A TCP Interface for Heterogeneous Cross-Layer Awareness

Siddharth Santurkar[*]
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA, 15213
ssanturk@cs.cmu.edu

Ahmet Emre Ünal
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA, 15213
aemreunal@cmu.edu

ABSTRACT

With increasing energy efficiency requirements of modern computing devices, one of the main culprits of energy use are the network devices that connect them all together. Through this project, we have made software more friendly to the power saving modes of the hardware. We have implemented custom logic to the common Linux kernel TCP socket to allow applications to state their delay tolerance in their network communications. This allows hardware (or other software) to choose the appropriate times for packets to be sent by TCP, while still meeting the delay requirements of applications. Through our tests, we've observed that we can improve energy efficiency through less hardware wake-ups on simple common scenarios. We further elaborate on the potential for future work in this area.

1. INTRODUCTION

The pursuit of performance and the pursuit of energy efficiency are deeply intertwined. Often, engineering choices present themselves as trade-offs between these two pursuits. Through our readings in 15-744, we have seen the need for both performance and energy efficiency pervasive in every element of the global network. The difficulty in modifying network elements like routers and switches has led us to pursue end-client improvements.

With today's difference in clients and link-layer technologies, the only common element seems to be the TCP protocol and the common Linux kernel that carries it. Improving the TCP stack of the Linux kernel, be it performance-wise or energy-wise, has the potential of a massive impact on network devices in use today.

We have proceeded to explore common scenarios that lead to sub-par power efficiency in Linux and explored options to improve it; a detailed explanation and definition of this problem is available in Section 2 of this report. The rest of the paper is organized as follows: Section 3 outlines the approach we took to solve this problem. Section 4 explains the simulation path we explored. Section 5 explains the Linux kernel path we explored and finally settled on. Section 6 presents some of the results we obtained through our modifications to the Linux kernel. Section 7 concludes this paper with a retroactive look at our work. Section 8 explores the related work done in this field. Section 9 explains the future work that may be done.

[*]The order of names were decided through a friendly coin flip.

2. PROBLEM DEFINITION

Mobile computing and Internet-of-Things devices raised the bar substantially for the need for power-efficiency, often through the sacrifice of performance. Network Interface Cards (NICs) on mobile devices impose strict power saving requirements on the operating system and the applications that are running on them. It should be noted, however, that the abstraction presented to each application running on these mobile devices cause them to behave in the same way when interacting with these NICs, regardless of their requirements, often causing sub-par power efficiency or extensive application-level network logic.

We started by exploring the different performance requirements of applications an email client has a different networking performance requirement than an interactive application; it is often the case that an email client can afford delays in its communication. The current Linux networking stack responds very actively to each packet, sending the packets by waking up the NIC even if the NIC is in power saving mode. This leads to substantial drops in power efficiency in some devices [14]. The ideal case would be to keep these packets from going to the network (which prevents the NIC from waking up) and sending them once the NIC wakes up on its own, signaling the OS.

3. SOLUTION APPROACH

Once we understood the presence of this problem, we proceeded to explore options to selectively delay packages in TCP layer for a certain amount of time, based on the needs of applications. Catering a general purpose solution for every application's need was not feasible, neither was predicting each application's needs. We have, therefore, settled on allowing each application to state their requirements, through a new socket flag (alongside the common BSD socket flags). With this flag, an application can state the deadline requirements of a socket in terms of how many milliseconds of delay it can afford.

In broad strokes, our project was introducing an API to the TCP layer in the Linux Kernel where the OS maker could define an arbitrary metric to deem the state appropriate for packets being sent out. If such an appropriate time did not come by the time the deadline stated by the application was reached, the TCP layer were to process and forward the buffered packets to the network layer as if they were just handed off by the application layer. This is best explained by a couple of examples, using an email client as an example for an application and different hardware states:

- An NIC maker can invoke the push of the buffered packets on every wake up of the network card. If, for example, the network card wakes up every 100ms (as is common for the *BeaconPeriod* of power-saving mode (PSM) of the IEEE 802.11 wireless LAN specification) and the application states 500ms of delay tolerance, the network card would cause the flush of packets whenever it wakes up, without having the OS wake it up. If, for some reason, this wake up doesn't happen before the deadline stated by the application, the OS can go ahead and initiate the push of packets.
- An LTE module maker can tie the invocation of buffered packets' push to the signal strength. Essentially, packets can be buffered while signal strength temporarily drops or is disrupted, then get pushed when it returns to 'acceptable' levels (deemed by the LTE module and the manufacturer's specifications). This could lead to power savings over the current approach of increasing power levels when signal strength drops.

This approach allows each application to state their own requirements, as well as have different requirements per each socket in a single application. An email client can set up its sockets as delay-tolerant up to several seconds, while an SSH client can set its sockets as delay-tolerant only up to, for example, a couple hundred milliseconds. A file-transfer application can omit the use of our flag and can be as performant as the underlying hardware and software allows it to. Even in a single application, there might be different sockets, each with different delay tolerance levels. We believe this approach allowed the most flexibility in application performance and energy-friendliness.

Initially, we explored two avenues for the implementation and testing of our approach. The authors simultaneously worked on the network simulator ns-3 and the Linux kernel. Our approach focused on introducing a layer between TCP and IP, intercepting packets that have been processed and sent by the TCP layer. The side-effects of such an approach, namely this layer's impact on TCP's congestion control and loss-recovery algorithms, were foreseen by the authors. While the structure of ns-3 was more friendly to this extra layer, the Linux kernel required extensive modifications to introduce it. Furthermore, the modifications needed to be done on the aforementioned algorithms of TCP, both for ns-3 and the Linux kernel were going to be very extensive. These two conditions led us to intercept packets inside the TCP layer, before they have been processed by these algorithms.

This second approach proved very fruitful for the kernel path, as explained in Section 5, but not as fruitful for the ns-3 path. The structure and philosophy of ns-3 is very different from those of the Linux kernel and these modifications didn't prove very applicable on ns-3, as explained in Section 4. Finally, we stopped working on ns-3 and focused our work on modifying the Linux kernel.

4. SIMULATION

4.1 Layered Approach

Our initial approach involved introducing a layer between the TCP and the IP layers in ns-3. The reasoning behind was to identify the points where packets are done being processed by the TCP layer and are handed off to the IP layer, intercept the packets at that point and buffer them, thus preventing packets from reaching the IP layer.

We achieved this goal by changing the aforementioned hand-off function calls in the TCP layer. These hand-off points, instead of handing the packets off to the IP layer, handed the packets off to the intermediate layer we introduced.

Prior to explaining the structure of this layer, an important note should be made about the way ns-3 simulations work: While a regular Linux kernel works through timers and interrupts, ns-3 simulations work by scheduling events that perform certain actions. For example, a regular kernel will run TCP algorithms when packets are received (which trigger a series of actions), while ns-3 will schedule events such as packets being sent, packets being retrieved, timeouts happening. These scheduled events do not happen in real-time and the ns-3 scheduler can skip long intervals of time, provided there are no events scheduled between said interval.

The extra layer we introduced is structured and managed as follows:

1. The test application decides to send a packet. The application proceeds to write the bytes to the socket it opened.
2. The socket receives the data, creates the TCP packet, does its algorithmic processing. At some point, the TCP layer decides it is time to push the packet to the network (this decision comes through things like receiving ACKs, window calculations, timeouts, etc.). Anything that happens until this point is 'vanilla' TCP implementation.
3. The TCP layer calls its functions to hand the packet off to the IP layer, in order to push it to the network.
4. The packet reaches our layer, instead of the IP layer. Our layer puts the packet in a buffer (structured as a queue) and schedules, if not already scheduled by a prior packet, a 'push' event for a certain amount of time into the future.
5. Subsequent packets that arrive to our intermediate layer are put to the back of the queue (so that packets are sent out in a FIFO order).
6. When the scheduled event fires, all the buffered packets are sent out from our intermediate layer to the IP layer[1].

Further work needed to be done on modification of timeout calculations to prevent pre-mature timeouts caused by the increase in RTT perceived by the TCP layer. The timeout calculations were to include the time the packet spent being buffered in the intermediate layer. However, due to the change in approach (the transition to buffering packets

[1]An important point to raise here is how pushing all the packets affect network congestion. Since the packets have been pushed out from the TCP layer after being subjected to processing, they've been already accounted for in TCP's window calculations. Therefore, other than the burst effect on network devices and IP layer buffers, the packets being pushed do not have a substantial negative effect on network congestion.

when they reach the TCP layer but before they are processed by TCP's algorithms, as explained in Section 4.2), this modification was not performed.

By having this intermediate packet buffer, the goal was to simulate a similar entity in the Linux kernel.

4.2 In-TCP Approach

Newer TCP flavors, such as NewReno, employ many changes over the 'vanilla' TCP they introduce new states such as fast recovery. These states have different properties in window and timeout calculations. After considering the number of cases needed to be handled by the modifications needed for the layered approach, a change in our approach altogether seemed to be a better solution. Thus, we have changed our approach: Instead of introducing a new layer between TCP and IP layers, we were to modify the point where data is handed off from the application to the TCP layer. The packets created from the data were to be held in TCP's buffers *before* TCP processed them with its window or timeout calculations.

This approach were to allow us to prevent many of the potential problems caused by the increase in RTT. However, the modifications required for this approach had unexpected effects on the scheduler of ns-3. Unexpected connection terminations and crashes in simulation led us to focus more on the Linux kernel after getting promising results from the same approach on the kernel. Thus, this approach was never implemented fully on ns-3.

5. THE LINUX KERNEL

The majority of our work went in to implementing the desired API in the Linux kernel. There were certain components that needed to come together for the API to function as intended:

- A timer functionality, in order to create timers that fire at (approximately) the desired time; used to create the deadlines for the sockets.
- A BSD-like socket flag; used by applications when opening a socket to indicate their deadline requirements.
- A list of sockets separate from the ones TCP uses internally; used to keep track of only the sockets that use the optional flag we introduce.
- A custom system call that signals TCP to flush the packets residing in the buffer of the sockets that use our custom flag; used by applications or hardware to signal the appropriate time to flush the blocked and buffered packets in TCP to the network layer.
- Interception of select TCP output functions for synchronization with our custom blocking timer.

The items were implemented in the order they were listed. Together with our modifications, the custom sockets in the Linux kernel's TCP layer is structured and managed as follows:

1. An application opens a socket to send data to another host. Along with the regular BSD-style socket flags, the application uses the custom flag to set the delay requirements of that specific socket. For example, in Python scripting language, a client can open a socket with a certain amount of delay as follows:

```
# Custom socket flag int value
SOCKET_FLAG_NUM = 100
# Define delay tolerance of the socket in ms
delayToleranceInMs = 5000 # 5 seconds of delay
# Create a socket
fd = socket.socket(socket.AF_INET,
     socket.SOCK_STREAM)
# Set socket option
fd.setsockopt(socket.SOL_SOCKET, SOCKET_FLAG_NUM,
     delayToleranceInMs)
```

2. The TCP layer, in addition to setting up the regular socket, also sets up additional data structures (like the aforementioned additional list) and a socket-specific timer to manage this socket's buffered packets and deadlines. The timer is set for the amount specified in the flag

3. When data is sent to the socket, a flag in the socket is checked in order to decide whether to forward data to the window or not. If the socket is currently blocked (which is the default state for a custom socket), the packets are buffered but the window calculations happen as if no data is present in the socket buffer. Essentially, TCP algorithms are unaware data has been sent.

4. Either when our system call happens or when the timer fires (thus signaling the deadline for the packets in buffer), our algorithms proceed to traverse our custom sockets' list and flushes their buffers one by one. This results in all of the outstanding data in a custom socket's buffer to be flushed to TCP's main algorithms and gets included in the window calculations.

5. When a new custom socket is opened, our algorithms compare the deadline stated by this new socket against the existing ones. The shortest deadline is chosen as the single timer value (there is always a single timer, no matter how many custom sockets are open) and the timer is set. This is best explained with an example:

   Assume that there is a single custom socket with a 5000ms delay tolerance. For the sake of argument, assume no other sockets (custom or otherwise) or no system calls being fired. This means that there is a single 5000ms timer, firing and flushing said sockets buffer every 5 seconds. Assume that a second socket is created, with a deadline requirement of 2000ms. Our code in the TCP layer: a) checks all the open sockets, b) determines that the shortest deadline over all the sockets is 2000ms, and c) cancels the existing timer of 5 seconds and schedules the new timer for 2 seconds. Since the second socket's deadline requirements will cause the flush of its packets every 2 seconds (thus waking up the entire networking stack and hardware), it is only logical to also send the packets of the first socket while everything is already awake.

   If or when the second socket, with the 2000ms deadline requirement, gets closed, the global timer is again canceled, our algorithms look through the existing custom sockets, determines that the earliest deadline among all the custom sockets is 5000ms and sets a new timer for 5s. This approach leads to dynamic adjustment of performance of our custom sockets.

The key part of this implementation is to synchronize our delay timer with the the TCP output interface. Based on our readings from [1], [2] and [3] we found that the `tcp_transmit_skb()` function is the lowest level function in TCP responsible for *all* output operations. Packets that reach this function are expected to be transmitted right away and are immediately tracked for timeouts and retransmissions. A preliminary approach was to buffer all packets that reached this call and asynchronously flush them on timeout. However, this would require us to do additional bookkeeping to avoid spurious timeouts or retransmissions. For these reasons, we identified the following higher-level functions that simplified interception:

1. `void tcp_push()` - This function traverses the queue of buffered packets within the send window and flushes them sequentially to the lower layers at `tcp_transmit_skb()`. We intercept this function in order to prevent transmission while the timer is active. The void return type aids in not requiring to maintain any state across multiple calls to `tcp_push()`. As `tcp_push()` traverses the entire queue of packets, a single call on timer callback would suffice to flush all packets that are ready to be sent.

2. `void tcp_retransmit_timer()` - This function is invoked by TCP when the retransmission timer expires and flushes the tail of the TCP write queue (i.e., unacknowledged packets). Similar to `tcp_push()`; all the unacknowledged packets at the time of call are retransmitted. Hence, we track if this function was ever invoked while our delay timer was running and invoke this function once to retransmit all the packets present in the retransmit window.

3. `void tcp_xmit_retransmit_queue()` - This function is invoked by TCP for fast retransmit (i.e. 3 dupacks). This interception and callback is similar to that of the previous function.

4. `void tcp_send_ack()` - This function is invoked every time TCP needs to send an ACK. Unlike, the previous functions, this function needs to be invoked exactly once per ACK. In our implementation we track the number of times this function was called (i.e., the number of ACKs to be sent) while the delay timer is running. We then invoke it these many times, once the delay timer has expired, sending all the ACKs at once.

The sockets that do not use our flag are set up and managed the same way as if no modifications are present; the timers or the system calls will not affect them in any way. The only exception to this is if a regular socket causes a change in the state of the system that leads to our buffer flush system call to be fired, in which case a regular socket's send causes the packets in the buffered sockets to be sent as well. This behavior improves performance and reduces the delay of the packets if the network card is woken up by a regular, non-delayed socket wanting something sent (which would be the cause of the invocation of the system call), it would be logical to also send the buffered packets while the card is already awake.

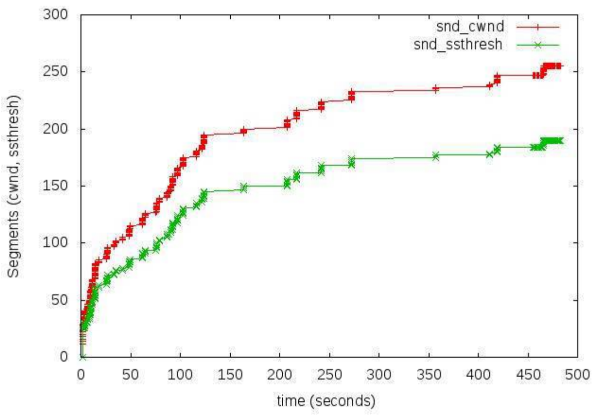

Figure 1: Bulk transfer over standard TCP (B=20Mbps, RTT=130ms)

6. EVALUATION

For evaluating our implementation, we wrote a simulation program that provides us with the NIC (Network Interface Card) behavior interrupts that we are interested in. Based on the original design of 802.11 PSM [19] the NIC can go to a low-power (sleep) mode if it has no incoming packets and no packets in its sending queue. It remains in this state for an amount of time that equals *BeaconPeriod*, which is typically around 100ms. The wireless Access Point (AP) buffers packets for the NIC while it is sleeping. This sleep could, however, be interrupted by packets that need to be sent. Through our evaluation, we inspect the modeled power savings achieved by applications that specify an upper-bound on the delay that they can tolerate, to assist our interface in trading off throughput for power savings.

The key observation here is, that, in static PSM [15], the NIC has to wake up and synchronize with the AP. This allows us to send packets for applications with higher delay tolerance, earlier, offering better throughput and reducing send message queue length as against waiting for the upper bound. While the power utilization is expected to increase from the upper-bound case, it does help in providing a good balance between throughput and power savings.

6.1 Bulk data transfer

As discussed in [15] while bulk data transfer is happening over a NIC in PSM, it never goes to sleep and shows similar characteristics as that of standard 802.11 NIC usage. To evaluate this, we configured our NIC simulator to not send any calls to TCP in the kernel. By not receiving these signals, our implementation detects that the NIC is always awake and doesn't block the TCP output interface. We performed this evaluation over a 20Mbps link with an RTT of 130ms. We implemented a sample client and server similar to iperf [17] wherein the client sends a bulk volume of messages and the server responds with lightweight acknowledgments. We designed the client to make use of our Socket flag to configure the delay settings. With the help of TCP

Probe [18] we were able to obtain the segment size, congestion window size and slow-start threshold from the client to server transfer.

Figure 1 and 2 show similar segment size growth patterns, except for the portion around $t = 60s$ and $t = 150ms$, where the congestion window fell to the slow start threshold and continued increasing after recovery. We believe that delayed acknowledgments are responsible for this behavior and the solution for this is to either snoop at the base station or modify the server to be aware that the clients are intentionally delaying packets for power-savings.

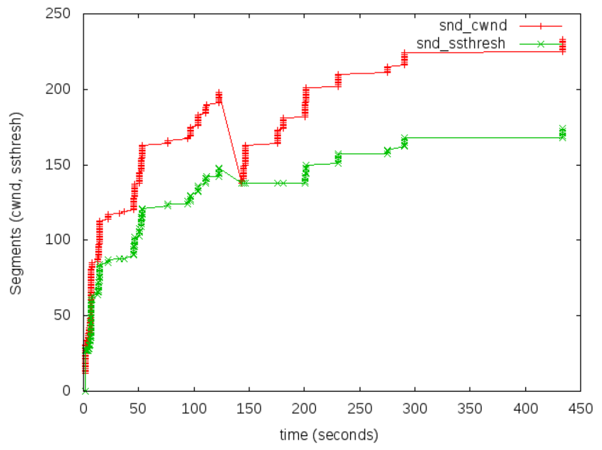

Figure 2: Bulk transfer over modified TCP (B = 20Mbps, RTT = 130ms)

6.2 Bursty transfers and power savings

In the second set of our experiments, we modified the bulk transfer client to send traffic in bursts (similar to a VoIP application). The motivation for this was to observe the potential power savings our modification to TCP had to offer, as bulk transfer has essentially the same power consumption characteristics as standard TCP. Our payload consisted of batches of 32 byte packets, with the batch size varying uniformly in random between 350 and 400 packets. The work done in [15] provides the specification for some WiFi cards. We designed our model around the interface card listed in Table 1.

WiFi Interface Card	**PSM** (P_{psm})	**Active** (P_{active})
Linksys WCF12	256mW	890mW

Table 1: Interface card configuration

We assumed that the NIC immediately proceeds to PSM once it is done transferring its payload. Based on this, our model can be mathematically represented as Equation 1, where we average out the the number of bytes transferred corresponding to the delay configuration supplied by the application. These are a set of N events that each transfer m_n bytes of data, generated throughout the entire experiment duration T. The link's bandwidth is B (= 10Mbps) and t_{sleep} is the time through the entire experiment duration when the NIC is not active. P_{psm} is the hard lower bound of power utilization.

$$E[Power] = \frac{\sum_{\forall n \in N} (P_{active} * \frac{m_n}{B}) + P_{psm} * t_{sleep}}{T} \qquad (1)$$

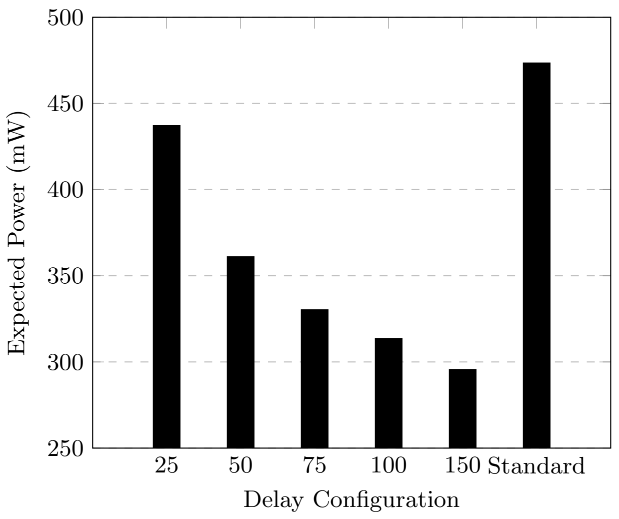

Figure 3: Expected power consumption in bursty transfer.

In our experiments, we refer to the delay passed by the application using our socket flag as the delay configuration. We set the delay configurations as 25ms, 50ms, 75ms, 100ms and 150ms for different experiment runs. Standard indicates unmodified TCP.

The key observation from Figures 3 and 4 is that significant power savings can be achieved from applications with relaxed QoS requirements by trading off bandwidth for best-effort synchronization with the NIC's beacon interval. Further analysis of our results are as follows:

1. Delay Configuration = 25ms: This is less than the *BeaconInterval* of the NIC. However, the NIC can remain asleep for 25ms at a stretch and we observe around 7.67% of power savings

2. Delay Configuration = 50ms: This is less than the *BeaconInterval* of the NIC. However, the NIC can remain asleep for 50ms at a stretch and hence we observe around 24% of power savings

3. Delay Configuration = 75ms: This is again less than the *BeaconInterval* and we observe around 30% of power savings

4. Delay configuration = 100ms: this matches with the *BeaconInterval* of the NIC and forms the upper bound on the savings that can be achieved. This is because an active NIC flushes all the active connections with buffered packets. We observe around 33% of power savings

5. Delay Configuration = 150ms: The bar plotted in the figures corresponds to a hypothetical *BeaconInterval* value that is $\geq 150ms$. In [13], the authors discuss how increasing the *BeaconInterval* adversely affects delay due to buffering of incoming packets at the AP and the rareness of the RTT values exceeding 100ms given

the high-bandwidth networks that we have today (i.e. only propagation delay is the limiting factor). In reality, this bar should be the same as the Delay Configuration of 100ms case, but this was plotted to get a better idea of the power saving potential by using longer BeaconIntervals. We observe 37.5% savings in energy.

The second key observation from our experiments is that power savings exponentially increase as we increase the configured delay until around 50ms. At this point, we approach closer to the hard lower bound of $P_{psm} = 256$mW, giving reduced savings despite higher increase in the configured delay. From these observations, we believe that the diminishing power savings do not merit an increase of the BeaconInterval to vales higher than 100ms.

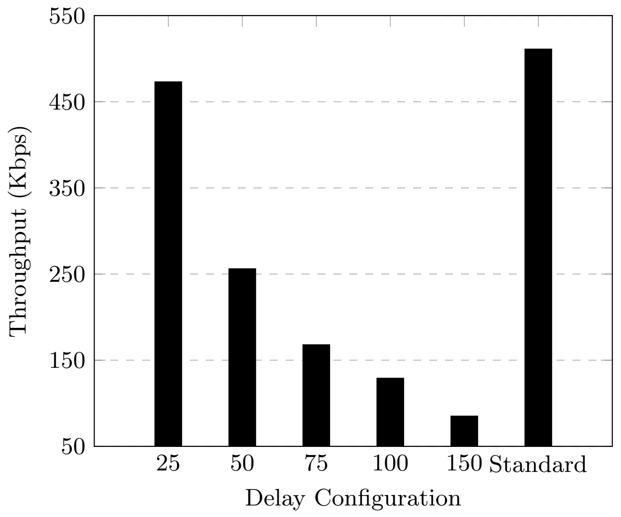

Figure 4: Throughput for each delay configuration.

7. CONCLUSION

Through our work on this project, we have observed the pursuit in energy efficiency in modern computing devices. We have then surveyed the literature on what the research world has done towards this goal. We have thus identified an area that didn't see much work performed on and sought to improve the energy efficiency in this area.

Our work amplified the presence of, along with the benefits, the problems layered approaches bring to the table. While working towards having common algorithms, used for all the scenarios present in OS networking environment, makes implementations adaptable, it is detrimental to certain use cases. These cases require inter-layer knowledge and communication.

We seeked to implement a way applications can signal their delay tolerance to the TCP layer, as well as a way hardware (or other software) can signal the appropriate times for network communication. Through a very simple cross-layer signaling mechanism, we have achieved significant power savings in simple common scenarios.

We believe that our work, while being small in scope, outlines the potential to improve the status quo through similar inter-layer approaches. While recognizing the problems that an inter-layer approach presents (such as not being hardware-agnostic and needing additional work from applications), we have identified several benefits of this approach:

- By having a simple, common modification to the Linux kernel, applications and device drivers can be easily modified to reap the benefits of our work. Through the literature we read, one common complaint prevalent in networking research was the difficulty of modifying existing network devices and/or deploying modified network devices to real-world networks. We believe that by only needing very small modifications to applications (and possibly network device drivers), we can achieve much faster and wider deployment.

- We believe that our cross-layer approach has the same trade-off as Domain-Specific Programming Languages that let go of some 'completeness' (applicability to many problems across many domains) to achieve higher performance while still being easy to develop with.

Our work can be found on our public GitHub repository[2].

8. RELATED WORK

Our initial work focused on studying and improving the performance of TCP over high-delay (and potentially lossy) networks like wireless networks. Our preliminary research led us to the project in its current form[4] [5] [6] [7] [8] [9]. We have observed a higher potential for new research in the area of improving power efficiency and performance, thus we have focused our efforts in that direction.

A substantial amount of the work in the field has been done on improving wireless performance or energy efficiency through modifications to the link layer. Most of such research went into making link layers more aware of TCP but we have seen little research that focused on modifying applications to improve performance or energy efficiency. The inability to do extensive deployment in networking research leads us to believe that application-level modifications have the potential of being deployed much quicker and wider than link- or transport-layer modifications.

While a number of articles are available on link-layer modifications to improve wireless performance, [10] contains a good comparison of various such mechanisms.

Some research focuses on modifications to the power-saving mode of network interface cards, through wake-up interval optimizations[11] [12] [13] [14]. While useful, such optimizations fail to account for arbitrary application behavior and only deliver a limited amount of energy efficiency. Most of the gains from the optimization of wake up intervals seem to be for performance than for energy efficiency.

Another area of research focuses on switching between different radio modules and multiplexing between them to improve energy efficiency in wireless communication [15].

One work that is close to our approach is Dogar et al.'s Catnap [16]. Catnap works by being a proxy in a communication and delaying transmissions in order to increase the sleep time between transmissions. This allows the network card to sleep for longer durations, rather than intermittent short periods. The Catnap approach is different in key areas:

1. Catnap is implemented as a separate proxy that intercepts and schedules transmissions. Our approach

[2]https://github.com/sid1607/linux-3.14.65-src

happens completely in-kernel and there is no middle entity.

2. Catnap is designed to be application independent. Our approach assumes that the application is in the best position to know its delay requirements and could communicate it, which, in turn, removes the 'guessing game' from the equation.

3. Catnap decouples the wired and wireless segments of the communication, being much more hardware-specific than our implementation. Our approach allows power savings in every connectivity scenario.

9. FUTURE WORK

There are a few outstanding problems with our implementations that require further work:

1. The current implementation schedules timers even if there are no data to send at any of the custom sockets. This leads to a slowdown due to the unnecessary traversal and checks of the custom sockets at the timer intervals.
   Future work could involve a smarter timer algorithm that schedules a timer only when there is data to send.

2. Due to the uncertainty in concurrency of the access to the custom socket list, the list traversal algorithms employ a [possibly] conservative hand-over-hand locking algorithm. This requires taking a lock on every single node, for all of the operations. With a better specification of the structure of concurrent access to the sockets in question, the socket traversal algorithms could be improved and an increase in performance can possibly be observed.
   Future work could involve the study of kernel socket access mechanism, creation of a stricter access mechanism and thus a less conservative approach to socket list traversal and modification.

3. Somewhat related to Item 2 above, the socket list traversal and modification algorithms currently in use take a long time for simple operations. For example, when a flush event happens (due to a timeout or system call), the traversal algorithm goes through each socket and initiates the push of the sockets' packets in a blocking fashion. Therefore, the nodes can stay locked for an extended period of time and have a negative impact on the performance of both the kernel and the callee of the system call. While the work mentioned in Item 2 would help, it wouldn't solve the entire problem.
   Future work could focus on making these events non-blocking and/or concurrent, so that performance can be improved.

4. Highly related to Item 3 above, the time it takes for single operations to complete makes it infeasible to have deadlines smaller than at least a few hundred milliseconds. This leads to somewhat unusable cases for quite a number of applications that can't afford more than approximately a second of delay but could afford few hundreds of milliseconds. The work on Item 2 and Item 3 would lead to a substantial increase in the number of applications that can use our implementation.

5. We have implemented our structure for a single kernel version and tested using a single TCP 'flavor', with a limited set of test scenarios.
   Further work could focus on testing our modifications on a wider range of scenarios, hardwares, applications, and situations.

6. In order to get most of the efficiency our implementation provides, a substantial number of sockets in a system need to use our custom socket flags and each application needs to put in an effort to understand their use and optimize their socket flags accordingly. Since our implementation completely relies on application choices, the benefit our modifications provide go as far as the applications that use them.
   While this situation has been our original aim and that there is nothing wrong with in in the scope of our project, future work could focus on ways to automatically and intelligently making such decisions for arbitrary sockets in the system.

7. Somewhat apparent, but still an important problem is the lack of substantial testing of the modifications against bugs that might have been introduced.
   Future work could focus on unit or integration testing our modifications and fixing bugs.

10. REFERENCES

[1] Miguel Rio, Mathieu Goutelle, Tom Kelly, Richard Hughes-Jones, Jean-Philippe Martin-Flatin, and Yee-Ting Li. A Map of the Networking Code in Linux kernel 2.4. 20, 2004.

[2] Pasi Sarolahti. Linux TCP. *Nokia Research Centre*, 2002.

[3] Paul Willmann, Scott Rixner, and Alan L Cox. An Evaluation of Network Stack Parallelization Strategies In Modern Operating Systems. In *USENIX Annual Technical Conference, General Track*, pages 91–96, 2006.

[4] Maxim Podlesny. Networking mechanisms for delay-sensitive applications. 2009.

[5] Doug McCreary, Kang Li, Scott A Watterson, and David K Lowenthal. TCP-RC: A Receiver-centered TCP Protocol for Delay-sensitive Applications. In *Electronic Imaging 2005*, pages 126–130. International Society for Optics and Photonics, 2005.

[6] Ethan Blanton, Kevin Fall, and Mark Allman. A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP. 2003.

[7] Hari Balakrishnan, Venkata N Padmanabhan, Srinivasan Seshan, and Randy H Katz. A Comparison of Mechanisms for Improving TCP Performance Over Wireless Links. *Networking, IEEE/ACM Transactions on*, 5(6):756–769, 1997.

[8] George Xylomenos, George C Polyzos, Petri Mähönen, and Mika Saaranen. TCP performance issues over wireless links. *Communications Magazine, IEEE*, 39(4):52–58, 2001.

[9] Brian Tierney. TCP Tuning Guide for Distributed Applications on Wide Area Networks. *USENIX & SAGE Login*, 26(1):33–39, 2001.

[10] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz. A Comparison of Mechanisms for Improving TCP Performance Over Wireless Links. *IEEE/ACM Transactions on Networking*, 5(6):756–769, Dec 1997.

[11] A. Agarwal and A. K. Jagannatham. Optimal Wake-up Scheduling For PSM Delay Minimization In Mobile Wireless Networks. *IEEE Wireless Communications Letters*, 2(4):419–422, August 2013.

[12] Daji Qiao and Kang G Shin. Smart Power-saving Mode For IEEE 802.11 Wireless LANs. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 3, pages 1573–1583. IEEE, 2005.

[13] Ronny Krashinsky and Hari Balakrishnan. Minimizing Energy for Wireless Web Access with Bounded Slowdown. *Wireless Networks*, 11(1-2):135–148, 2005.

[14] Suman Nath, Zachary Anderson, and Srinivasan Seshan. Choosing Beacon Periods to Improve Response Times for Wireless HTTP Clients. In *Proceedings of the second international workshop on Mobility management & wireless access protocols*, pages 43–50. ACM, 2004.

[15] Trevor Pering, Yuvraj Agarwal, Rajesh Gupta, and Roy Want. Coolspots: Reducing The Power Consumption of Wireless Mobile Devices with Multiple Radio Interfaces. In *Proceedings of the 4th international conference on Mobile systems, applications and services*, pages 220–232. ACM, 2006.

[16] Fahad R Dogar, Peter Steenkiste, and Konstantina Papagiannaki. Catnap: exploiting high bandwidth wireless interfaces to save energy for mobile devices. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 107–122. ACM, 2010.

[17] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. iPerf: The TCP/UDP Bandwidth Measurement Tool. *https://iperf.fr*, 2005.

[18] Anders Persson, Cesar AC Marcondes, Ling-Jyh Chen, MY Sanadidi, and Mario Gerla. TCP Probe: A TCP with Built-in Path Capacity Estimation. In *Proceedings of the 8th IEEE Global Internet Symposium*, 2005.

[19] LAN/MAN Standards Committee et al. ANSI/IEEE std 802.11: Wireless Lan Medium Access Control (MAC) and Physical Layer (PHY) Specifications. *IEEE Computer Society*, 1999.

11. ACKNOWLEDGMENTS

The authors would like to thank Prof. Srinivasan Seshan for his support and guidance on the inception, evolution, and realization of this project. His knowledge, experience in the field and enthusiasm was a guiding light.

The authors would also like to thank Junchen Jiang for his guidance during the development of this project.