Protocols for Low-Power

Srinivasa Aditya Akella, Rajesh Krishna Balan, Nikhil Bansal

Abstract

We present ideas for Multi-modal protocols. These are protocols which alter/determine their operating mode depending on the environment. We identify the challenges involved in coming up with a suitable architecture for a protocol stack supporting multiple modes of operation. We present a framework for a Multi-modal network stack.

To exemplify the issues involved, we consider the example of a Multi-modal transport layer. We discuss the modifications that can be done at the transport layer (TCP) for end-systems to conserve power. These modifications each map onto a different mode of operation. We identify scenarios where our algorithms perform well, based on simulations in ns2 as well as transfers over the Internet. We show that it is possible to save power by using various operating modes at the transport layer.

1 Introduction

The Internet today is a huge collection of various different kinds of hosts which operate in a variety of environments. For example, the hosts range from small sensor networks, hand held PDA's to more powerful desktops and mainframe machines. The scenarios in which these devices work are usually very different. For example a PDA would probably be operating on wireless links which are bandwidth constrained and quite lossy. Sensors are likely to be computationally constrained and operate in a bandwidth constrained environment. Another important concern with these devices is that they are power constrained.

The current network stack design does not take the above factors into account. In particular all the protocols work in the same mode regardless of the environment in which they are operating. While this is an important first step in implementing any functional system, more needs to be done when performance is an issue.

We need a multi-modal network stack as there are quite a few situations where different modes of operation would be beneficial. These are:

1. Power limited: This is the case where one of more devices are power constrained. In this case, the TCP algorithm can be modified to be power aware. For example, an asymmetric version [9] can be implemented which transfers most of the functionality to the other host.

2. Bandwidth limited: This applies in cases where devices are either bandwidth limited (e.g., they are connecting via infrared) or they want to minimize their network footprint (soldiers in a battlefield etc.). The stack can be optimized to reduce the number of transmissions to the bare minimum and first compress the date before transmitting it.

3. Memory and/or computationally constrained: Devices like sensors fall into this category. They will need a stack that uses as little of the CPU and/or memory as possible. I.e., the stack will not keep a lot of state or buffer too many packets in memory etc.

Our goal is to come up with a framework for multi-modal protocols. These are protocols which work differently in different environments with the goal of optimizing the usage of scare resources. There are various challenges to doing this though. For instance, the various scenarios and the constraints need to be identified. Moreover, we need to quantify what is meant by being resource constrained. For instance we may need to choose who is the most constrained among the various hosts. Also, there is likely to be a trade-off involved among the usage of resources under different modes. In order to meaningfully decide on the mode of operation, there is a need for metrics to evaluate these trade offs.

To illustrate these issues, we consider a particular question. *What are the various ways in which TCP can be redesigned so that it can operate to conserve power?* For example, it is possible for TCP to require fewer transmissions from power constrained nodes? Can we reduce the transmission of redun-

dant information? Can we use additional transmissions from power rich nodes to reduce transmission from power poor nodes.

This paper is organized as follows. Section 2 lists the relevant previous work. In Section 3, we consider the various challenges and issues related to multi-modal protocols. In Section 4, we focus on the particular example of what can be done at the network transport layer to reduce the power consumption of a device. We first present the theoretical basis for our various approaches and then illustrate our results via implementation and simulation. We evaluate these results in Section 6. Finally, Section 7 concludes with the future work and conclusions.

2 Related Work

In the last few years, there has been a large amount of work in developing low power networking solutions for various environments. Papers like [10] outline a number of broad principles which can be used to design low power systems. Zorzi and Rao analysed the energy efficiency of TCP [23] and concluded that it was dependent on the particular flavour of TCP used as well as on the amount of error correlation. This seems to suggest that the power efficiency of TCP is very dependent on the environment it is used in. The environment most likely to be used by small power constrained devices is the wireless environment. Stemm and Katz have characterized the power used by traditional wireless cards in [21] where they show that there is scope for substancial power savings in the transmission of packets over a wireless medium.

A number of schemes have been developed to improve the energy efficiency of TCP and/or other network protocols [3, 2, 8, 11, 12, 13, 14, 15, 5, 18, 4, 22]. Some of the more interesting mechanisms suggested by these schemes are

- Changing the granularity of TCP timers to reduce the number of transmissions.
- Making one side of the TCP connection "dumb" and stateless. Hence the dumb side will not have to waste resources keeping various states around. It will also only transmit packets when it receives an explicit request from the other end host.
- Reduce the number of pre-emptive transmissions or re-transmissions. i.e., only transmit a packet when you are sure that it needs to be transmitted. Do not transmit a packet just because it makes things more efficient or because it might have been lost (by default, TCP uses timers to determine when to retransmit packets. Usually, packets which are retransmitted are actually required by the other end host as the packet has been lost in transit. However, this may not always be the case)
- using base stations to buffer packets so that end hosts can sleep for long periods of time. These end hosts wake up every now and then and retrieve any packets addressed to them from the base stations.

There have also been initiatives to optimize TCP's peformance for metrics other than power. Schemes like WebTP [9], I-TCP [1] and NETBLT [6] were designed to improve the performance of TCP with relation to latency, loss recovery in wireless mediums and throughput for bulk transfers respectively.

Finally, there have been been projects like Odyssey [17] and Spectra [7] which attempt to provide adaptive application behaviour (conserving power is one of the behaviours) by providing additional functionality to the kernel. These systems do not modify the network stack layer of the kernel and work at higher layers.

3 Challenges

Designing a network stack supporting multiple modes of operation is not trivial. There are many issues that need to be taken into account to ensure that the system is stable, comparable in performance with the existing implementations and which actually helps realize the basic goal of a multi-modal stack.

In this section, we present some of the questions which are integral to the design of a viable architecture for multi-modal protocols. They are:

(i) How does an end system identify its operating mode?

(ii) How should the end system convey its desire to operate in a certain mode to its peer? How would the two, then, proceed to a common mode of operation? In other words, what would be a good negotiation scheme?

(iii) How would this solution for the unicast case apply to the multicast case? What would be a

Mode	Wavelan	Bluetooth
Idle	280	25
Sleep	16	0.1
Receive	285	52
Transmit	335	44

Table 1: Current drawn in mA for various network interface cards

good generic solution that could apply for both cases?

(iv) How does the end system keep track of the aptness of the choice it made with regard to the operating mode? What would be a good method for the end system to identify inefficiencies in it's operation mode and switch to a different mode?

(v) What would be a good metric to measure performance? How good is the performance improvement?

These questions are answered in subsequent sections as far as possible. Some of these questions have yet to be properly answered and are discussed further under the future work section.

4 Low Power TCP

4.1 Theory

Networks cards operate in three modes: sleep, idle and active. In the active mode, the card is either transmitting or receiving data. In the idle state, the card is on, but isn't receiving or sending data. In the sleep state, the card is powered off. Typical number for power consumption in each mode are shown in Table1 for Bluetooth and IEEE 802.11 [12, 20]. As can be seen, there is significant difference in power consumption in the various modes. These numbers bear significance for battery-dependent devices like PDAs, laptops and sensors in the extreme case. A good way to save power in these scenarios would be to keep the card in the active state only when necessary and minimize idle power dissipation.

Power can be saved in one of many ways. They are:

1. **Transferring functionality** An end node may save power, for example, by transferring functionality like timers and/or state to the other end. For example, the receiving end could maintain a retransmission timer, and send a NACK to the sender on timing out, instead of the sender having to maintain it.

2. **Opportunistic Sleeping** A receiver might be able to make the sender transmit data in a predictable manner, and sleep during 'dry' epochs, when there are no packets to receive. Alternately, the sender could send data in bursts and sleep for time on the order of an RTT.

3. **Proxy Support** A proxy allows a host to sleep for specific periods of time and then query the proxy to see if any packets are outstanding for it. As such, the host can save power by sleeping for a period of time and then waking up periodically to check with the proxy if there are any packets outstanding for it. This is the approach used by the IEEE 802.11 standard (used by Wavelan cards) which allow compliant cards to sleep for a period of time (Wavelan cards typically sleep for 100ms) and then wake up and poll the proxy for packets.

4. **Modified Data Transfers** A data transfer could be made concise by avoiding transmission of redundant data, or compressing data before a transfer. This could result in fewer transmissions and receptions.

In this report, we describe solutions for the case where there is no proxy support. Two cases arise here: (ii) both end hosts are modified and (ii) only one end host is modified. In the first case, finer grain synchronization can be put to good use. In particular, the 2 end hosts can negotiate a particular traffic pattern which they promise to use. Each end host can then sleep in the lull periods of this traffic pattern. Modifying both end hosts would definitely result in better results than just modifying one host as the traffic pattern can now be predictable (barring sudden changes in the congestion state of the network) and does not have to be guessed or artificially induced.

In this report, we consider case (ii) in great detail. Specifically, We now consider the case where the receiver of data wants to save power, without requiring sender-side modifications. The essential trick is to use TCP to make the sender transmit in a predictable manner. For case (i), we show no results, as throughout the bulk of the report we assume receiver side modifications only.

4.1.1 Delayed Acknowledgements

Vanilla TCP sends ACKs for every received packet. This could mean a lot of overhead. The typical solution to this problem (*delayed ACKs*) would be to send a cumulative ACK for a certain number of packets, called the delayed ACK threshold (this is set to 2 in standard TCP) but send an ACK immediately if no packet arrives before a certain timeout (this timeout is set to 200ms in TCP) or if a packet arrives out of order.

Delayed ACK-ing results in a natural bunching of packets (into pairs, in the typical case). This causes the sender to send packets in bursts of the order of the delack threshold. This suggests that a natural way to gain sleep time would be to cause the sender to transmit in long bursts, by choosing a large delack threshold. However, this might not be a desirable solution if the connection is bandwidth constrained, as the bottleneck would cause packets to be spread out. Also, sending in larger bursts, might cause queue overflows at intermediate routers, leading to a higher loss rate and reduced throughput.

4.1.2 Delack Timer

Choosing an appropriate value for the delayed ACK timer is important for various reasons. A large value of the timeout, when using a large delayed ACK threshold might prove disastrous mainly because in such cases, the receiver will typically have to wait until timeout to send an ACK, as it might take too long for the entire burst of packets to arrive. So, we have choose the timeout very conservatively. Also, a large delayed ACK threshold interacts poorly with the slow start phase of TCP causing the receiver to send ACKs every $RTT + timeout$ seconds, leading to poor ACK clocking, or in the worst case, to the sender timing out and exiting slow start. A solution to this would be to ACK more frequently (one every one or two packets during slow start) and less frequently later on.

An adaptive delayed ACK timer might also aid in increased bunching of packets. The essential idea is to delay ACKs for different pairs of packets by different amounts (delay the ACKs for first few pairs by a greater amount than the last few) causing the pairs to be lumped up into bigger chunks. This could prove particular useful in window constrained transfers using a delayed ACK threshold of 2.

4.1.3 Slow-start: Issues

Slow-start Prediction One nice aspect of slow start, is that packets arrive in bursts separated by time intervals of the order of the round trip time. This is not necessarily true after slow start terminates, unless the transfer is not window constrained or bursty due to reasons pointed out above. It might seem that the receiver would have to explicitly predict when the sender exits slow start, so that he could begin to adjust his sleep timers to a lower granularity. As will be seen in the section about setting sleep intervals, this is not completely necessary, given our algorithm for sleeping. However, for a different and a possibly more efficient algorithm for setting sleep times, it might be a significant addition.

For such an explicit prediction of slow-start, we have a reasonably accurate algorithm. The receiver keeps a count of the packets it receives during each round trip time, starting from the very first *data* packet onwards. Let D_n denote the number of packets received in the n^{th} RTT. Irrespective of the use of delayed acknowledgements, D_n should show an exponential increase, roughly as a function of n. Typically, the sender exits slow start, when its congestion window is at least as large of $ssthresh$, the slow start threshold. Starting from this time on, the sender increases his congestion window linearly every RTT. This would mean D_n either stops increasing exponentially, or would start decreasing (upon entering loss recovery). The receiver could use this change in D_n to predict the end of slow-start.

Short Transfers More than 70% of traffic on the Internet is HTTP web transfer based. It is reasonable to expect that power-constrained devices like PDAs would also be involved in a significant amount of HTTP transactions. Typical size of such a transfer is about 8KB, which is about 7-8 TCP packets. Consequently, if these transfers were TCP based, most of them would be in slow start throughout the duration of the transfer. The receiver could sleep between bursts as suggested above. An alternative to doing so would be to get the sender to exit slow start and transmit all packets in one large burst. To do this, the receiver will have to wait for the first data packet, D_1 (in fact, the receiver might not have to wait for the first packet to arrive). Upon receiving D_1, the receiver could then acknowledge each byte in it and send these acknowledgements in a burst. This would cause the sender to increase its congestion window and possibly send a larger burst in the next RTT. The sender could also have exited

slow start, if the acknowledgements we enough in number to increase the sender congestion window beyond *ssthresh*.

4.1.4 Send/Receive side Window Constraints

In every RTT, a TCP sender would transmit $max(cwnd, window)$ of data, where *window* is the receiver's advertised window and *cwnd* is sender's congestion window (both in bytes). If the receiver advertises a small window size, the sender is always forced to send atmost *window* bytes each time. Thus by advertising a sufficiently small window size, the receiver could control the transfers made by the sender, increasing the predictability of the transfer.

If the bandwidth available between the sender and the receiver is large enough, this could lead to data being received in bursts with sufficient separation – another opportunity to sleep. However, when bandwidth is insufficient packets would be spread out leading to reduced inter-burst gaps. Using a smaller window size could be a solution here, but that would imply vastly reduced throughput, which *might* be undesirable.

4.1.5 Loss Notification

How should the sending side respond to the receiving end dropping packets while sleeping? If no care is taken to address this issue, the receiver would continue to send duplicate acknowledgements forcing the sender to enter fast retransmit/recovery, inferring a congestion loss, and reducing its congestion window. There is no way for the sender to infer the nature of a loss (congestion related or drop on sleep) without explicit signalling from the receiver.

Hence, there should be some means for the receiver to let the sender know that the loss was not a congestion loss, but was due to packet arriving on a sleeping network interface card. We propose to use a snoop style solution with the end system setting ELN bits in its acknowledgements. Here is how the receiving end, R would have to work: Let A denote the interval of time that the receiver was last awake for. Let N be the last in order sequence number received by R during A. Suppose that at the end of A, R sleeps for an amount of time S. Let M be the sequence number of the first packet received after S. If $M = N + 1$, the receiver proceeds normally. Otherwise, it sends duplicate acknowledgements for N with the ELN-bit set to 1. This would cause the sender to retransmit just $N + 1$ without reducing its congestion window.

Notice that, it is possible that, at the end of S, a packet with sequence number N', $N < N' < M$, that would have otherwise reached R as the first packet during the awake period following S, was lost due to congestion. Even this loss, would invoke an ELN kind of a behavior from the receiver, causing the sender to not reduce its congestion window. This, we believe, is not much of a problem,as subsequent congestion losses would anyway cause the sender to reduce its transmission rate.

4.2 Implementation

We first consider how the sequence plots for some typical web transfers look like. We can partition the types of behaviors into two different categories:

1. **Window constrained transfers** The rate of transfer, in this case, is controlled by the size of receiver's advertised window (usually limited to 8-10KB)

2. **Bandwidth constrained transfers** The receiver, in this case, advertises a sufficiently large window. So the transfer rate is governed by the bandwidth of the bottleneck link.

A note on the Metric Choosing the right metric for measuring the performance improvement was tricky. The metric should ideally capture the tradeoff between throughput loss and power gain. This issue is discussed in greater detail in the Future Work section. For our experiments, we chose to use "energy per packet" as a metric for performance measurement. Energy per packet is equal to the ratio of the total energy spent during the transfer to the total number of packets received at the receiver. All the results presented in this number are for Wavelans.

The figures below show the sequence plots for each of these type of connections. The transfers were done on the Internet. The window constrained connection was between CMU and Singapore. The receive window was constrained to be 8KB for this case (which is much less than the bandwidth-delay product). For the bandwidth constrained case, the connection was between CMU and MIT. The windows on both ends were not constrained.

Observe in Figure 1, that for the window constrained case there are gaps after every RTT interval. This occurs because the sender cannot send more than a window's worth of packets during a single RTT. We now show how the scheme proposed in the previous section performs in this scenario.

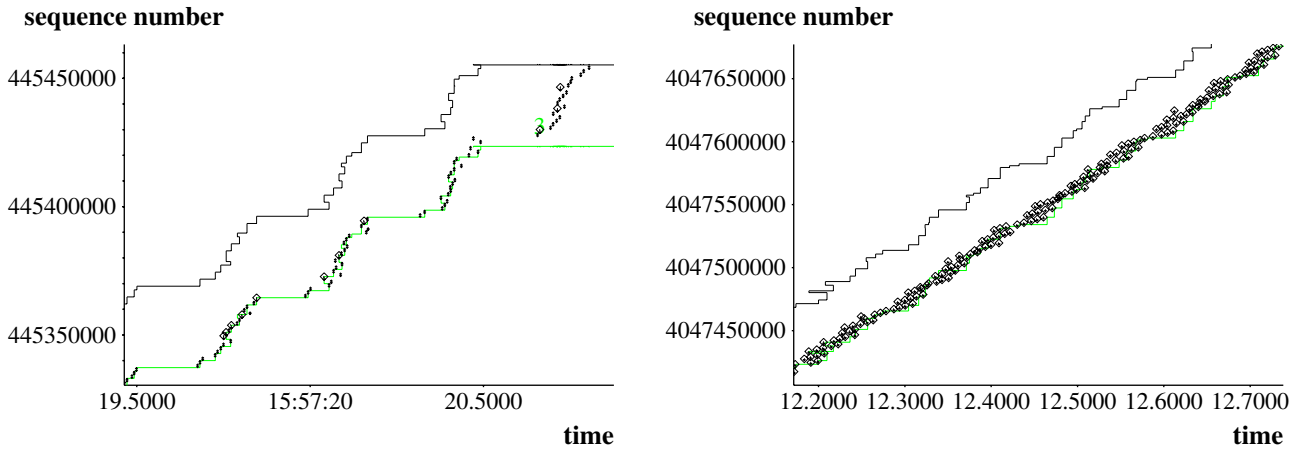

Figure 1: The figure on the left shows the sequence plot for the window constrained connection. the figure on the right shows the sequence plot for the bandwidth constrained connection.

As the sequence plot in Figure 1 shows, there are not many predictable gaps in a bandwidth limited transfer. Most gaps, if any are in the slow start phase which lasts for about a few seconds only. Even bunching does not help increase the spacing.

To test the efficacy of our arguments for opportunistic sleeping, we implemented Multi-modal TCP in ns2. Delayed ACK threshold (number of segments received per ACK sent) was made variable, and we ran a few simulations over the following topology.

We implemented the following algorithm for sleep interval prediction in ns2:

1. Suppose that the receiver just woke up (moved from sleep to idle mode). Upon receiving the first packet after waking up, say at time t_1, the receiver starts a timer, called the sleep timer, for $f * RTT$ seconds, where RTT is his current estimate of RTT and f is a fraction of RTT (usually between 0.7 and 0.9). The use of f would become clearer later on. Notice that, this requires the receiver to make an online estimate of the round trip time.

2. Thereafter, for each received packet, the receiver also sets a timer, called the awake timer, for *delta* seconds, starting at the reception time of the packet. If a packet arrives before the awake timer goes off, the timer is rescheduled for *delta* seconds.

3. If, say, the timer goes off at time t_{off} and no packet arrived before t_{off}, the receiver decides to sleep between t_{off} and $(t_1 + f * RTT)$, if there is sufficient time left in the interval ($\geq$ $20ms$)and drops any packet that arrives in this interval.

4. The receiver wakes up when the sleep timer goes off.

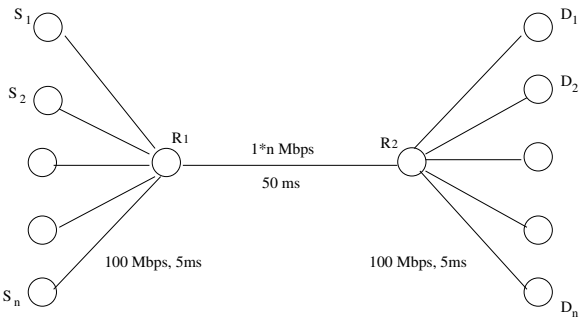

Figure 2: Topology for the NS simulations

4.2.1 Window Constrained Transfers

The results obtained for the single flow case are shown in Table 2, while those for the multiple flow case are shown in Table 3.

In either of the two tables, *delta* refers to the time that the receiver waits (in active mode) after receiving a packet, for the next packet to arrive. After waiting for *delta* seconds, the receiver enter sleep mode, if possible. *Fraction* refers to the fraction of the estimated RTT, described in the sleep algorithm.

The first two entries in either table correspond to the default case where no sleep algorithm is implemented. This will be our base case for comparison against other entries, where sleep is employed.

For the window constrained case, our windows were limited to a size of 10 packets. Observe when the delayed ACK threshold is 2, the savings in power per packet are negligible, if not worse. The

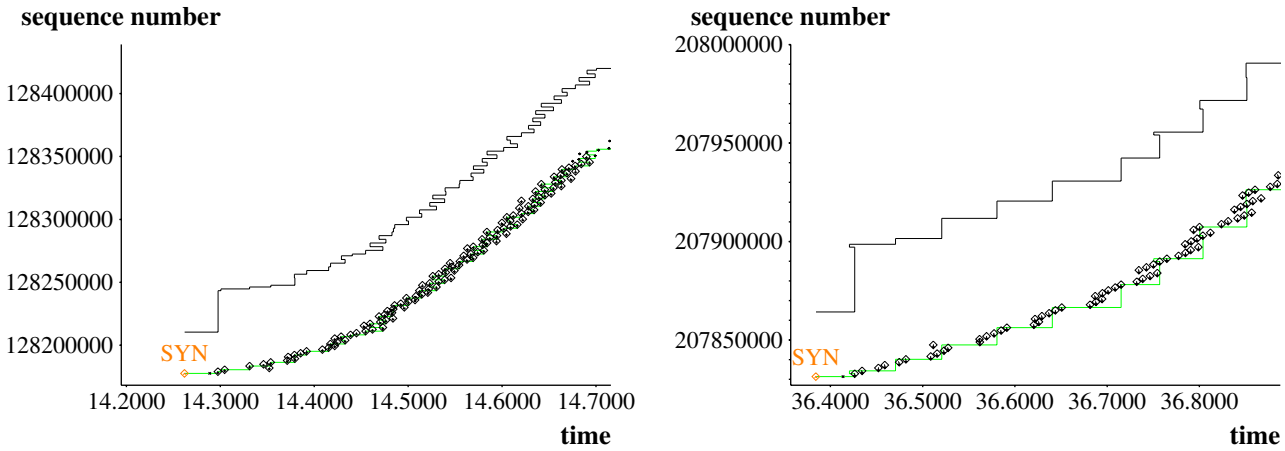

Figure 3: The figure on the left shows the sequence plot for the window constrained connection with a small delack threshold (2). The figure on the right shows the sequence plot for a larger threshold (8).

basic observation is that, a threshold of 2 causes the 10 packets to be bunched into pairs with a arbitrary amount of spacing as shown in Figure 3. (These gaps are more predictable for larger delack thresholds). Specifically, observe that for the case when delta is 0.01s, there is a significant drop in throughput. This occurs because, the receiver enters sleep way too often, causing many drops on its interface card. For a larger value of delta (0.06s), the throughput is comparable, but power savings are negligible because, the receiver never enters sleep state for the gaps available to sleep are never more than 60ms, as pairs of packets arrive spaced apart.

As we start increasing the delayed ACK threshold, we observe that the power per packet improves substantially, even at low values of delta. In particular, when the threshold is set to 4, the throughput drops due to packet drops, but there is a gain in sleep time due to more bunching and greater spacing. This results in a lowered per packet power. Finally, the results are most impressive, when the delayed ACK threshold is close to the advertised window size. The throughput is comparable to the base case , and there is significant gain in power too. Observe that delta can be set to really small values as essentially due to larger amounts of bunching. All the packets within a bunch arrive immediately, and there is a substantial gap between bunches. This leads to increased opportunities for sleeping.

For the multiple flow case, the results are more interesting. Observe that in this case, small values for delta are bad, even for high delayed ACK thresholds. This happens because, even when the sender sends packets in bursts, they might be spread apart at the receiver due to interaction with other flows at intermediate routers. Besides, at lower delayed ACK thresholds, the bunches of packets tend to be more spread out than they are at the higher values. This implies that at lower values, the receiver can only spend a smaller fraction of RTT sleeping, than in with the higher values of the threshold, as shown in the table.

4.2.2 Bandwidth Constrained Transfers

Table 4 shows the power gains, if any, for the bandwidth constrained case, as a function of the delayed ACK threshold. As above, the first two entries are the base case against which we make our comparisons. Observe that for a threshold value of 2, setting delta to a small value decrease the throughput substantially and the power gains are negative! This is due to increased drops. Setting delta to a higher value, gives very marginal gains, because, the total time available to sleep is about 1% of the transfer length. The trend looks similar even at higher values of the threshold and the correspond values of delta.

We also used the program Sting [19] to check how the traffic is received at the receiver even if we force the sender to send the data is bursts. We modify Sting to send the ACKs in bursts[1]. We observe that even if we send, say 10 ACKs in a bunch, the packets corresponding to the ACKs arrive quite spread apart at the receiver. This supports our observation in NS that increasing delayed ACK threshold does not help significantly in a bandwidth constrained scenario.

For the multiple flow case, we find that trends look identical to that of the single flow case. In particular, if delta is kept high, there is very little

[1]Sting forces the sender to sender a packet on every ACK

Delayed ACK threshold	delta	Fraction of RTT	Energy per packet (mJ)	Throughput
1	-	-	17.48	0.66
2	-	-	18.25	0.63
2	0.01	1.0	21.61	0.19
2	0.06	1.0	18.28	0.64
4	0.01	1.0	15.39	0.26
8	0.01	1.0	12.30	0.53
8	0.01	1.1	10.71	0.53

Table 2: Single Flow Case, Window Constrained

Delayed ACK threshold	delta	Fraction of RTT	Energy per packet (mJ)	Throughput
1	-	-	26.96	0.43
2	-	-	26.56	0.43
4	0.01	0.70	47.79	0.13
4	0.09	0.70	29.66	0.37
4	0.09	0.80	45.18	0.24
6	0.09	0.70	30.88	0.35
8	0.09	0.75	31.35	0.34
8	0.10	0.80	29.53	0.38
10	0.08	1.00	24.69	0.40
10	0.07	1.00	23.37	0.40

Table 3: Multiple Flow Case, Window Constrained

Delayed ACK threshold	delta	Fraction of RTT	Energy per packet (mJ)	Throughput
1	-	-	12.39	0.93
2	-	-	12.07	0.95
2	0.01	0.7	36.24	0.14
2	0.05	0.8	12.03	0.96
4	0.01	0.7	24.16	0.23
4	0.04	1.0	11.96	0.97
6	0.01	0.7	22.75	0.33
6	0.05	0.7	12.18	0.95
8	0.01	0.7	20.96	0.43
8	0.05	0.7	24.79	0.45
8	0.10	0.7	13.05	0.89
8	0.10	0.9	12.44	0.94

Table 4: Single Flow Case, Bandwidth Constrained

gain (if at all) in power, since the receiver never goes into the sleep state. If we try to be aggressive about sleeping, we observe that the throughput goes down substantially.

In figure 2, there are almost no gaps in the sequence plots (except possibly during the slow-start). Moreover the packets arrive at regular intervals (ACK clocking has begun). One way to create a gap would be to artificially introduce gaps (by possibly, controlling the way ACK are sent back, either by increasing the delayed ack timer value or the delayed ack threshold). However it can be seen that this will not be of any help, since the duration of the connection will be increased and the amount for client the network card at the client has to be active will not go down.

4.2.3 Overhead

The packet overhead of the various low power TCP modifications is negligible. Some additional packets are introduced if negotiation is done between the 2 end hosts. However, these are just a few packets at most and do not add any kind of substantial overhead. The rest of the algorithms do not introduce any additional packets into the network.

As far as computational overhead goes, some of the algorithms require some amount of computational work (predicting traffic patterns etc.). However, this computational overhead is minimal and hence does not introduce any form of substantial overhead into the system.

4.3 Kernel

Some aspects of this work has been implemented into the linux 2.4.2 kernel. We have implemented support for changing the number of packets to wait for before sending a delayed ACK into the 2.4.2 kernel. By default, linux will send a delayed ACK after receiving 2 data packets.

The modification to the kernel adds an additional proc entry which allows us to change this value.

The proc entry is /proc/sys/net/ipv4/tcp_ack_packet and changing this value determines the number of packets that the linux kernel will wait for before sending an ACK. However, it must be noted that the kernel will only wait a specific amount of time for these packets before it will automatically send an ACK. Normally the period of time the kernel will wait is at least 200ms and at most the RTO of the connection. To allow changes to these values, 2 additional proc entries have been added to the kernel. They are

- /proc/sys/net/ipv4/tcp_ack_timer_offset_high This allows us to add a constant offset to the 200ms lower limit for the ACK timer.
- /proc/sys/net/ipv4/tcp_ack_timer_offset_low This allows us to add a constant offset to the RTO upper limit for the ACK timer.

5 Discussion

5.1 Multi-modal Stack

5.1.1 Description

The ultimate goal of this project is to design a multi-modal network stack that automatically uses the right power saving algorithms depending on the current situation. Such a stack will require a negotiation protocol through which end systems can decide what set of power saving algorithms they will be using for a particular connection.

In what follows, we give a high level picture of the basic operation of the system assuming that both ends of a connection are multi-modal enabled.

The network stack at each enabled end system stores, what we call, a set of 'constraints'. These constraints, as the name suggests, are resources that the end node would want to optimize, when working in a particular operating mode. Constraints could include, power, computational ability, memory, bandwidth or network footprint and latency.

We assume that the operating system has some means by which it would convey all its 'desires' to the transport layer. At a high level, these 'desires' are directives indicating the operating system's operating intent, like low power, minimal network footprint and such.

At the transport layer, each directive maps into a set of capabilities. Each capability represents one possible way in which the directive (that maps onto it) can be realized. For example, for the 'low power' directive, the corresponding capabilities would be encodings of the following: delayed ACKs, delayed sends, opportunistic sleeping, no timers, no state and such. This handing down of directives from the OS which are then mapped onto capabilities is assumed to happen at each end system.

Coming up with an exhaustive and representative list of constraints and the corresponding capabilities along with an efficient way of storing them is not trivial. Also interesting is the question of

how these capabilities would map onto operational modes and how these various modes would co-exist. These issues are part of our future work and are not addressed in this report.

5.1.2 Negotiation

Given a set of directives from the OS, the 2 end systems will then proceed to negotiate a common set of power saving algorithms with each other.

The negotiation protocol will be an extension of the regular TCP 3-way handshake. Each specific power saving algorithm will be negotiated as TCP options in the normal TCP SYN negotiation. To ensure that both end systems are agreeable to the various algorithms being negotiated, a 4-way handshake will be used (i.e., an additional ACK packet will be required before the handshake is complete). The handshake is as follows.

1. A sends a SYN message to the B. This SYN message will contain all the power saving algorithms that A wants to use. These will be represented as individual options in the SYN message. This is similar to the way current TCP options like SACK, time-stamps, large windows etc. are negotiated.

2. B sends a SYN-ACK message to A with all the power saving algorithms that B wants to use. These options can be the same algorithms that A wanted to use or it could be different options.

3. A sends an ACK message to B containing the common subset of the algorithms that A and B wanted to support. (this could also be just the algorithms that A wants to use without any consideration for what B wants to use. It is for this reason that B is required the acknowledgement the choices made by A and confirm that it is willing to accept the choices).

4. B sends an ACK back to A acknowledging that it is willing to support the algorithms listed in the previous packet).

As mentioned earlier, the standard 3-way handshake was extended to a 4-way handshake to allow B to confirm that it is willing to accept the choices made by A. If B is not allowed this opportunity to confirm A's choices, it may lose out if A decides to push most of the burden to B.

5.1.3 Complexity

There are a number of complex issues involved in this negotiation however.

Firstly, how are A and B going to converge to a commonly acceptable subset of algorithms. It is quite conceivable that A and B would have conflicting sets of algorithms. There are certain capabilities, like timers and state, for which at least one of the two end systems has to take responsibility. Resolving such conflicts, is what makes negotiation non-trivial. The question then becomes, in cases where a commonly acceptable subset of algorithms cannot be agreed upon, should the 2 end systems

(i) Close the connection and report its inability to negotiate a common subset of algorithms to the higher layers of the network stack?

(ii) Default to a standard TCP connection with no additional power saving algorithms?

(iii) Negotiate a connection using the algorithms desired by A?

(iv) Negotiate a connection using the algorithms desired by B?

Each of these options has its advantages and disadvantages. Doing (i) punts the problem to the higher layers which would probably have the most amount of information with which to resolve the conflict. However, it fails to establish any sort of connection and is thus probably not the best solution

(ii) is the most neutral of the solutions and leaves the connection in the default open situation it would have been in without a multi-modal network stack. However, it doesn't particularly help either A or B.

(iii) and (iv) help one side while ignoring the needs of the other.

Given these advantages and disadvantages, the best option is (ii). This leaves the connection in a neutral state and higher layers can re-negotiate the algorithms used if they feel like it.

5.1.4 Re-negotiation

As such, an obvious question is whether or not A and B are allowed to re-negotiate the algorithms used in a connection during the lifespan of the connection itself and if so how.

It is quite conceivable (especially for long lived connections) that either A and/or B's power state may change substantially and that they may want

to use different power saving algorithms as a result. Also if the initial negotiation fails due to conflicts and the connection defaults to b) as described above, it is very likely that A and/or B will want to attempt to re-negotiate the algorithms used for that connection again.

There are 2 options for re-negotiating a connection.

- Just close the existing connection and start a new connection where the new requirements can be negotiated at startup.
- Dynamically re-negotiate the algorithms used while the connection is still active.

The 1st option is very simple to implement, however it is very inefficient. Closing the connection means losing any TCP state that the connection may have had. This could result in a large number of unnecessary data transfers over the new connection.

For example, say A is retrieving a large file from B via FTP. A now wants to use a different power saving algorithm for the connection. As a result, A closes the connection and creates a new connection with the new algorithms successfully being negotiated. However, it is very like that A will be unable to just resume the FTP where the previous connection left off. In this case, A will have to retrieve the large file from scratch and this will result in a very large number of unnecessary transmissions.

A will only be able to resume the FTP where the previous connection left off iff A's FTP software supports partial downloads. As such, the amount of energy lost by A (due to having to retransmit the same parts of the file again) depends very much on the application software used. This is not a desirable situation.

The better option is to allow A and B to dynamically re-negotiate the power saving algorithms that they are using during the lifetime of the connection between them. This can be done as follows.

1. A sends a packet to B with a special option indicated. This informs B that A would like to re-negotiate the algorithms used in this connection.
2. B sends an ACK to A with a special option to indicate that B is agreeable to the re-negotiation.
3. A sends a packet to B indicating the algorithms that A would like to use.
4. B sends an ACK to A indicating the algorithms that it is willing to use (these could the same algorithms that A wants to use or it could be different algorithms).
5. If A is agreeable to the choices made by B (as indicated in the previous packet), it sends a packet to B indicating that the re-negotiation was successful.
6. Otherwise, A sends a packet to B indicating that the re-negotiation was unsuccessful. A can then restart the re-negotiation process from scratch if it so desires (hopefully with a set of algorithms that would be more amenable to B).

5.2 Discussion of Experimental results

We briefly discuss our experimental results:

- For a window constrained transfer we observe that it is possible to obtain reductions in the power used per packet by almost a factor of 2. This is obtained by increasing the delayed ACK threshold to a value that is almost equal to the window size. This bunches up the packets received during an RTT period and hence introduces predictable gaps during which the receiver can sleep and hence save power.
- For a bandwidth constrained transfer we observe that the packets are always spaced apart at small regular intervals, and hence it is not usually possible to save power by sleeping during the intervals. Increasing the delayed ACK threshold value does not help much either, since the transfer is bandwidth constrained and the gaps between the bunches of packets are still small to sleep. The only possible savings in power that are obtained are by sleeping during the gaps in the slow-start period. This might be helpful in scenarios like HTTP web transfers, since most of the transfers there consist of small files, and hence a on the average, a connection spends most of its time in the slow-start state.

6 Future work

This section states some of the future directions that this project can take.

1. **Development and verification of various metrics for low power TCP.** Before we can

verify the effectiveness of any power saving algorithmic modifications to TCP, it will be necessary to develop some metrics to quantify the "goodness" of a particular metric.

Simply basing the metric on the amount of power saved is not good enough. If we did this, we would be biased towards solutions that send 1 bit every 1000 seconds and sleeps in between. Such solutions may result in the largest power savings (especially if you consider only a particular period of time), however they do so at the expense of horribly sacrificing throughput.

To counter this, we need a metric which correctly encapsulates the tradeoffs between power savings and throughput. However, it is not obvious what the "correct" metric would be.

Should the metric be a simple multiplicative prod-
uct of power savings and throughput? (i.e., $metric = energy\_saved \times average\_throughput$ or $energy\_used/average\_throughput$). Would that suffice or will we need "fancier" metrics which are biased towards one side or the other?

For example, metrics like metric = $(energy\_saved)^2 \times average\_throughput$ are biased towards saving power.

It will be necessary to test out a number of metrics in a number of different environments (PDAs, sensor networks, different network conditions etc.) to determine which metric is the best. It may well be that different situations have different "best" metrics.

2. **A complete negotiation infrastructure** More work needs to be done to clearly state the possible modes of negotiation. I.e., given a set of requirements for host A and host B, which subsets of those requirements are allowed and under which conditions?

   For example, given a communication between a sensor network device and a PDA, it is obviously not allowable for the PDA to push additional computational burden to the sensor network device.

3. **Mechanism to quantify relative states of network devices** Related to the previous item, the problem given in the example arises because the PDA thinks that it is computationally starved. Hence, it tries to offload its computation to whichever device it connects to. It would be very useful to have some sort of way of characterizing the relative difference in the capabilities of the 2 end hosts involved in a communication. I.e., the PDA will be able to realize that even though it is computationally starved, the device it is connecting to is even worse in that respect!!

   This is especially important for device specific information like power. Exchanging raw power numbers is not meaningful as 100 milliwatts of power remaining could mean a battery life of 4 days for a sensor network device versus a battery life of 10 seconds for a large notebook. It will be necessary to convert device specific information like power remaining, computational capability etc. into device independent metrics that can be used to make relative comparisons between 2 devices.

   Without this form of relative metric, it will be very difficult to negotiate any sort of optimal solution when 2 devices which both think that they are constrained by the same resource (2 notebooks running on battery for example) communicate with each other.

4. **Implementation and testing of the negotiation protocol** The negotiation protocol (currently thought off as modifications to TCPs normal handshake) needs to be implemented into the linux kernel and then tested.

5. **Feedback / Logging mechanisms** For devices to know which algorithms work best under which conditions, some sort of history must be maintained. This history will contain the results of using that particular algorithm the last time the device used it. Based on this history, the device can run some machine learning algorithms coupled with some smart heuristics to determine the best set of algorithms to use given its particular situation. The use of history to determine the correct energy saving behavior of an application is detailed in [16]. The same ideas can be used by the multi-modal network stack to decide which algorithms are the best for the device given its particular situation.

References

[1] A. Bakre and B. R. Badrinath. I-tcp: Indirect tcp for mobile hosts. In *Proceedings of 15th International Conference on Distributed Computing Systems*, 1995.

[2] H. Balakrishnan, V. N. Padmanabhan, and R. H. Katz. The effects of asymmetry on TCP performance. In *Proceedings*

of 3rd ACM/IEEE International Conference on Mobile Computing and Networking, 1997.

[3] R. Caceres and L. Iftode. Improving the performance of reliable transport protocols in mobile computing environments. In *IEEE Journal of Selected Areas in Communications*, 1995.

[4] I. Chlamtac, C. Petrioli, and J. Redi. Energy-conserving go-back-n arq protcols for wireless data networks. In *Proceedings of 9th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, 1998.

[5] I. Chlamtac, C. Petrioli, and J. Redi. Energy-conserving selective repeat ARQ protocols for wireless data networks. In *Proceedings of IEEE International Conference on Universal Personal Communications*, 1998.

[6] D. D. Clark, M. L. Lambert, and L. Zhang. Netblt: A bulk data transfer protocol. In *RFC 998*, 1987.

[7] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of 17th ACM Symposium on Operating Systems Principles*, 1999.

[8] J. Gomez, A. T. Campbell, M. Naghshineh, and C. Bisdikian. Power-aware routing in wireless packet networks. In *Proceedings of 6th IEEE International Workshop on Mobile Multimedia Communications*, 1999.

[9] R. Gupta, M. Chen, S. McCanne, and J. Warland. Web-tp: A receiver-driven transport protocol for the web. In *Proceedings of Informs*, 2000.

[10] P. J. M. Havinga and G. J. M. Smit. Design techniques for low power systems. In *Journal of Systems Architecture*, 2000.

[11] P. J. M. Havinga, G. J. M. Smit, and M. Bos. Energy-efficient adaptive wireless network design. In *Proceedings of 5th Symposium on Computers and Communications*, 2000.

[12] P. J. M. Havinga, G. J. M. Smit, and M. Bos. Energy-efficient wireless networking for multimedia applications. In *Wireless Communications and Mobile Computing, Wiley*, 2001.

[13] L. Irish and K. J. Christensen. A Green TCP/IP to reduce electricity consumed by computers. In *Proceedings of IEEE Southeastcon*, 1998.

[14] R. Kravets and P. Krishnan. Power management techniques for mobile communication. In *Proceedings of 4th ACM/IEEE International Conference on Mobile Computing and Networking*, 1998.

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

[16] D. Narayanan, J. Flinn, and M. Satyanarayanan. Using history to improve mobile application adaptation. In *Proceedings of 3rd IEEE Workshop on Mobile Computing Systems and Applications*, 2000.

[17] B. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, and K. Walker. Agile application-aware adaptation for mobility. In *Proceedings of 16th ACM Symposium on Operating System Principles*, 1997.

[18] J. Redi, C. Petrioli, and I. Chlamtac. An asymmetric, dynamic, energy-conserving arq protocol. In *Proceedings of 49th IEEE Annual Vehicular Technology Conference*, 1999.

[19] S. Savage. Sting: A tcp-based network measurement tool. In *Proceedings of USENIX Symposium on Internet Technologies and Systems*, 1999.

[20] T. Simunic, L. Benini, P.W. Glynn, and G.D. Micheli. Dynamic power management for portable systems. In *Proceedings of 6th ACM/IEEE International Conference on Mobile Computing and Networking*, 2001.

[21] M. Stemm and R. H. Katz. Measuring and reducing energy consumption of network interfaces in hand-held devices. In *IEICE Transactions on Communications*, 1997.

[22] V. Tsaoussidis and H. Badr. Tcp-probing: Towards an error control schema with energy and throughput performance gains. In *Proceedings of 8th IEEE Conference on Network Protocols*, 2000.

[23] M. Zorzi and R. R. Rao. Is tcp energy efficient? In *Proceedings of 6th IEEE International Workshop on Mobile Multimedia Communications*, 1999.