumulative and Delayed ACK Implementation

In the current implementation of TCP in Linux, delayed ACKs are implemented based on the packet count as well as time. Thus, for every two full packets received, the receiver sends an ACK to the sender. This is done so as to provide instant feedback to the sender about the condition of the path – necessary in LFNs because of the high delay-bandwidth product of these networks. For mobile systems, on the other hand, delayed ACK implementation based on packet count is not necessary - specifically the receiver need not send an ACK for every two packets received from the sender. The more useful choice here is to implement delayed ACKs based on time so that we can reduce the number of ACKs sent. Power savings are obtained here because the receiver saves by having to send fewer ACKs and the sender saves by having to respond to fewer ACKs. Also the contention and collisions at the MAC layer would be reduced thus contributing to the overall power savings.

Another point to bear in mind is that the IEEE802.11 MAC protocol has a low bit error rate and also has link layer acknowledgements and retransmissions. Hence the assumption of having an error prone wireless link is not entirely true from the point of view of the TCP layer. The TCP layer in fact can view the link to be quite error-free and so should be tuned accordingly to get better power and throughput performance.

Finally, we observe that most of the options discussed above are important for LFN's and/or very high-speed networks. For low-speed wireless networks, it would be a performance optimization to NOT use these options. A TCP user concerned about optimal performance over low-speed wireless paths should consider turning these extensions and options off for low-speed reliable wireless paths.

III. Experimental Study

We ran experiments to evaluate the potential for energy savings using the above modifications. For our testbed we used two Samsung SENS 800 laptops with Pentium-90Mhz processors and 24MB RAM and one Toshiba Liberetto 100 with 32MB RAM and a 266MHz Pentium. The operating system installed on the Samsung laptops was RedHat Linux v6.1 while the Liberetto ran Win95. The wireless PCMCIA cards used were Lucent WaveLAN TURBO 11Mb SILVER with 64-bit encryption capability.

The power characteristics of the WaveLAN cards is the following: Doze Mode – 10mA, Receive Mode – 180mA, Transmit Mode – 280mA, Power Supply – 5V. These wireless PC cards use the IEEE 802.11b standard as the MAC (CSMA/CA) protocol and the transmit range varies depending on the transmit speed. They support 4 speeds namely 11Mb/s, 5.5Mb/s, 2Mb/s and 1Mb/s. The R-F Frequency band range is 2.4GHz and the number of usable channels is 11, as specified by the FCC.

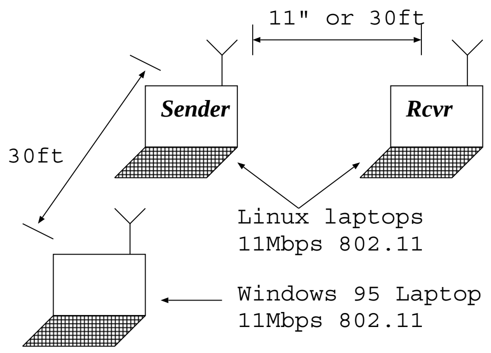


Fig. 1. Experimental Setup for single-hop experiments.

A. Methodology

We used the driver [1] developed by Andreas Neuhaus and the version of the driver was waveLan/IEEE driver v1.0.3. The driver is available for Linux Kernel v2.x.x. This driver allowed us to modify the segment MTU size used in transmissions. In ad-hoc mode the driver allowed us to setup the speed of transmission and also the channel to be used. For our experiments we selected channel 1 (the default) and set the speed to its maximum i.e. 11Mb/s. The encryption was turned off. For the initial experiments the two Samsung laptops were placed about one inch from each other and the distance between the two antennas was about 13 inches. We repeated the experiments for the case when they were 30ft apart and did not notice any change in the results. For all the experiments one of the Sansung laptops was used as a sender and the other as a receiver. The third laptop was used as a source that generated interfering traffic. In the first set of experiments (section III-B) we did not use the interfering source because we wanted to reduce the number of variables in the experiment. We then repeated some of the more interesting experiments with the interfering source turned on and report those results in section III-C.

For each experiment the battery for the sender was always fully charged for 2 hours with the laptop in off/charge mode. It should be noted that the charge of the battery is typically restored to its full capacity in about 1.5 hours but the laptop was kept in the same mode (off/charge) for another 0.5 hours. The setup of the laptops was changed so that no power saving feature was on. All the various devices (disk, display, I/O devices etc) of the laptop were in full power consumption mode as long as the battery lasted. Before the start of the experiments the monitor display of the sender was turned off and the experiment was started by disconnecting the power connector from the sending node and hence the sender operated exclusively on its battery. The experiment itself consisted of the sender transmitting a 1MB buffer continuously to the receiver using a TCP socket. This buffer, when received at the receiver, was discarded and the receiver was immediately ready for the next buffer. Tcpdump with appropriate filter settings was run on the receiver to record the amount of time the sender was alive and the number of bytes the receiver received from the sender.

One concern we had about the above methodology was whether the energy consumption by the other hardware devices within the laptop (disks etc.) would swamp out the energy consumed by data transmission and processing. If so we would not be able to make meaningful statements about TCP's energy behavior. Therefore, before we ran any experiments we performed a benchmark where we charged the laptops and let them run until discharged without transmitting any data. We then repeated this experiment but performed data transmission where the sender sent out pings continuously. The time to discharge without any data transmission was 223 minutes while with time to discharge in the second case was about 101 minutes. The difference in these two times (i.e., with and without data transmission) is therefore significant with the data transmission case resulting in a 50% reduction in battery lifetime. Therefore we note that data transmission and processing does consume a significant amount of battery power and, in addition, can be reliably measured by measuring the time to discharge of the laptop's battery.

A second concern we had was the reliability of the laptop's battery itself and whether the battery would have a shortened lifetime after several hours of experiments. To ensure that the battery was reliable over the course of our experiments we measured the time to discharge (without any data transmission) periodically and compared this value with the time we had measured when we started our experiments. Thus far the battery has proved to be reliable with no significant reduction in its lifetime or capacity.

B. No interfering sources

The first set of experiments we ran did not use the third laptop as an interferer. The reason was that we wanted to reduce the number of variables in order to better understand the impact of turning on/off the various options discussed earlier. In section III-C we discuss the results obtained by running some of the more interesting experiments with an interferer. Finally, note that in many experiments we did not use RTS/CTS (Request To Send/Clear To Send)[1] because we had a single sender and a single receiver. However, we did evaluate how RTS/CTS would impact on the overall savings and this is reported in section III-B.7.

In sections III-B.1 to III-B.6 we use the *default settings* for all options except for the one being evaluated. The default settings are: *MTU is 1500 bytes, SACK on, Window Scale on, Time Stamp on, Header Prediction normal, Delayed ACK every 2 packets*. In section III-B.7 we use a combination of several settings to get a value for the cumulative savings possible.

B.1 Varying the MTU size

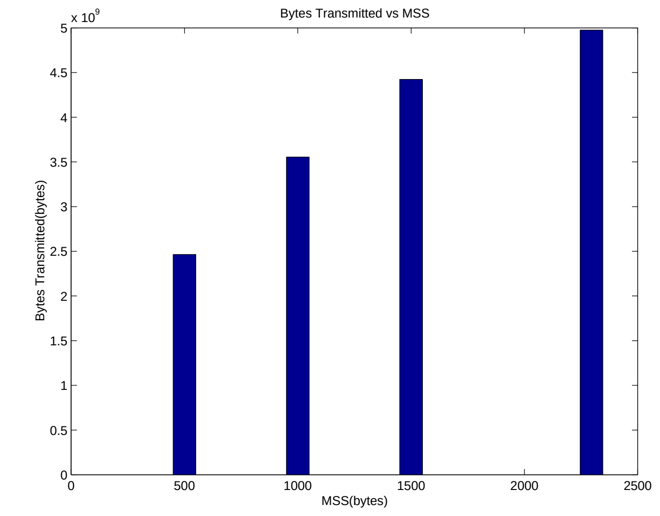
We ran experiments with four different sizes of MTU – 2296, 1500, 1000 and 500 bytes. Figure 2 shows the results of the number of bytes transported during the lifetime of the sender and the number of minutes for which the sender was active for the different MTU sizes. As we can see the number of bytes transported increases as the MTU size increases. This is because of three reasons:
1. With larger MTUs we save on header overhead.
2. The amount of protocol processing does not depend on the packet size, therefore with larger MTUs the total protocol processing is reduced.
3. Once the sender acquires the channel, more data is sent if we use larger MTUs.

It is interesting to note that the sender's lifetime *decreases* with increasing MTU sizes. The reason for this is quite simple – with larger MTU sizes the sender transmits a lot more data and therefore consumes more energy. Finally, note that with larger MTUs the total protocol processing overhad at the receiver is also reduced thus resulting in energy savings there as well.

B.2 Time Stamp Option

Figure 3 shows the effect that turning the time stamp option off has on the number of bytes transmitted by the sender. Note that we only ran this experiment with the maximum MTU size (2296 bytes) because using the maximum size is more efficient than using smaller sizes (see section III-B.1). In this figure we

[1] This is a mechanism in 802.11 that allows a sender to reserve the channel prior to sending data.



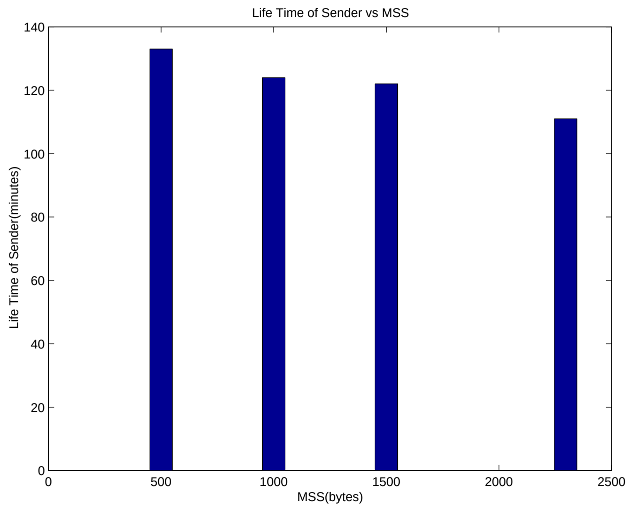
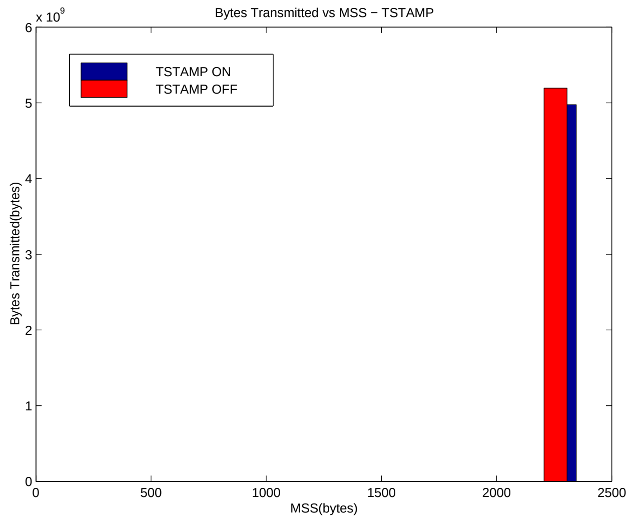


Fig. 2. Effect of varying the MTU size (no RTS/CTS).



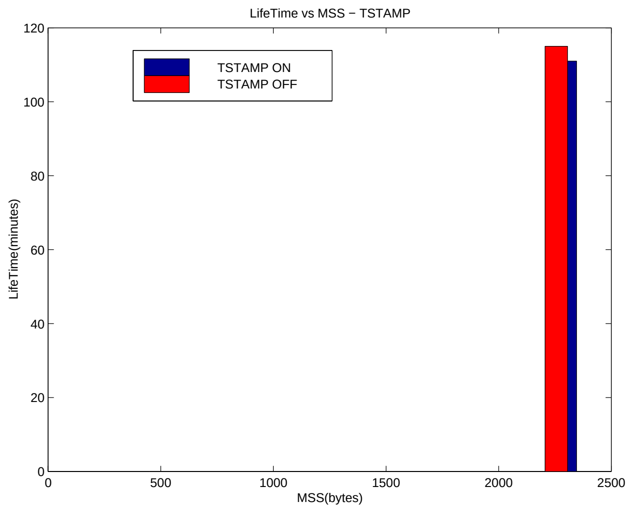


Fig. 3. Effect of using the Timestamp option (no RTS/CTS).

see that the effect of turning this option off is quite significant and this can be explained by the fact that turning this option off causes the frequent fast path to be much faster because, if there is no time stamp, then there is no processing for PAWS checking. However, when combined with modified header prediction in which the PAWS checking is done away with, we will not get any improvement by having this option off. So it is better to have this option on and at the same time get the saving in energy by modifying the header prediction to get rid of the PAWS checking.

B.3 Window Scale Option

Figure 4 shows the effect that turning the window scale option off has on the bytes transmitted by the sender. As explained previously this option does not cost much protocol processing and so the improvement is minimal.

B.4 SACK option

Figure 5 shows the effect doing away with the SACK option. Here we vary the MTU size and measure the total number of bytes transmitted with and without SACK. In this graph we see that turning off SACK only helps for the maximum MTU size of 2296 bytes. For all other MTU sizes we get better performance by having the SACK option turnd on. This can be explained by the fact that the SACK option only increases efficiency when there are multiple losses in a single window. As the MTU size increases a single window contains fewer number of packets and hence the probability of having multiple losses from within a single window deceases. So for large MTU it is in fact better to have the SACK option off and save power by not having to undergo the SACK protocol processing instead.

B.5 Header Prediction Modification

Figure 6 shows the effect of header prediction modification on the number of bytes transmitted by the sender. As mentioned earlier the improved efficiency is due to the fact that the PAWS checking is moved out of the frequent fast path. This saving is quite significant and is comparable to the saving achieved by turning the time stamp option off. But, as explained, it is better to have the time stamp option on and at the same time modify the header prediction so as to get the same effect.

B.6 Delayed ACK implementation

Figure 7 shows the effect of the implementation of delayed ACK based on time on the number of bytes transmitted by the sender. This improvement is due to the fact that fewer number of ACKs are sent by the receiver to the sender and fewer number of

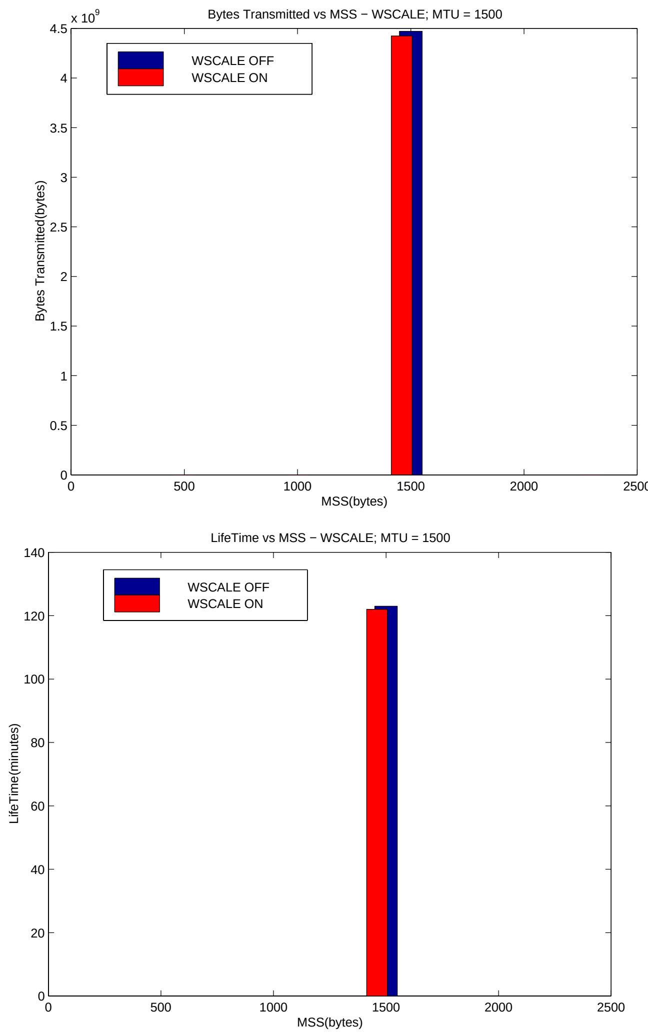


Fig. 4. Effect of turning on/off the window scale option (no RTS/CTS).

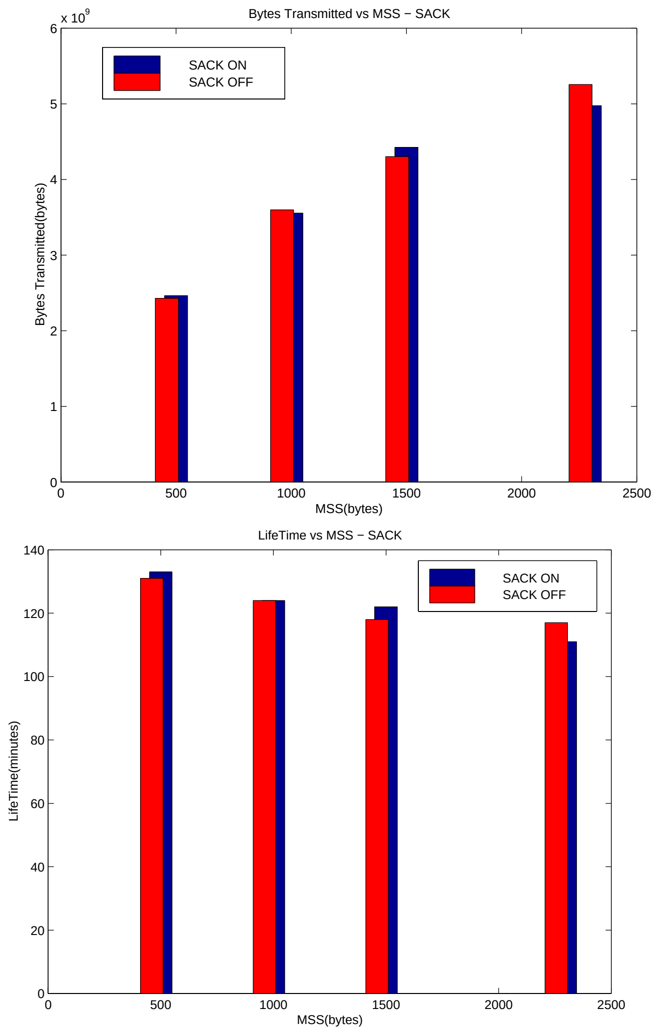


Fig. 5. Comparison of bytes transmitted with and without SACK (no RTS/CTS).

ACKs have to be received and processed by the sender. Hence the savings are at both the receiver and the sender. On examining the tcpdump output we observed that without the modification one ACK is sent for every two packets received but with the modification there is a delayed ACK every 500ms if there are no errors in transmission.

B.7 Cumulative Savings

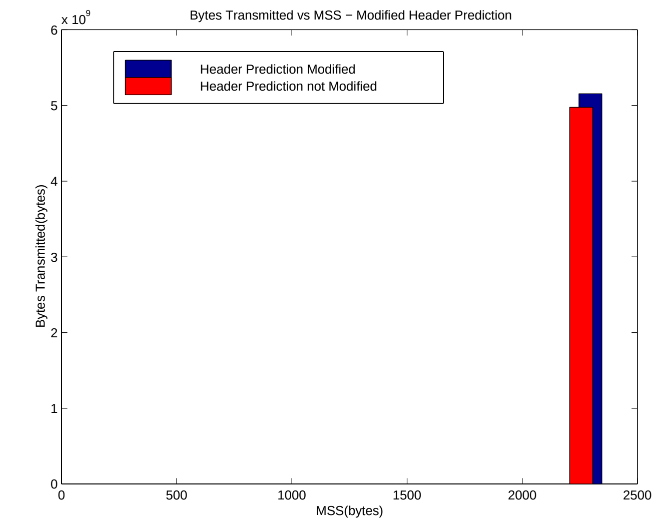
Figure 8a shows the comparison between unmodified and modified TCP with all the significant optimizations incorporated. From the above discussion we know that the optimizations that should be incorporated are: *MTU increased to 2296, SACK option off, header prediction modified to remove the PAWS check from the critical path and delayed ACK implemented based only on time*. This graph shows that the energy efficiency achieved is almost about 25%.

Figure 8b is the same as Figure 8a with RTS/CTS enabled. For all the above experiments as mentioned we had this feature turned off but for this experiment we had this featured turned on. For a packet of size greater than 1000 bytes there would be an initial RTS/CTS handshake performed between the two MAC layers to ensure that the media is reserved. By choosing the size as 1000 bytes for mandatory handshaking we ensure that ACKs do not need any initial handshaking whereas data packets do need handshaking. This figure shows us that there is a greater saving in energy with the RTS/CTS feature turned off. This can be attributed to the extra processing involved in processing and exchanging RTS/CTS packets.

Figure 9 shows the savings in the case that all optimizations are turned on except that the MTU size used is 1500 bytes. This experiment was done because of the observation that Ethernet supports a maximum MTU of 1500 bytes and if a node in an ad hoc network sets up a connection that traverses a LAN (Ethernet), then the max MTU TCP will be able to use will be no more than 1500 bytes. As we can see, there are still significant savings (8%) though not as dramatic as in the case that a 2296 byte MTU is used.

B.8 Variable Distance

For this experiment we increased the distance between the sender and receiver to 30ft. However within experimental limits we found the results to be similar to the results obtained when the distance between the sender and receiver was 13 inches.



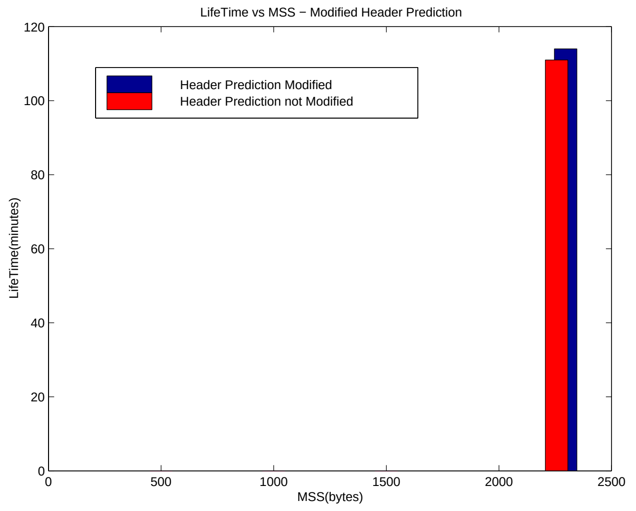
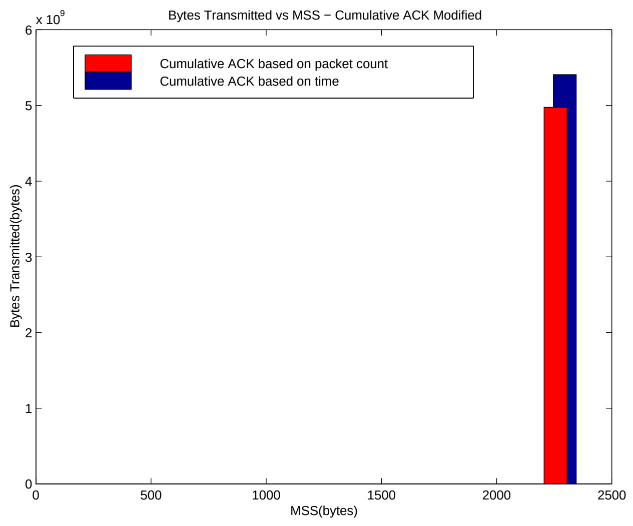


Fig. 6. Effect of modifying header prediction (no RTS/CTS).



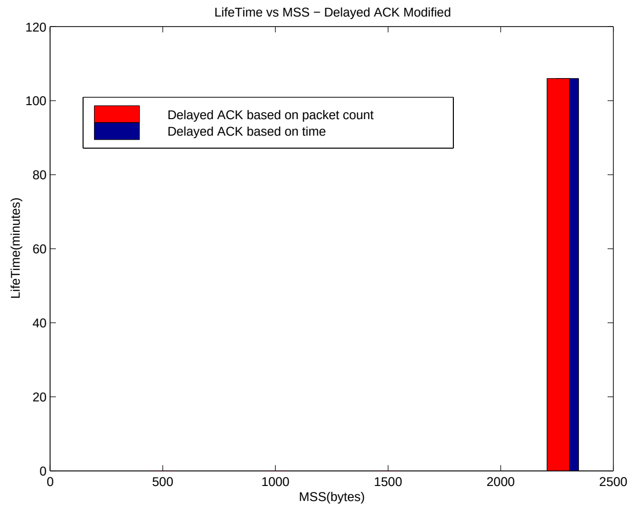


Fig. 7. Effect of delayed ACKs (no RTS/CTS).

B.9 Errors at the TCP layer

We measured the total number of retransmissions at the TCP layer of the sender during its life-time. This gave us a measure of the quality of the wireless link. The number of retransmissions in the course of the experiment for an MTU of 1500 was found to be about 50. This is quite a high error rate considering the fact that the MAC protocol is considered to be reliable. However the number of retransmissions is generally high because of the timer interactions between the TCP and the MAC layers and also because of the competing and redundant retransmissions by the TCP Layer since duplicate ACKs are not suppressed by the MAC layer. Similar error rates were observed for other MTU sizes as well.

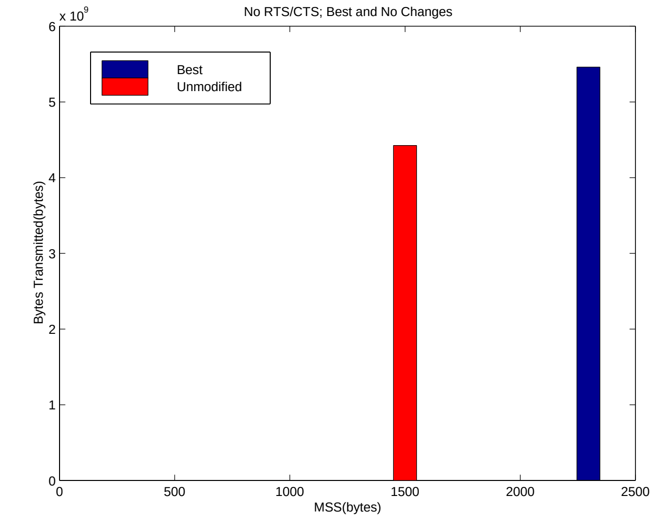
C. Interfering Trafic

To verify our results for the case of interfering traffic we ran additional experiments. A total of 8 sets of experiments were run and Tables I and II summarize the results that we obtained. In the experiments summarized in Table I continuous interfering traffic was generated by the third laptop. While for the experiments summarized in TableII the interfering node would send about 5MB of data every 5 minutes. In these experiments we included all the modifications to TCP and we used an MTU of 1500 bytes. RTS ON means that all the three nodes had the setting for RTS/CTS handshaking on for packet exchanges in excess of 100 bytes. From the table we can see that for the case of continuous traffic the lifetime of the sender was generally high but the bytes transmitted was generally low. Also there is a significant difference between the SACK on and SACK off case when RTS/CTS is off. This shows that a huge amount of power is expended in executing the code for SACK and hence the total number of bytes transmitted is lower during the lifetime of the sender. This is not the case with intermittent load however. This means that when we have no initial handshaking and the

Lifetime	SACK OFF	SACK ON
RTS/CTS ON	128	132
RTS/CTS OFF	124	117

Bytes Transmitted	SACK OFF	SACK ON
RTS/CTS ON	2970847921	3058387131
RTS/CTS OFF	3811377004	3647866576

TABLE I
CONTINUOUS INTERFERING TRAFFIC CASE.



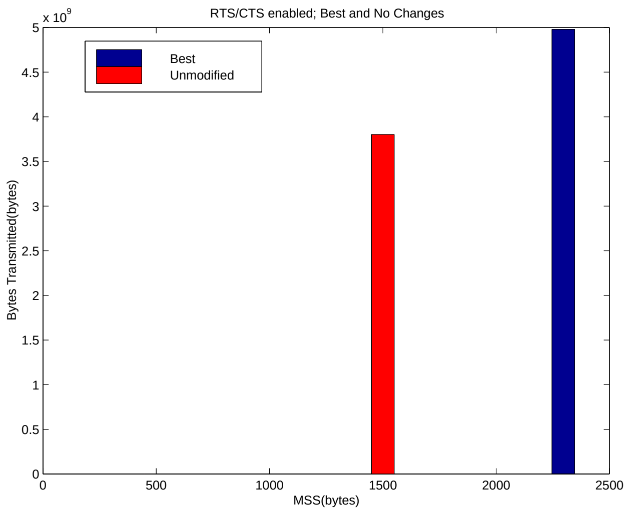
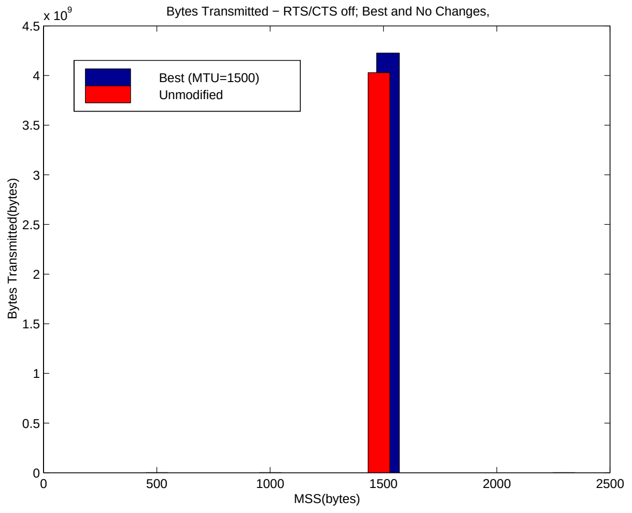


Fig. 8. Cumulative savings.



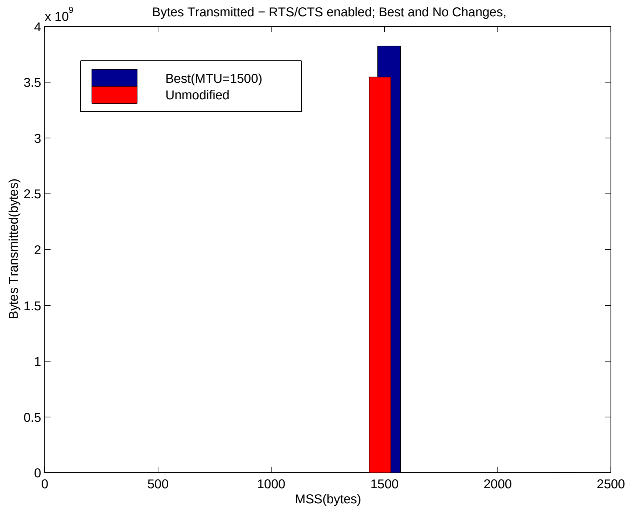


Fig. 9. Cumulative savings with smaller MTU of 1500 bytes only.

Lifetime	SACK OFF	SACK ON
RTS/CTS ON	111	112
RTS/CTS OFF	106	109

Bytes Transmitted	SACK OFF	SACK ON
RTS/CTS ON	3646355620	3631339254
RTS/CTS OFF	4183867981	4185749921

TABLE II
INTERMITTENT INTERFERING TRAFFIC CASE.

traffic load is high it is better not to have the SACK option on - since a lot of power would be consumed in going through the SACK code. Again it should be noted that depending on the traffic conditions a decision about having the SACK option on or off has to be made. Other than that the table shows results that are intuitive.

IV. FUTURE WORK

In the next stage of this experimental study are exploring questions such as, how do multi-hop wireless TCP connections perform? if the last hop of a TCP connection is over a wired network, will the use of large MTUs cause a decrease in performance (due to fragmentation)? in more complex environments with many transmitters, how will the modified TCP behave as opposed to the unmodified case? We will use a similar methodology to study these questions.

REFERENCES

[1] Andy's Homepage - Linux - WaveLAN/IEEE802.11 driver, `http://www.fasta.fh-dortmund.de/users/andy/wvlan/`

[2] Nicholas Bambos and Sunil Kandukuri, "Power-Controlled Multiple Access (PCMA) in Wireless Communication Networks", *IEEE Infocom 2000*.

[3] Jae-Hwan Chang and Leandros Tassiulas, "Energy-Conserving Routing in Wireless Ad Hoc Networks", *IEEE Infocom 2000*.

[4] V. Jacobson, "4BSD Header Prediction", *ACM Computer Communication Review*, Vol. 20, No. 1, April 1990, pp. 13-15.

[5] V. Jacobson, "Congestion Avoidance and Control", *ACM SIGCOMM '88*, Stanford, CA., August 1988.

[6] V. Jacobson, R. Braden, and L. Zhang, "TCP Extension for High-Speed Paths", RFC-1185, LBL and USC/Information Sciences Institute, October 1990.

[7] S. Singh and C. S. Raghavendra, "PAMAS: Power-Aware Multi-Access protocol with signalling for ad hoc networks", *ACM Computer Communication Review*, July 1998.

[8] S. Singh, M. Woo and C. S. Raghavendra, "Power-Aware Routing in Mobile Ad Hoc Networks", *ACM/IEEE Mobicom 1998*, pp. 181-190.