Energy Consumption of TCP Reno, Newreno, and SACK in Multi-Hop Wireless Networks*

Harkirat Singh
Department of Computer Science
Portland State University
Portland, OR 97207
harkirat@cs.pdx.edu

Suresh Singh
Department of Computer Science
Portland State University
Portland, OR 97207
singh@cs.pdx.edu

ABSTRACT

In this paper we compare the energy consumption behavior of three versions of TCP – Reno, Newreno, and SACK. The experiments were performed on a wireless testbed where we measured the energy consumed at the sender node. Our results indicate that, in most cases, using total energy consumed as the metric, SACK outperforms Newreno and Reno while Newreno performs better than Reno. The experiments emulated a large set of network conditions including variable round trip times, random loss, bursty loss, and packet reordering. We also estimated the idealized energy for each of the three implementations (i.e., we subtract out the energy consumed when the sender is idle) and here, surprisingly, we find that in many instances SACK performs poorly compared to the other two implementations. We conclude that if the mobile device has a very low idle power consumption then SACK is not the best implementation to use for bursty or random loss. On the other hand, if the idle power consumption is significant, then SACK is the best choice since it has the lowest overall energy consumption.

1. INTRODUCTION

Today, sophisticated wireless devices are gaining popularity as the platform of choice for deploying a range of mobile applications. Since these devices operate on battery power alone, it is important to ensure that energy-efficiency considerations are incorporated into the design of their hardware and software. Data communication plays a key role in many mobile applications and accounts for a large proportion of the cost in running these applications on handhelds. For example, many applications require access to remote file systems or databases (e.g., electronic record-keeping in hospitals) and are thus heavy datacom users. Likewise, some mobile applications require real-time communication for audio/video/streaming media and are also communication-heavy. We therefore believe that it is important to understand and characterize the energy cost of wireless communications and use this information in the design of the communications component of these devices.

*This work was supported by DARPA under contract number F33615-C-00-1633.

In this paper we focus our attention on the energy efficiency of three variants of TCP for connections running over wireless links. Our goal is to characterize the energy consumption as well as the throughput of these versions of TCP for a variety of wireless network conditions including loss (random as well as bursty), variable round trip times (RTT), and packet reordering. We used a testbed consisting of wireless laptops and measured the energy consumed by the sender. Our results are interesting and can be summarized as follows:

- TCP SACK consumes the lowest total energy in most scenarios and has the highest throughput.
- However, if we discount the energy consumed by the sender in idle state (i.e., when the sender is awaiting ACKs prior to transmitting more packets), SACK appears to have the highest energy cost in many cases. This is due to the fact that SACK introduces additional computational complexity at the sender thus resulting in a higher energy consumption.

This difference is interesting in mobile computing because it points to the need for a careful selection of protocols for the handhelds. If the idle energy for a handheld is very small then SACK is probably not a good choice for that device. On the other hand, if the handheld has a high idle energy cost, then SACK is a good choice since it completes the data transmission the earliest.

The remainder of this paper is organized as follows. In section 2, we define the energy metrics used to compare the energy cost of TCP. Section 3 summarizes the key differences between Reno, Newreno, and SACK. In section 4 we discuss the current state-of-the-art in energy and other performance studies of these protocols. Section 5 describes our experimental hardware and software setup and we discuss the experimental parameters used. The results are presented in section 6 and we discuss the implications of this work in section 7.

2. ENERGY CONSUMPTION IN TCP

Consider the case where a node in an ad hoc network needs to transmit B bytes of data reliably. It transmits a window of packets and waits to receive ACKs. Upon reception of ACKs, it moves its window and transmits more packets. Figure 1 shows the evolution in time of a sender where we plot the current drawn by the sender (assuming a fixed voltage) as a function of time. When the sender is

idle, it draws a fixed amount of current[1]. When a packet is to be transmitted, there is some processing energy consumed[2] in addition to the transmission energy. Likewise, when a packet is received, there is energy consumed to receive the packet (by the interface card) plus the processing energy needed to process the received packet.

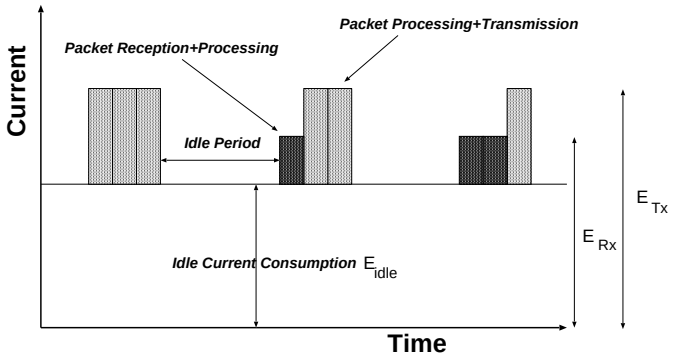

Figure 1: Total energy consumed.

In this paper, we make a distinction between the *total energy* E consumed and the *idealized energy* E_I that is consumed. The total energy refers to the total system energy, from the start of data transmission to the end, that is consumed by the system. The idealized energy refers to the total energy minus the energy consumed in the idle periods. The reason this distinction is interesting is that E_I depends on the protocol processing, transmission, and reception costs only whereas E depends on E_I as well as the *throughput* of the connection. As we will show in section 6, for some cases SACK has a lower E (than Reno and Newreno) but a higher E_I due to the additional computation involved. Finally, it is interesting to note that as the idle power consumption is minimized by improved power-management in hardware, E will asymptotically approach E_I.

2.1 Approximate Relationship between E and Throughput

We can write a simple expression for the total energy consumed by a node to transmit B bytes of data reliably as follows:

$$\begin{aligned} E &= E_{\text{idle}}(t_{\text{total}} - t_{\text{Tx}} - t_{\text{Rx}}) \\ &\quad + E_{\text{Tx}}t_{\text{Tx}} + E_{\text{Rx}}t_{\text{Rx}} \end{aligned}$$

where E_{idle} is the idle energy consumed by the sender, t_{total} is the total time needed to complete the transmission of B bytes, t_{Tx} and t_{Rx} are the time spent in transmitting and receiving packets, and E_{Tx} and E_{Rx} are the energy expended at the sender for packet transmission and reception.

The first term in the above expression denotes the *idle energy cost* at the sender. This is the energy consumed by the sender while it awaits reception of ACKs from the receiver or timeout events. Thus, if the channel has a low bit rate, or if the losses are high, the sender is likely to spend a large amount of time in the idle state consuming energy. The second term in the expression denotes the energy expended for packet transmission. For a given amount of data B, the value of this term will primarily depend on the number of transmissions and the MTU size (Maximum Transmission Unit or packet size) used. Larger MTU sizes will require fewer packet transmissions but a higher loss rate will result in a larger penalty since a larger amount of data will need to be retransmitted (recall that packet transmission cost is a sum of at least two memory copy operations, a checksum calculation, and, eventually, the actual transmission). Finally, the third term in the expression for E is the packet reception cost. If we consider the case when the data transmission is one-way only (i.e., the sender only receives ACKs), then the reception cost is actually quite small (ACKs are small packets and since they contain no data, there is a much smaller copy cost). We can therefore drop this third term from the expression for E to simplify it and obtain:

$$E = E_{\text{idle}}(t_{\text{total}} - t_{\text{Tx}}) + E_{\text{Tx}}t_{\text{Tx}} \quad (1)$$

Notice that we have also dropped t_{Rx} from the first term because the total time spent in receiving and processing ACKs is quite small.

If we assume that the *average* connection throughput is τ bytes/sec and the transmission speed is r bytes/sec we can write,

$$\begin{aligned} E &= E_{\text{idle}}(B/\tau - B/r) + E_{\text{Tx}}B/r \\ &= (B/\tau)E_{\text{idle}} + (B/r)(E_{\text{Tx}} - E_{\text{idle}}) \\ &\propto 1/\tau \end{aligned} \quad (2)$$

Thus, we see that the total energy consumed is *inversely* proportional to the average throughput achieved by the connection.

3. OVERVIEW OF RENO, NEWRENO, AND SACK

All the current TCP implementations are based on TCP Tahoe that incorporated algorithms for slow-start, congestion avoidance, fast retransmit. and modifications to the formula for estimating round-trip times (RTT), see [16]. TCP Reno is essentially similar to Tahoe but with a modified fast retransmit algorithm that includes fast recovery as well. When the sender receives three duplicate ACKs, it retransmits one segment and reduces its ssthresh by half (minimum of two segments). However, unlike Tahoe which performs slow-start, Reno increases its congestion window more rapidly by setting it to min(recvr_window, CWND + ndup). In other words, after retransmitting one segment and reducing ssthresh by one half, Reno sets ndup to 3 and increments it for every duplicate ACK received. When the sender receives an ACK for new data, it exits fast recovery by setting ndup to zero. It is easy to see that Reno's fast recovery algorithm is optimized for single packet losses from a window of data and will not perform well for multiple losses. In this case the retransmit timer will go off resulting in congestion avoidance and very low throughput.

TCP Newreno tries to overcome the shortcomings of Reno in the presence of bursty losses by using information contained in *partial* ACKs differently. A partial ACK is an ACK that acknowledges some but not all of the unacked packets in the sender's window. In Reno, a partial ACK takes the sender out of fast recovery. In Newreno, on the other hand, a partial ACK received during fast recovery is taken as an indication that the packet following the partial ACK was lost and should be retransmitted. Thus, in the presence of multiple losses from within a window, partial ACKs ensure that the lost packets are retransmitted without waiting for retransmit timers to go off. Newreno only comes out of fast recovery when all the packets that were in the window at the time fast recovery started are acknowledged.

[1]We are considering the idle energy used by the sender as a whole, i.e., the interface card, the processor, the memory and any other devices that are powered on.

[2]The main source of energy consumption is the copy operation (user space to kernel space and then to the interface card).

TCP SACK, built on top of Newreno, adds an additional capability that allows faster recovery in the presence of multiple packet losses. When the receiver receives a block of data which is out of sequence, that data creates a hole in the receiver's buffer. This causes the receiver to generate a duplicate ACK for the segment preceding the hole. The receiver also includes the starting and ending sequence numbers of the data that was received out of sequence. This information is a SACK. The first block in a SACK option is required to report the data receiver's most recently received segment, and the additional SACK blocks repeat the most recently reported SACK blocks. This algorithm generally allows TCP to recover from multiple segment losses in a window of data within one RTT of loss detection.

When a sender detects a lost packet (via three duplicate ACKs), it retransmits one packet, cuts the congestion window by half, and enters fast recovery as in the case of Reno and Newreno. SACK maintains a variable called *pipe* that estimates the number of packets in flight. It is incremented for every transmission and is decremented when a duplicate ACK is received containing a new SACK. The sender maintains a list of segments deemed to be missing (based on all the SACKs received) and retransmits segments from this list when *pipe* is less than CWND. Finally, when partial ACKs are received, the sender decrements *pipe* by two rather than one (see [8] for a discussion of why). SACK exits fast recovery under the same conditions as Newreno.

Previous papers (see section 4) have compared the throughput of different versions of TCP. The results indicate that SACK has the highest throughput for a large percentage of network conditions. Based on equation (2), we can therefore predict that SACK would consume the *lowest* total energy (E). This is borne out in our measurements as we discuss in section 6. The discussion of SACK makes it clear that the sender needs to execute more code to maintain and use the SACK-related data structures. We had assumed that this added cost would be negligible. However, as section 6 shows, the idealized energy cost E_I of SACK is higher (and measurable) than Reno and Newreno for many cases.

4. LITERATURE SURVEY

Several authors have studied the behavior of TCP and its variations under different networking conditions.

[6] states that networks with wireless and other lossy links suffer from significant non-congestion-related losses due to reasons such as bit-errors and handoffs. The results show that TCP responds to all losses by invoking congestion control and avoidance algorithms, resulting in degraded end-to-end performance. The authors compared the use of Explicit Loss Notification (ELN), I-TCP, Snoop, and SACK to improve performance. They implemented and tested the various protocols in a wireless testbed consisting of Pentium PC base stations and IBM ThinkPad mobile hosts communicating over a 915 MHz AT&T Wavelan, all running BSD/2.0. The primary result of their work was that SACK implemented with Snoop has a 30% higher throughput than the other protocols for bursty errors.

[8] shows that the SACK algorithm performs better than several non-sack based recovery algorithms when 1–4 segments are lost from a window of data. Reno's Fast Recovery algorithm is optimized for the case when a single packet is dropped from a window of data. The Reno sender retransmits at most one dropped packet per round-trip time. Reno suffers from performance problems when multiple packets are dropped from a window of data. For the scenario with two dropped packets, the sender goes through Fast Retransmit and Fast Recovery twice in succession, unnecessarily reducing the congestion window size twice. For the scenarios with three or four packet drops, the Reno sender has to wait for a retransmit timer to recover putting the sender into Slow-Start. However, in all such cases (when multiple packets are lost from a single window of data), Newreno can recover without a retransmission timeout, retransmitting one lost packet per round-trip time until all of the lost packet from the window have been retransmitted.

[4] compared Reno, Newreno, and SACK for communication over satellite links. They used two Intel machines with NetBSD1.1, two cisco routers and ACTS VSAT connected to a satellite, and a hardware emulator for dropping packets. They dropped 1,2,3 and 4 packets for different experiments, used two data sizes (200KB and 5MB) and experimented with three bit error rates of 10e-5,10e-6 and 10e-7. For 1,2,3 and 4 packet loss the performance was similar to [8]. For losses of 10e-7 and 10e-6, they saw the same results as ours, and for the high loss case 10-5, they show that all of the TCP variants performed poorly and SACK was similar to Reno and Newreno.

[10] evaluated the performance of Reno, Newreno, and SACK for a satellite network. Experiments were carried out in the simulator ns. They used a RTT in the range of 100 – 600 msec. The paper shows that using partial ACKs to trigger retransmissions, in conjunction with SACK, improves performance when compared with TCP using fast retransmit/fast recovery alone. Specifically, SACK-Newreno is better than SACK-Reno which is in turn better than Reno.

[7] performed experiments in which two PCs running FreeBSD were connected by a SUN UltraSparc acting as a router that inserted link delays. The link delay was 25 msec and bandwidth was limited to 2 Mbps. The buffer space at the sender and receiver was set to 16KB. They studied two types of losses – random uniform loss (0% to 9%) and bursty loss where three packets were dropped randomly. In the random uniform loss case, they observed that if the loss probability is low then both Reno and SACK behave similarly. Likewise, if the loss probability is high, these two protocols again behave similarly because retransmitted packets at high loss probabilities will be lost causing SACK to timeout which is a normal case with Reno. However, if the loss probability is between 2% and 4% then SACK outperforms Reno. In the burst loss case, SACK improves throughput by a significant amount (60% to 70%). This is because for Reno the loss of three isolated packets or three consecutive packets results in the same behavior while SACK recovers quickly from a bursty loss and does not cause timeouts.

[13] is the first paper describing a SACK implementation in FreeBSD, and it also proposes TSACK (so that receiver can use the SACK option based on the timestamp option) which was not widely accepted. The paper gives a distribution function indicating the number of blocks which are present in the SACK queue. This data shows that in the case of a bursty loss, the queue size is smaller than in the case of random uniform loss (with the same average loss probability). An implication of this is that the idealized energy cost (E_I) of SACK should be smaller in the case of bursty loss than random uniform loss – this observation is confirmed in our experiments.

[5] discusses the existence of packet reordering in the internet and gives reasons why it may be undesirable to eliminate reordering.

There are situations where average packet latency can be reduced, link efficiency can be increased, and/or reliability can be improved if reordering is permitted. Examples include certain high speed switches within the Internet backbone and the parallel links used over many Internet paths for load splitting and redundancy

5. EXPERIMENTAL SETUP

Characterizing the energy consumption of TCP in multi-hop wireless networks poses a challenge due to the fact that, on the one hand, to emulate complex networking scenarios we need to use a simulator like NS-2 [1] while, on the other hand, measuring the actual energy consumed requires us to use a real system (the energy models in NS-2 do not consider node costs whereas simulators like SimplePower [2], which simulate node-level energy costs, do not simulate networks of communicating nodes). The approach we settled on was a hybrid one – we used an actual wireless network (as illustrated in Figure 2) where we could measure the energy consumed by the sender. And, to simulate a wide range of network properties (packet loss, delay, etc.), we ran Dummynet [12] at one of the intermediate nodes.

As shown in Figure 2, we use four laptops equipped with 802.11b 11Mb/s cards. The path from the sender to the receiver traverses three hops and at node C we run Dummynet to simulate a wide range of network conditions. Specifically, we use Dummynet to introduce delays in both directions (for data packets as well as for ACKs), simulate packet loss (random uniform loss as well as bursty loss), and to periodically reorder packets. Dummynet can also be used to emulate different bandwidths though we did not use this facility in our studies. Our experiments used the FreeBSD 4.3 kernel and our own SACK implementation derived from the implementation in OpenBSD2.9 [3]. Finally, to measure the energy consumed, the sender's power supply is connected to a HP 34401A multimeter that is controlled by a separate laptop. We measure the current every millisecond and the raw data is stored on the measuring laptop. Our sender wasa Toshiba Satellite laptop whose idle current draw was 1.2A at a voltage of 15V. Figure 3 shows a sample trace of a simultaneous measurement of the total system energy and the energy draw of the radio card. See [9] for a detailed characterization of the radio energy consumption.

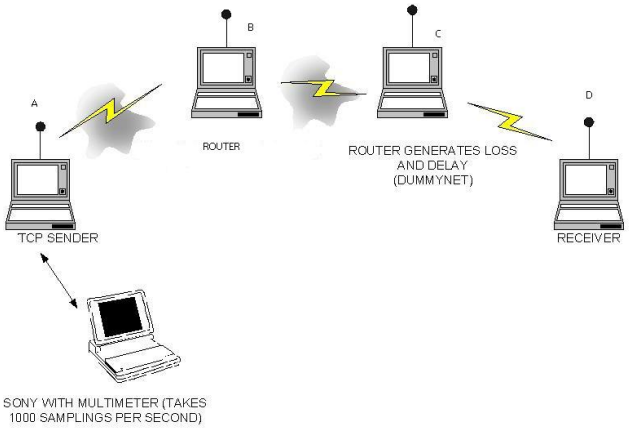

Figure 2: Experimental Setup.

5.1 Experimental Design

In section 6 we describe the results obtained for three different conditions: *random packet loss*, *bursty loss*, and for the situation when there is *packet reordering*. Let us use this division to explain our experimental design. Table 1 describes the variable experimental parameters and their values for the case when packets are lost randomly uniformly (section 6.1). As the table indicates, we experimented with increasing RTT values, two MTU sizes, and different loss probabilities. The total number of experimental settings is therefore $5 \times 3 \times 2 \times 2 \times 3 = 180$. For each experimental setup, we ran between 10 and 15 repetitions and computed 90% confidence intervals.

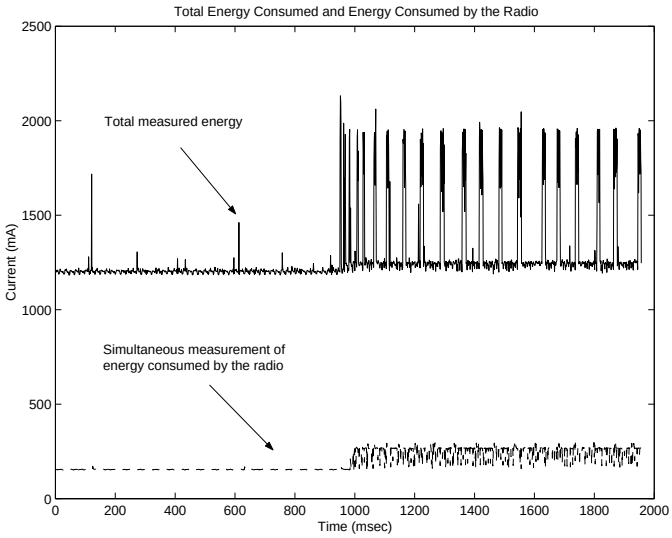

Figure 3: Sample of measured data.

Parameter	*Values*
Average RTT	15, 40, 70, 100, 130 msec
Packet Loss	1%, 5%, and 10%
MTU Size	512 and 1500 bytes
RTS/CTS	ON, OFF
Protocols	Reno, Newreno, and SACK

Table 1: Experimental parameters for the loss case, Section 6.1.

For the bursty loss case (Section 6.1.1), we reduced the number of experimental parameters (Table 2) resulting in fewer experimental scenarios (a total of 15). We decided to use only one bursty loss scenario because it clearly distinguishes between the three protocols. In addition, we decided to drop the RTS/CTS ON case because the results we obtain are very similar to the RTS/CTS OFF case. The packets were dropped with a probability of 0.85 for one second after 12 seconds of zero loss. With an MTU of 1500 bytes that we used, the average number of dropped packets was 40 – 80. For the packet reordering case (Section 6.2), the number of experimental setups we used was 30 (see Table 3).

6. EXPERIMENTAL RESULTS

We evaluated Reno, Newreno, and SACK using three metrics:

- *Total Energy/bit* E measured in Joules/bit. This includes the energy consumed while the sender is idle.
- *Idealized Energy/bit* E_I measured in Joules/bit. This measure excludes the idle time energy and thus more closely approximates the cost of the various protocols.

Parameter	*Values*
Average RTT	15, 40, 70, 100, 130 msec
Bursty Packet Loss	85% loss rate for 1 second every 12 seconds
MTU Size	1500 bytes
RTS/CTS	OFF
Protocols	Reno, Newreno, and SACK

Table 2: Experimental parameters for the bursty loss case, Section 6.1.1.

Parameter	*Values*
Average RTT	15, 40, 70, 100, 130 msec
Packet Loss	None
Reorder Rate	1% and 5% packets reordered
MTU Size	512, 1500 bytes
RTS/CTS	OFF
Protocols	Reno, Newreno, and SACK

Table 3: Experimental parameters for the packet reordering case, Section 6.2.

- *Goodput* in kb/s.

Section 6.1 looks at the case when the packet losses are random uniform, section 6.1.1 looks at the situation when the losses are bursty, and section 6.2 looks at the case when some packets are reordered in the network.

6.1 Random Uniform Loss Case

Table 1 outlines the experimental design for this group of experiments. For each case, we transmitted 5MB of data using ttcp. Figures 4, 5, 6 plot (1) the total energy E as a function of different RTTs for two values of the MTU and (2) the goodput as a function of RTT[3]. We can make several observations based on these results:

- In general, smaller MTUs are better (i.e., consume less energy) at high losses (see Figure 6, the 10% loss case) whereas larger MTUs are better at low loss (see Figure 4, the 1% loss case). This observation has been made by several previous researchers when throughput was the metric studied.
- For most cases, SACK consumes the least amount of energy. The reason is that SACK retransmits missing segments earlier than Reno (which waits for a timeout in most cases) and Newreno (which does not know which of the unacked segments is missing at the receiver). Thus, SACK completes transmission of the data sooner resulting in lower overall energy.
- Figures 4, 5, 6 show that for the 1% and the 5% loss cases, SACK consumes the least amount of energy (for both MTU sizes) while for the 10% loss case, SACK consumes the most energy at an MTU of 1500. There are two reasons for this:
  - When the loss rate is high, with large MTU, the number of segments in a window is small and hence the possibility of receiving 3 dupacks is small. In this case the fast retransmit algorithm is seldom triggered. Thus SACK does not really help improve throughput (see also RFC 3042 which notes the same problem and suggests the use of a "Limited Transmit Algorithm" which we have not implemented).

    Figure 8 shows that the number of timeouts with or without SACK at MTU 1500 is about the same. With a smaller MTU, on the other hand, SACKs do help more and therefore we see that at higher packet loss rates, the timeouts for the SACK case are smaller.
  - Since the SACK throughput is small, the total energy consumed by SACK will be at least as high as Reno and Newreno. However, SACK adds an additional burden in terms of computational overhead and this results in a higher total energy consumption.
- The energy consumed increases with an increase in RTT. This is clear because the throughput falls with an increase in RTT.
- We plot the total energy per bit as a function of goodput for the 10% loss case and a MTU of 512 bytes in Figure 7. As we can clearly see, as the goodput decreases the energy used increases and there is an inverse relationship between these two quantities as predicted by equation 2.

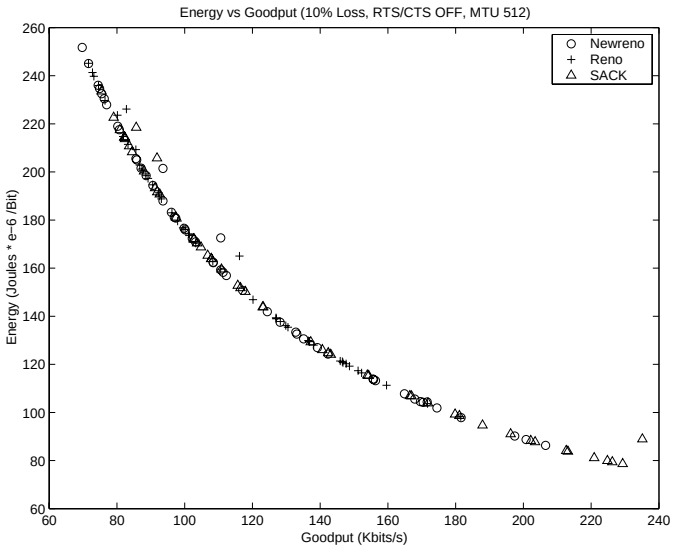

Figure 7: Relationship between energy and throughput (RTS/CTS Off).

In Figure 9 we plot the ideal energy E_I consumed for the 1% and 10% loss cases. At the 1% loss case we see that SACK actually consumes *more* energy than either of Reno or Newreno at both MTU sizes! This is because, as we noted above, SACK has an added computational burden that only becomes visible when we discount the idle energy cost. We see a similar pattern in the 5% and 10% loss cases as well.

Table 4 summarizes the relative performance of the protocols as measured by the total energy, the goodput, and the idealized energy for all of the loss and MTU combinations (we derive these rank orderings based on a comparison of the mean values of energy, goodput, and idealized energy for the different RTTs and MTUs). We note that the total energy and goodput are inversely related. Thus,

[3]These graphs correspond to the case when RTS/CTS was turned off. The graphs for the ON case are quite similar and have been left out for ease of explanation.

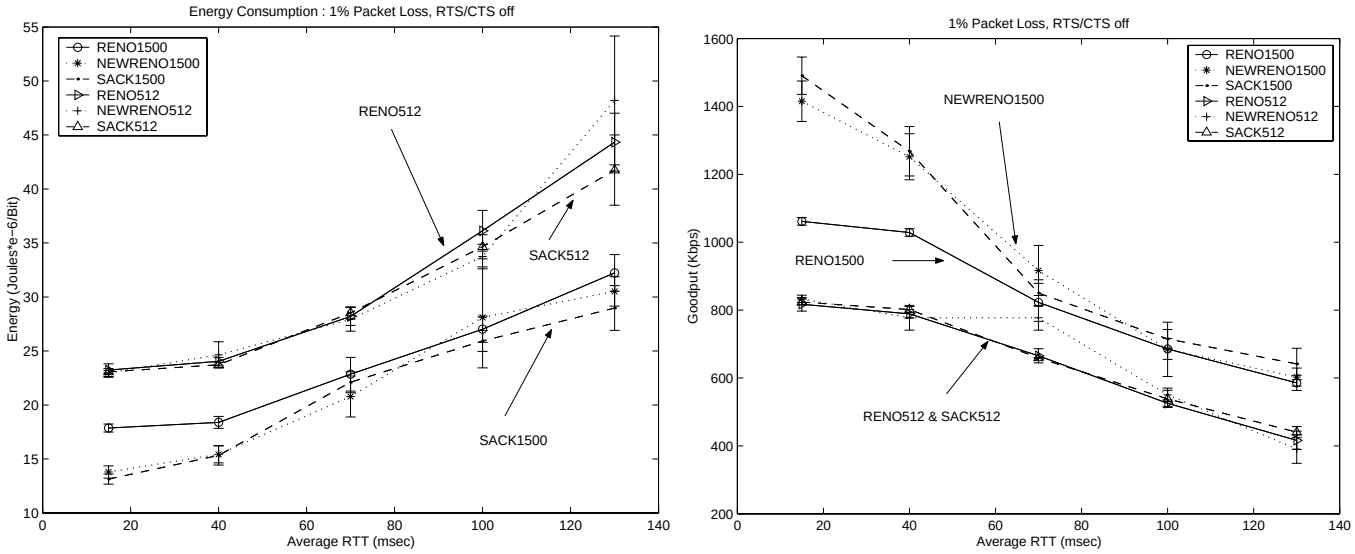

Figure 4: Total Energy E per bit and Goodput for 1% packet loss with no RTS/CTS.

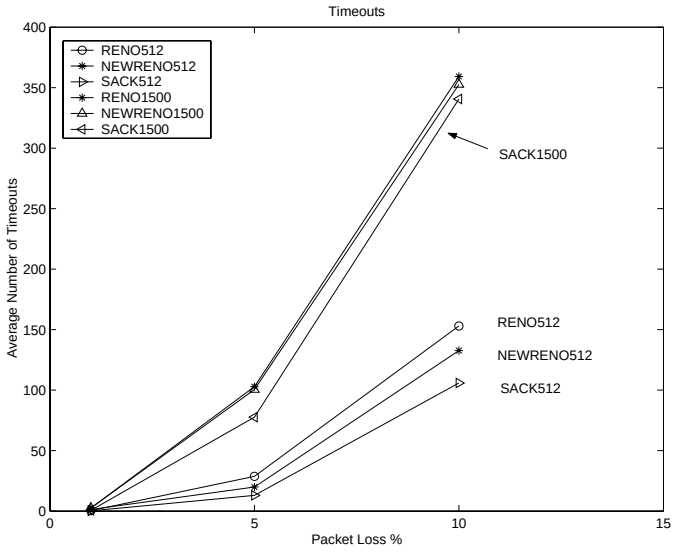

Figure 8: Comparison of the number of timeouts.

the protocol that has the highest goodput also has the lowest total energy. We also note that in all cases except MTU 1500 & 10% packet loss rate *SACK has the highest goodput and the lowest total energy*. In this one case, Newreno has the lowest energy and highest throughput.

When we look at the idealized energy, on the other hand, we see that *either Newreno or Reno use the lowest idealized energy* in all cases except in the MTU 512 & 5% packet loss case where SACK and Newreno perform equally well. In general, we believe that SACK performs poorly with respect to this metric because of the additional data structures and computation it performs. Newreno and Reno, on the other hand, have very similar computational overhead. The only reason Newreno sometimes performs better in some cases (e.g., MTU 512 & 10% packet loss) is that in fast recovery it retransmits unacked packets before the retransmit timer goes off. Reno, on the other hand, only retransmits one packet (the one that received three duplicate acks) and then retransmits the remaining packets when the retransmit timers go off. Figure 8 plots the number of timeout events for different MTU sizes. As we can see, at a MTU of 512, Newreno has fewer timeouts than Reno and thus performs better by this metric. However, even though SACK has the least number of timeouts, its computational cost is high enough to offset any gain due to its better throughput.

The lesson here is that if the wireless devices can be designed to enter deep power saving modes when there is inactivity (i.e., reduce the idle cost as much as possible), then SACK may not be a good choice for mobile environments. On the other hand, if the idle power remains high, then SACK definitely results in overall energy savings.

6.1.1 Bursty Loss Case

Figure 10 plots the energy and idealized energy cost for the experimental setup described in Table 6.1.1. In the energy plot, we plot both the total energy (E) as well as the idealized energy (E_I) in the same graph. We see that SACK has the lowest total energy while Newreno has the lowest idealized energy consumption. The reason for this behavior is that in the case of a bursty loss, Newreno will retransmit lost packets without waiting for the retransmit timers to go off (based on partial ACKs received). SACK, likewise will retransmit these packets as well (as indicated by the SACKs) but it also has the added overhead of maintaining SACK-related data. This additional cost results in SACK having a higher idealized energy cost even though its goodput is the highest.

6.2 Packet Reordering Case

Figures 11 and 12 plot the total energy and idealized energy for the reorder case (see Table 6.2). We can summarize the main findings as follows:

- SACK is by far the winner in terms of total energy as well as idealized energy. The goodput of SACK is also the highest.

  Let us first look at the reason for SACK's better goodput

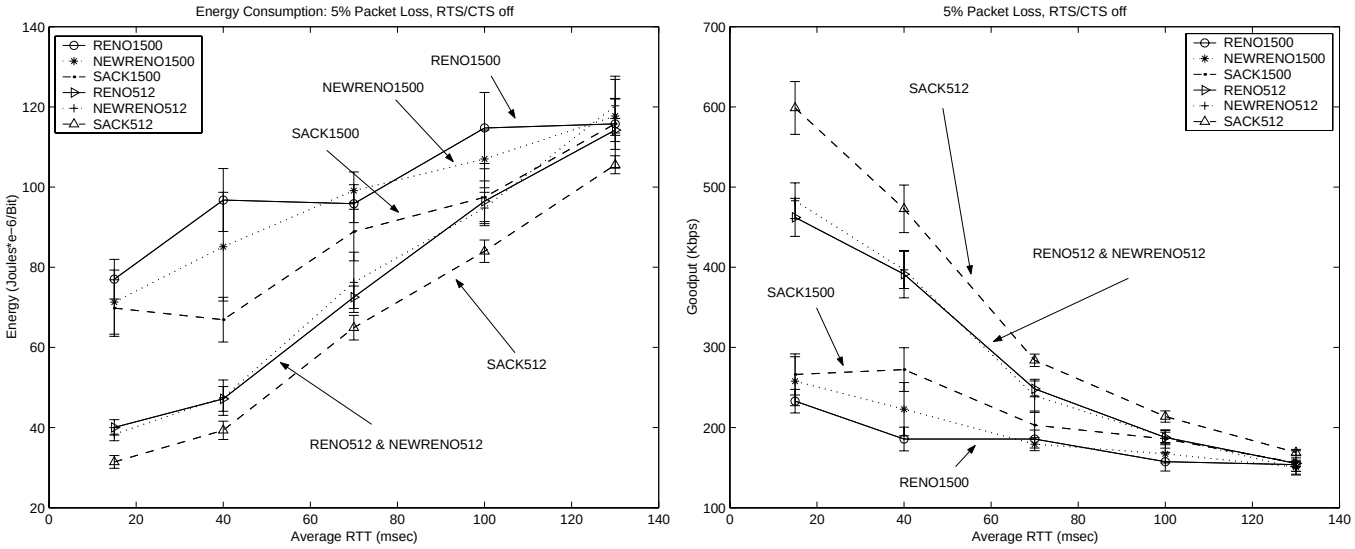

Figure 5: Total Energy E per bit and Goodput for 5% packet loss with no RTS/CTS.

	1% Loss	5% Loss	10% Loss
MTU 512	$E^S \approx E^N \approx E^R$ $\tau_S \approx \tau_N \approx \tau_R$ $E_I^R < E_I^N \approx E_I^S$	$E^S < E^N \approx E^R$ $\tau_S > \tau_N \approx \tau_R$ $E_I^N \approx E_I^S \approx E_I^R$	$E^S < E^N \approx E^R$ $\tau_S > \tau_N > \tau_R$ $E_I^N < E_I^R < E_I^S$
MTU 1500	$E^S < E^N < E^R$ $\tau_S \approx \tau_N > \tau_R$ $E_I^N < E_I^R \approx E_I^S$	$E^S < E^N < E^R$ $\tau_S > \tau_N > \tau_R$ $E_I^N \approx E_I^R < E_I^S$	$E^N < E^R < E^S$ $\tau_N > \tau_R > \tau_S$ $E_I^R < E_I^N \approx E_I^S$
R – Reno, N – Newreno, S – SACK			

Table 4: Summary of total energy (E), goodput (τ), and idealized energy (E_I) data.

(and total energy). When the sender receives three duplicate ACKs (that also contain information about holes in the receiver's buffer), the sender retransmits one segment and then retransmits the segments that corresponded to holes in the receiver's buffer as and when *pipe* is less than CWND. Newreno, on the other hand, sequentially retransmits segments on receipt of partial ACKs. This results in some packets not being retransmitted early enough and we get timeout events. In our experiments, we noted that SACK never had any timeouts for the reorder experiments while both Reno and Newreno had timeout events (Reno more than Newreno).

- The above discussion explains why SACK has a higher goodput and lower total energy than Newreno and Reno. The reason it also has the lowest idealized energy is because (1) there are no timeouts for SACK thus reducing the processing involved, (2) SACK retransmits fewer packets than Reno and Newreno.

7. DISCUSSION

In this paper we examined the relative energy consumption profiles of TCP Reno, Newreno, and SACK. We note the following:

- The total energy consumed (for any protocol) is inversley proportional to the throughput.
- The total energy consumed for SACK is the lowest in almost all cases except for MTU 1500 at a loss rate of 10%. The reason SACK has the lowest energy cost is that it has the highest throughput and the idle energy of our device was high (18W). This high idle energy cost plays a dominating role in the total energy measurements.
- SACK has a poor idealized energy performance due to the fact that SACK implementations require additional data structures and processing. This is the reason that SACK performs poorly for the 10% loss case with 1500 MTU – here the benefits of using SACK do not come into play since the CWND contains few segments. However, the computational overhead is still present and this causes SACK to have a higher total energy consumption.
- In the case of packet reordering, SACK is by far the winner in terms of total energy and idealized energy. This is because SACK has no timeouts for the cases we looked at (both Reno and Newreno have timeouts).
- In the case of bursty losses, SACK has the lowest total energy while Newreno has a slightly lower idealized energy. The reason is again that SACK's computational cost overwhelms the gains due to a higher throughput.

Using the above results we can conclude that when designing a protocol for a mobile device, we must first consider the operating environment (bursty loss or random loss, etc.) and then select the appropriate level of protocol complexity since the computational

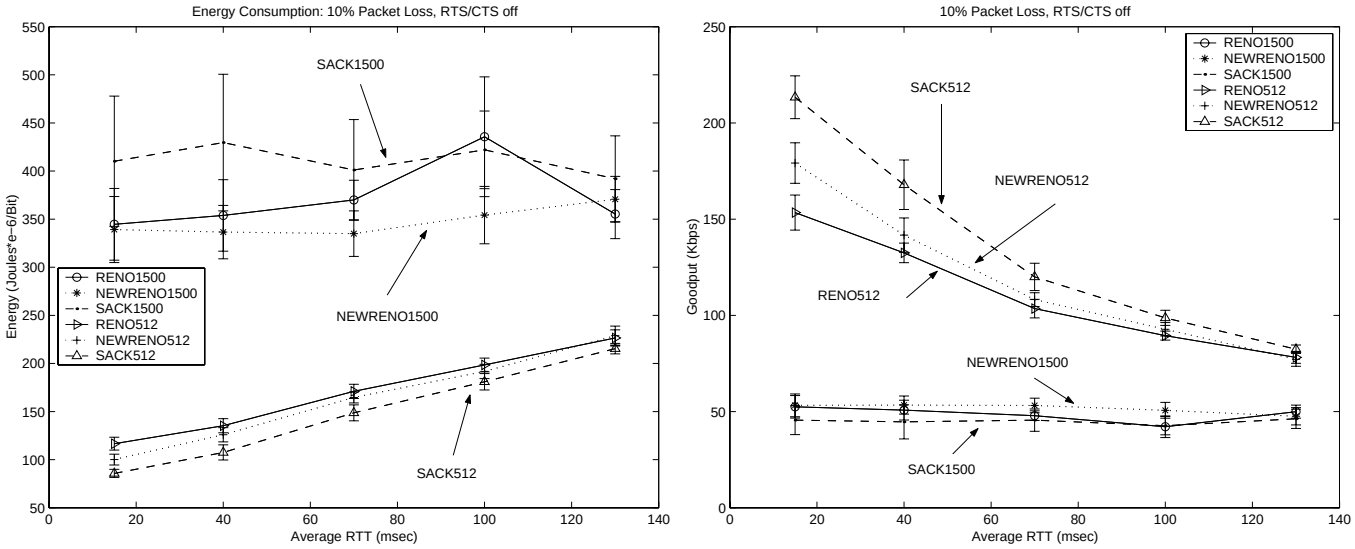

Figure 6: Total Energy E per bit and Goodput for 10% packet loss with no RTS/CTS.

overhead of a protocol can be significant. In our case, if we extrapolate and imagine a device with very low idle power consumption then it is clear that SACK would be a poor choice for most situations.

Acknowledgements

We would like to thank Jim Binkley for his technical support in implementing our testbed, L. Rizzo for technical support on Dummynet, and the FreeBSD community for technical support during our implementation of SACK in FreeBSD 4.3.

8. REFERENCES

[1] NS-2 Network Simulator, http://www.isi.edu/nsnam/ns/ (October 15, 2001).

[2] Simplepower, http://www.cse.psu.edu/ mdl/SimplePower.html (October 15, 2001).

[3] OpenBsd2.9, http://daedalus.cs.berkeley.edu (August, 2001).

[4] Mark Allman, Chris Hayes, Hans Kruse, and Shawn Ostermann, "TCP Performance Over Satellite Links", *In Proceedings of the 5th International Conference on Telecommunication Systems*, March 1997.

[5] J. Bennett, C. Partridge, and N. Shectman, "Packet Reordering is Not Pathological Network Behavior", *IEEE/ACM Transactions on Networking*, December 1999.

[6] H. Balakrishnan, V. Padmanabhan, S. Seshan, and R. Katz, "A Comparison of Mechanisms for Improving TCP Performance over Wireless Links", in *ACM SIGCOMM*, Stanford, CA, Aug. 1996.

[7] R. Bruyeron, B. Hemon, and L. Zhang, "Experimentations with TCP Selective Acknowledgment, *ACM Computer Communications Review*, Vol. 28(2), April 1998.

[8] K. Fall and S. Floyd, "Simulation-based Comparison of Tahoe, Reno, and SACK TCP", *ACM Computer Communications Review*, Vol. 26(3), July 1996, pp. 5 – 21.

[9] Laura Feeny and Martin Nilsson, "Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment", *Proceedings INFOCOM 2001*, Anchorage, Alaska.

[10] Tim Henderson, Randy Katz, "Transport Protocols for Internet-compatible Satellite networks", *IEEE Journal on Selected Areas of Communications*, February, 1999.

[11] G. Holland and N. Vaidya, "Analysis of TCP Performance over Mobile Ad Hoc Networks", *Proceedings ACM Mobicom'99*.

[12] L. Rizzo, "Dummynet: a simple approach to the evaluation of network protocols," *ACM Computer Communication Review, Vol.27,n.1, Jan. 1997.*

[13] L. Rizzo, "Issues in the Implementation of Selective Acknowledgments for TCP", January, 1996, http://www.iet.unipi.it/ luigi/selack.ps

[14] V. Rodoplu and T.H. Meng, "Minimum Energy Mobile Wireless Networks," *IEEE Journal on Selected Areas in Communications, vol. 17, pp. 1333-1344, August 1999.*

[15] V. Tsaoussidis, H. Badr, X. Ge, K. Pentikousis, "Energy/Throughput Tradeoffs of TCP Error Control Strategies," *In Proceedings of the 5th IEEE Symposium on Computers and Communications, France, July 2000.*

[16] W. Richard Stevens, TCP/IP Illustrated, Volume I: The Protocols, Addison Wesley Publishers, 1994.

[17] M. Zorzi, R.R. Rao, "Error Control and Energy Consumption in Communications for Nomadic Computing," *IEEE Transactions on Computers, March 1997.*

[18] M. Zorzi, R.R. Rao, "Is TCP Energy Efficient?," *Proceedings IEEE MoMuC, November 1999.*

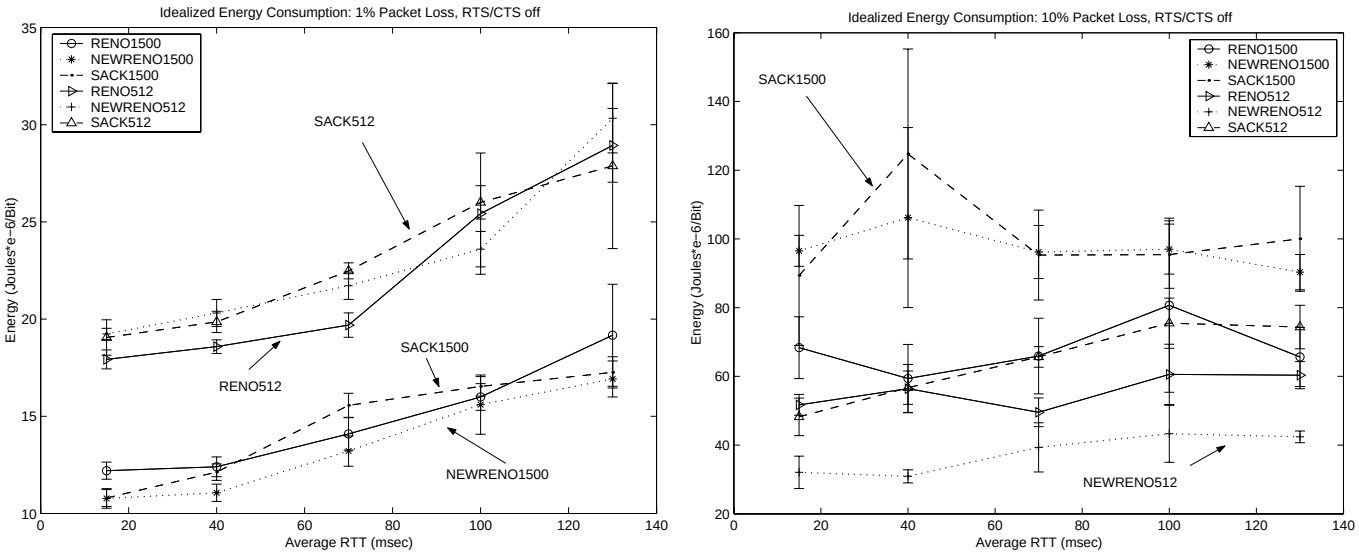

Figure 9: Idealized energy per bit of goodput for 1% and 10% packet loss.

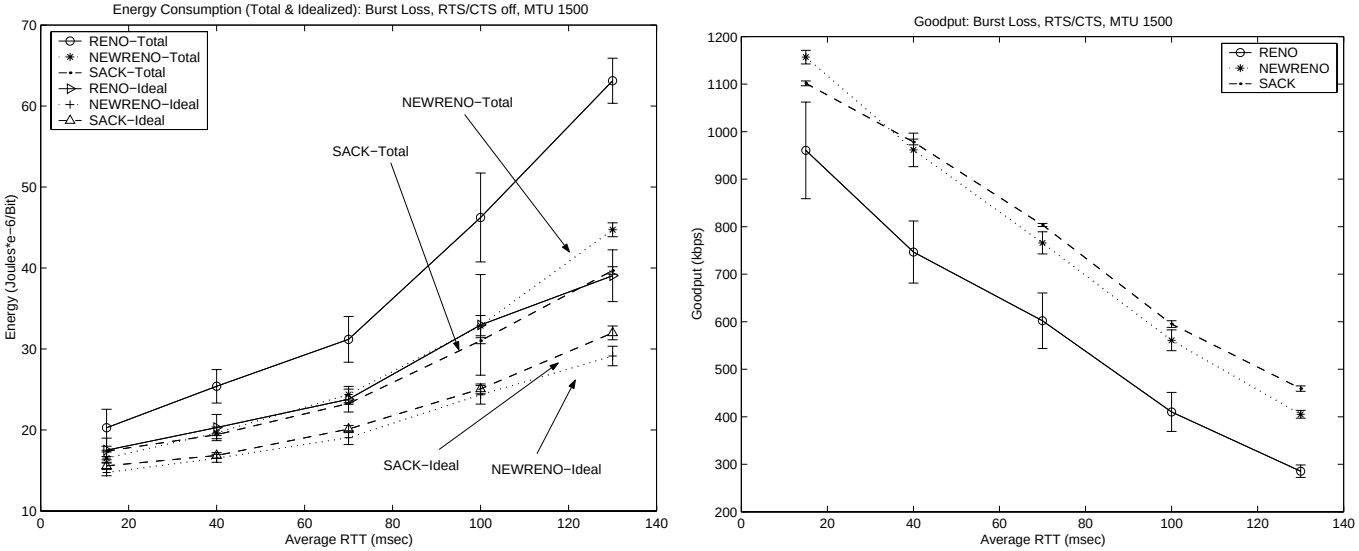

Figure 10: Summary of energy and throughput for bursty loss.

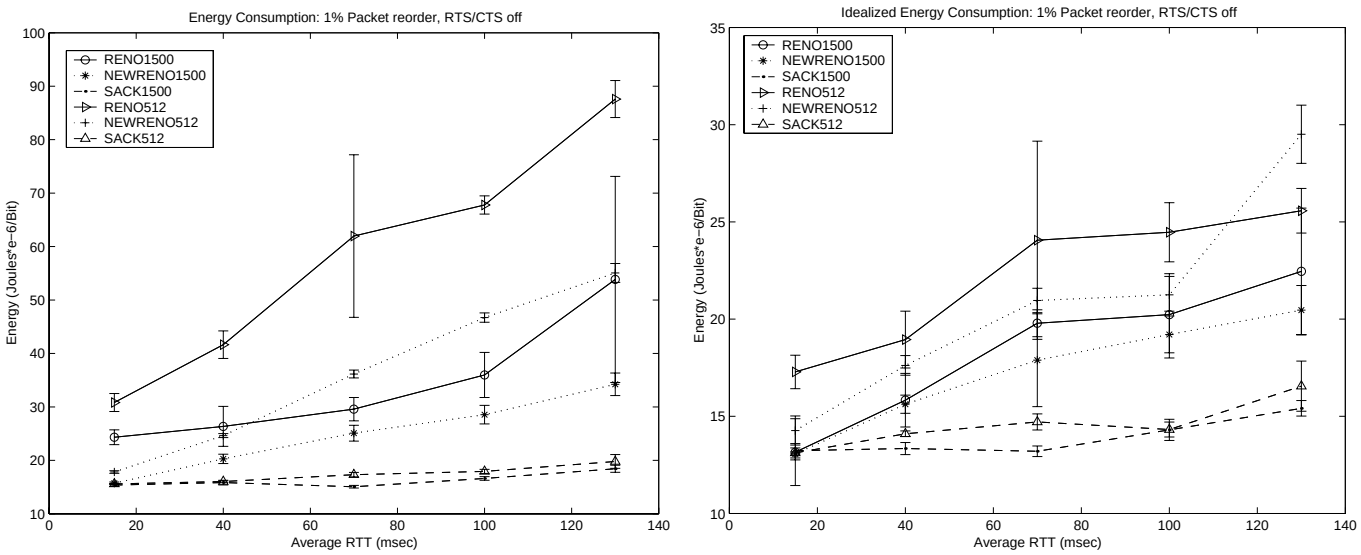

Figure 11: Total energy and idealized energy for a 1% packet reordering case.

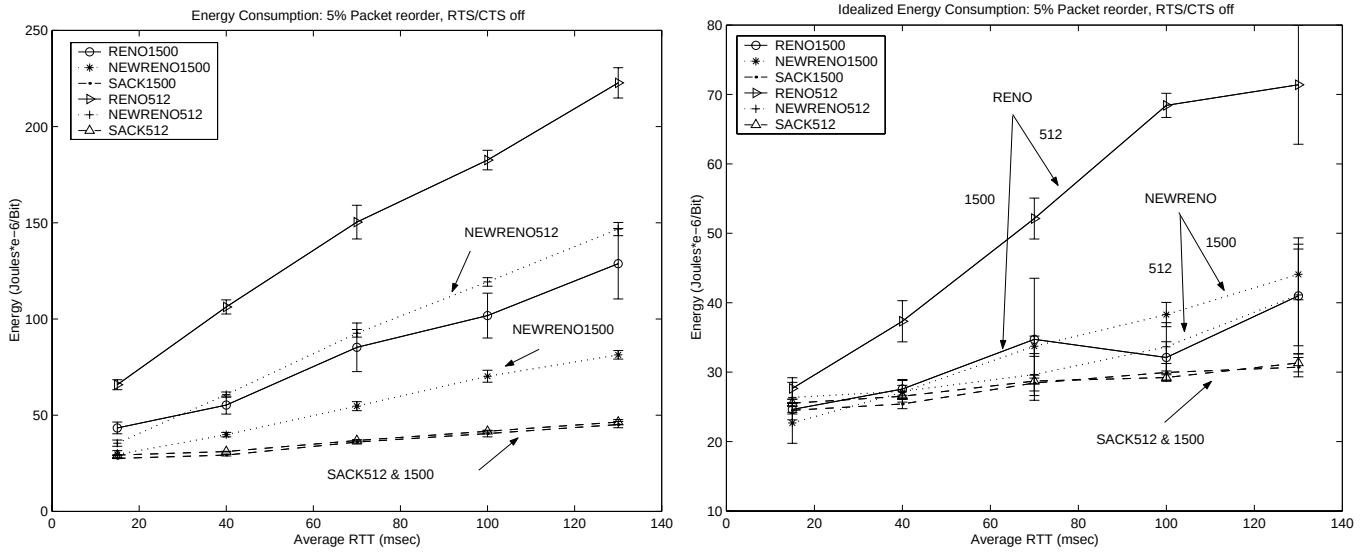

Figure 12: Total energy and idealized energy for a 5% packet reordering case.

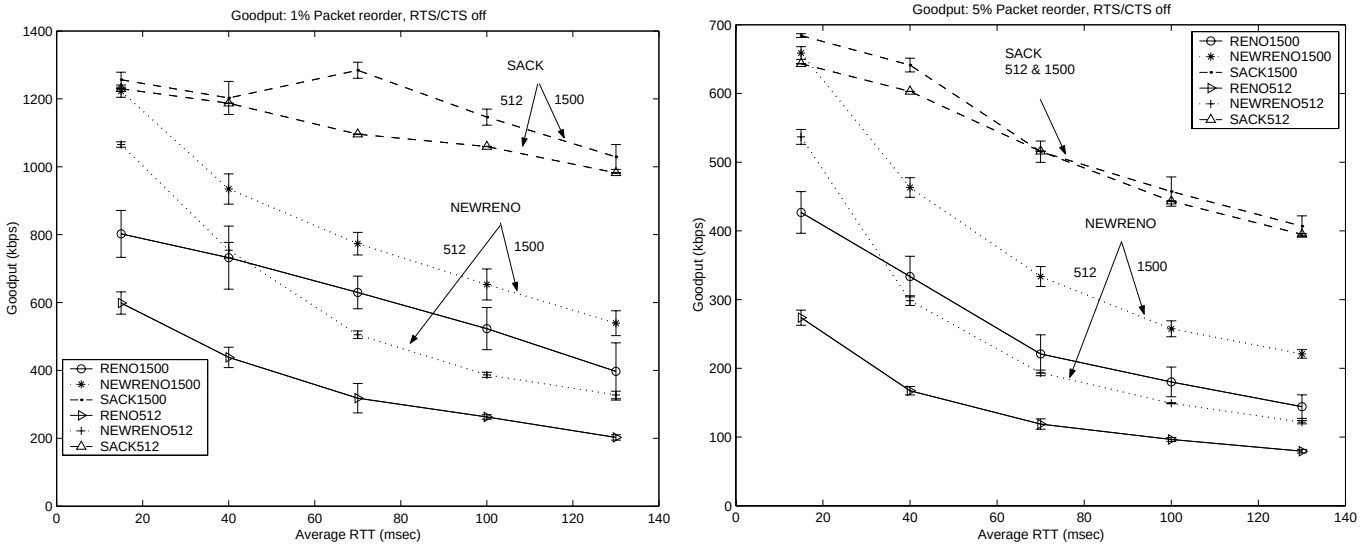

Figure 13: Goodput for the reordering case.