Next Generation TCP

Accumulated Acknowledgements on advanced TCP Implementations

Seunghoon Jeong, Han S Kim, Richard Lohwasser

Class Project
Computer Communication Networks, Amin Vahdat
Department of Computer Science and Engineering
University of California, San Diego

Abstract

Deployment of Active Networks severely increases per-packet cost of routers, shifting their performance limitation from bytes/sec to packets/sec. We address this issue by presenting a method to decrease packets transmitted for next generation, queuing delay based TCP implementations by accumulating acknowledgements when a stable network state is detected. Simulation results show acknowledgements can be reduced by 70% in uncongested, high bandwidth, low latency environments while throughput is only decreased by 3% and fairness is still maintained. The main highlight is a Linux Kernel implementation that is able to fully maintain throughput but only needs 7% of the acknowledgements in a local area network environment. All modifications are client side only and do not require modifications on the sender.

1. Introduction

Active Networks provide an elegant solution to the inextensible structures current networks impose. If Active Networks is deployed widely, modern routers will be limited by packet processing capability rather than network bandwidth [9]. As an example, network bandwidth is keep doubling every nine months whereas computing power doubles every 18 months. Decreasing the amount of packets will therefore lead to an increased performance, elevating Active Networks to a key motivation of our work.

Stable performance is another aspect of a possible future generation of TCP. Many existing protocols address this issue as current TCP Reno implementations show inherently unstable performance based on the TCP saw tooth architecture. Among others, TCP Vegas addresses this issue by using queuing delay to achieve more stable transmission rates compared to TCP Reno. In addition, it is also important for new protocols to be adequate for high-speed network environment. TCP Reno's fast recovery mechanism does not work correctly when traffic has burst packets. Therefore, our goal in this paper is to cope with these upcoming conditions by finding a way to effectively control packet usage in future network environments.

In this paper, we present a new TCP acknowledgement (ACK) mechanism that can be combined with existing unmodified TCP implementations. We present simulation results and real Linux kernel experiment results to verify correctness and validity of our algorithm. The main result in this report is that our new TCP enables normal TCP communication to reduce redundant ACKs up to 93% in local area network environments. An in-depth discussion about how our algorithm affects overall TCP throughput is also presented. With simulations, we describe fairness properties of our algorithm and that there is no performance degradation when lossy links are used, as this situation is detected. Moreover, we compare two results from real TCP implementation and the NS2 simulator to verify the two results.

The rest of this paper is organized as follows. Section 2 discusses previous works related to improving TCP performance. Section 3 presents our algorithm to improve TCP performance. Section 4 then describes experimental results with network simulation and Section 5 discusses results from real-world implementation of our algorithm. Finally, Section 6 and 7 discuss relevant future works and make conclusions.

2. Related Work

Numerous improvements of TCP are consequently proposed, such as TCP[1], XCP[2] or TCP Vegas [3]. These variants can be employed to our work that will eventually improve TCP performance. Fast TCP is an enhanced version of TCP under high capabilities and large latency using queuing delay as congestion detection mechanism and multiplicative increase multiplicative decrease to recover faster from losses. In addition, XCP is also proposed for high bandwidth delay product networks, but relies on explicit congestion notification by routers. Both protocols assume better network capability in terms of bandwidth, which will come true in near future. The common problem these variants are dealing with is about congestion *avoidance*. As current TCP implementations need to create losses to trigger the congestion control mechanism, it has been blamed for unstable transmission as we can see in the saw-tooth shape of window size. Queuing delay as a congestion avoidance mechanism provides a basis for our work in that we believe future generation of TCP implementations will employ this mechanism. However, these still fail to address a reduction of the overall packet numbers in the network.

In respect to reducing packets, there are series of theoretical works, [5, 6] providing very solid foundations for this matter. By using both deterministic algorithms and online algorithms, they calculate the optimal acknowledgement method. The goal here is to minimize the cost function of latency and ACKing. However, the more important thing in high-speed network is to achieve maximum throughput so the direction of the analysis is slightly different from our work.

Another approach is selective acknowledgement (SACK) [10]. SACK is useful especially when one segment is missing after long series of messages arrived at a receiver. Rather than retransmitting all over again after the missing segment, SACK enables senders to retransmit only the missing segment. Though the number of retransmitted packet can be reduced dramatically, the loss rate is going down nowadays and, dependant on the network characteristics, the number of redundant ACKs will be a more dominant factor than that of retransmitted data packets.

3. Algorithm

This section presents two algorithms and the rationale for the main algorithm. The first algorithm is on how the ACK-reducing mechanism operates. We illustrate the overall mechanism by looking into its behaviors on TCP receiver side, where our algorithm runs. In the latter part, we introduce the second algorithm on how the accumulated ACK size is dynamically and optimally determined during transmission.

3.1. Design Rationale

TCP ACKs are not only used for reliable transmission, but also for flow control and congestion control. In slow start, the TCP sender rapidly finds spare bandwidth by exponentially increasing its congestion window using each ACK. Duplicated ACKs are used to inform the TCP sender about a packet loss. XCP, FAST TCP and TCP Vegas also use ACKs to calculate RTT time to adjust its sending rate. Therefore, when ACKs are used to change the sending rate or indicate a packet loss event, those meaningful ACKs should be delivered promptly. Meanwhile, when transmission throughput is in a steady state, ACKs do not affect the sending rate or inform TCP sender of any meaningful change of the network state; those ACKs just tell the TCP sender to move its sliding window. Then, by accumulating those ACKs the TCP receiver can hypothetically reduce the number of ACKs without harming the transmission throughput. In high bandwidth-delay product networks, the major portion of the connection time is in a data transmission state with its maximum throughput. Since the algorithm benefits from the period of data transmission of maximum throughput, the scheme is adequate for high bandwidth-delay product. Furthermore, since TCP Vegas and FAST TCP show more stable throughput as the congestion window is more stable, those are favorable for this scheme.

3.2. Main Algorithm

Delayed ACKing and accumulated ACKing

The algorithm can be in two ACK modes. When TCP transmission is in a slow start state or in an unstable network state, the algorithm is in delayed ACK mode. In delayed ACK mode, the algorithm behaves just like the standard TCP delayed ACK algorithm. In contrast, when throughput is in a stable state, the algorithm is in the accumulated ACK mode, i.e. where ACKs can be accumulated.

ACK mode transition

Every 200ms for NS2 or 100ms for Linux, the TCP receiver calculates the current transmission rate and compares this value to the previous one. If the RTT is larger than 100 / 200ms, this calculation will be done every RTT. If the throughput delta is below 5%, accumulated ACK mode is triggered. Otherwise, it would be in delayed ACK mode in which the TCP receiver sends ACK packets for every other ACK, i.e. an unmodified acknowledgement policy. Therefore, when the transmission rate reaches a steady state, the algorithm changes its mode into the accumulated ACK mode. On the other hand, when the transmission rate decreases by an erroneous event such as packet loss or congestion or just less data sent initially, the throughput change would be over 5% and the algorithm shifts back to the delayed ACK mode, which can handle the erroneous event more quickly via letting the TCP sender know quickly. The value 5% was chosen as it provided a good balance, which is further discussed in section 4.5.

Sending accumulated ACKs

There are four cases when a receiver decides to send off immediate ACKs. First, a receiver sends an ACK when a timer is expired. The algorithm keeps a timer for any accumulated ACKs just like the delayed ACK timer. The standard delayed ACK mechanism [11] withholds an ACK no longer than 200ms. The expiration time for the accumulated ACK mode is also chosen to be the same value as that of the delayed ACK mechanism to guarantee responsiveness to losses and congestion. Therefore, if no further data packets from a TCP sender arrive for 200ms in the accumulated mode, any accumulated ACK packet is sent.

Second, when the number of accumulated ACK packet exceeds the maximum number of packets that specifies how many packets can be accumulated in a receiver and that is

determined dynamically as explained in Section 3.3, an (accumulated) ACK packet is sent. This value is initially set to 2, in which the accumulated ACK mode behaves identically with the delayed ACK mode.

Third, when unexpected data arrives. This is the case for packet loss, in which duplicate AKS are immediately transmitted and the algorithm disabled accumulated ACK mode. This is also the case for out-of-order data or specially flagged, e.g. with FIN.

Finally, when the data size of accumulated ACKs exceeds the advertised window, the algorithm sends any accumulated ACK immediately. Otherwise, a receiver keep waiting for more packets and a sender cannot send more data because the sender thinks the receiver's queue is full. In this case, the flow is abnormally disconnected[1].

The Flow chart for this algorithm is discussed in detail in Appendix B.

3.3. Optimization of Accumulated ACK Size

As mentioned above, the core idea of our algorithm is that the receiver accumulates ACK packets until the amount of received but not yet acknowledged data packets exceeds a threshold value. Since the link bandwidth and RTT are variable for each data transmission of different networks, setting an optimal threshold is an important problem in the algorithm. We suggest a simple and heuristic algorithm to find it. Prior to that, we assume that transmission rate reaches a stable state after some time and the throughput is maintained by the end of the transmission except for some number of instant unstable periods.

The main algorithm initially begins in the learning mode with the threshold value 2. Every time the current throughput is calculated, it also saves it as well as the current ACKing mode. After sampling 20 throughputs, (i.e. 4 seconds are required in NS2, 20 x 200ms) the algorithm checks if the recent half of the samples show that it was continuously in the accumulated ACK mode and if its averaged throughput was within 5% of the maximum throughput that ever was seen. In this case, data transmission has reached a steady state. Then, the algorithm concludes that it can accumulate one more ACKs than now and restarts sampling. Since accumulating ACKs keeps the sender from moving its window, the throughput that the receiver sees decreases as more ACKs are accumulated in the receiver. At some point, the receiver notices that it is accumulating too many ACKs as the throughput decrease observed by the receiver exceeds the maximum throughput by at least 5%. Then, the algorithm decreases the number of ACKs withheld by one, thereby recovering to a state of higher throughput, and stopping its learning mode. After that, the algorithm reevaluates this value every 10 seconds.

More detailed flow chart of the learning mode is presented in Appendix C.

4. Simulation Results

4.1. Network Simulator NS 2

The NS2 network simulator was used for simulation. The advantage of this simulator is that it is very handy to implement an algorithm, run tests and verify results. Rather than waiting for real time to transfer data between network channels, NS2 can finish 120 seconds scenarios within a second. From building prototype of our work to tuning our algorithm and verifying the effect of changes, it was used extensively.

For basic experiments, there are some parameters to be controlled. To find out how our TCP works under various network environments, four bandwidth values were used. To simulate very low performance network, the link speed was configured to full-duplex 1 Mbit/s. For high bandwidth networks, a 1000 Mbit/s link was used. However, most experiment results discussed here use a 100 Mbit/s link, which is now the prevalent configuration in real networks. In addition, three latency values were used, 10ms, 50ms and 100ms. The advertised window size is set to 1MB, ensuring it will never limit the sender. This was a major problem experienced during prior tests, as performance used to be limited by the advertised window size even though network capability still had spare bandwidth. For example, if a simulation with just 40 KB of advertised window size, which is the default value for the NS2 simulator, throughput is at most 50Mbit/s even in a 1000 Mbit/s bandwidth link.

4.2. Performance with TCP Vegas

First experiment is to verify whether our TCP modifications are really working. The goal of this experiment is to show how much performance we can get from our new TCP implementation in terms of the number of ACK packets that are really transferred to a sender. Another goal is to show if and if yes how much we degrade the performance of unmodified TCP senders in terms of throughput and window size. The TCP Vegas protocol was used as a sender of FTP data to our modified TCP receiver. Each packet delivered contained 1000 bytes including headers and delivery was only limited by TCP, i.e. not by a virtual network interface card. A Point-to-point network channel with no background traffic was configured for transmission with various latency and bandwidth configurations. Every test was also completed with an unmodified TCP receiver to compare results. Figure 1 is the result generated from a communication channel with 100 Mbit/s of bandwidth and a latency of 10ms.

[1] This feature has not been implemented in the Linux Kernel yet.

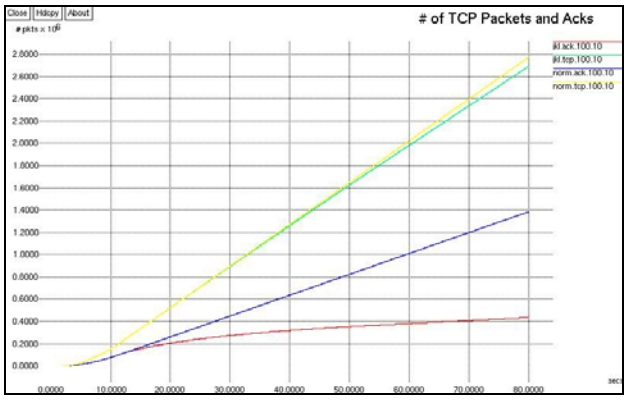


Fig. 1: Number of TCP ACKs and data packets

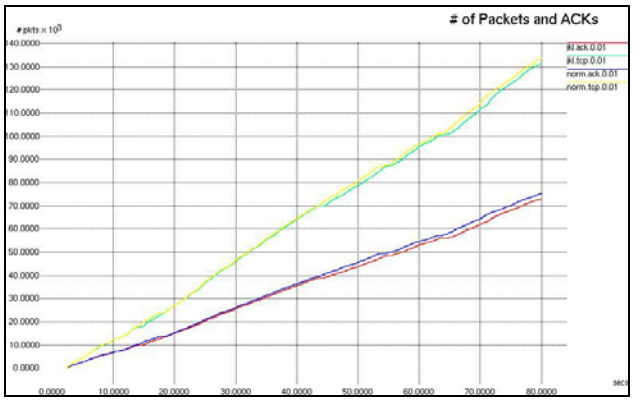


Fig. 2: Transmission under Unstable Network Link

Figure 1 shows the amount of packets (green and yellow) and ACKs (red and blue) that are transmitted to a receiver and sender over time. The yellow and blue lines indicate the normal TCP protocol. The reason for the blue line being about half the yellow line is that the TCP protocol uses delayed ACK, i.e. sends an ACK for every other TCP packet. Our TCP protocol shows a dramatic decrease in the number of ACKs compared to the original TCP as the blue line. The exact figure is 430,000 ACKs compared to 1,380,000 ACKs within 80 seconds with normal delayed ACK. This is a 70% reduction of ACKs. However, the total number of TCP packet transmitted with our TCP protocol is only slightly (< 3%) less than normal TCP. This throughput degradation effect is discussed in Section 4.5. This graph clearly shows that our algorithm can reduce most of the redundant ACKs during communication and there is little performance penalty for this gaining in a 100Mbit, 10ms point to point environment.

4.3. Loss Rate

The next experiment analyzes how loss rates affect the performance of our TCP modification. As explained in Section 3, if there is a change in network state, throughput calculated at receiver's side would also change. In this case, our mechanism goes back to normal TCP mechanism in at most 200ms. Consequently, the goal for this experiment is to show that our TCP mechanism can achieve the same throughput with normal TCP when there are losses in links. Another method to verify this property would be to inject only one loss and see what happens over time. This would involve an examination on the delay of our TCP switching back to normal TCP and how much delay and throughput loss is caused by the loss. However, in the NS2 simulator, it is hard to design in this way. Rather than specifying an exact time to invoke a loss, losses are defined as a random variable. Therefore, every packet transmission is an event and for each event the loss model decides whether this packet can be delivered safely or not. The loss model we use is a uniform random variable with probability 0.01, i.e. 1 out of 100 packets is dropped on average during transmission.

By the randomness, for 80 seconds, there were 444 packet losses in our TCP transmission with 452 packet losses in the normal TCP. Figure 2 shows the difference between our TCP and normal TCP is negligible as the number of packet transmitted to a receiver is almost identical (< 1.5%) to the normal TCP transmission. As the two ACK lines are almost identical, we can conclude that our TCP mechanism stayed in normal TCP ACK mode in most of time during simulation. This indicates that throughput change with immediate fallback to normal ACK mode on packet loss is a reliable indicator of the network condition in this case.

4.4. Fairness

Whenever a new TCP variant is proposed, it always has to guarantee fairness. If the protocol cannot prove that it is fair when a flow of the protocol shares a link with other TCP protocols, there is little possibility for the protocol to be deployed in a real environment. If the new protocol outperforms and hurts performance of other TCP, then ordinary TCP users would suffer from the protocol. Conversely, if the new protocol always underperforms a regular TCP, no system administrator would try to deploy the protocol. Therefore, proving fairness for our TCP protocol is essential for wide deployment.

However, our TCP imposes no modifications on congestion control and flow control mechanisms as explained in Section 3. One could therefore easily conclude that our TCP would not alter any fairness properties.

To prove this concretely, we conducted an experiment shown in Figure 3 in which two TCP flows share one common 100Mbit/s, 10ms latency link. Initially, only one sender transfers data over the wire. After 40 seconds, a second stream is established which uses the same wire. The experiment was independently conducted once with a regular TCP Vegas implementation (red and green), and once with our modified receiver (blue and yellow).

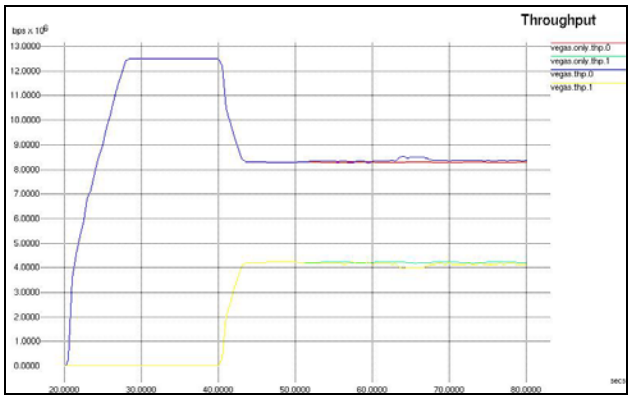


Fig. 3: Fairness Test with a TCP Vegas flow

Figure 3 shows the result of fairness test. In this fairness test, TCP-Vegas itself does not show that it is fair in terms of sharing bandwidth. In TCP-Vegas paper [3], a discussion about fairness tells that without some help of routers fairness property is unlikely to happen. However, they claim that Vegas is at least as fair than Reno. Therefore, our strategy to prove fairness properties is also similar to that of Vegas; if we can show that our TCP is as fair as Vegas, then we also can conclude our fairness to TCP.

In Figure 3, two TCP-Vegas flows shares one link of 100Mbit/s bandwidth and 10 ms latency. The red and green lines represent two TCP-Vegas flows and blue and yellow line represent a TCP-Vegas flow and our TCP flow. Even though there is a little difference between the two pairs, this difference is caused by the threshold value discussed in Section 4.5 and not by fairness properties.

4.5. Threshold Value

'Threshold value' stands for the degree of throughput change in a flow that is tolerated. If the difference in throughput measured every 200ms[2] stays within the threshold value, a receiver switches to accumulated ACK mode, i.e. sending an ACK after accumulating a certain number of ACKs. For example, if the threshold value is 5%, then our TCP can stay in accumulating ACK mode as long as the throughput change over time is less than 5%. The maximum number of ACKs that can be accumulated in a receiver is determined dynamically as explained in Section 3.3. On the contrary, if a receiver accumulates too many packets before sending out an ACK, a sender should wait for a long time to get responses. Then packets waiting for an ACK can be queued up at the sender, which eventually decreases sending rate and throughput of the flow. Therefore, the learning process, which determines on the fly how many packets can be accumulated, is influenced by the threshold value. Thus, the threshold value plays a role on determining two factors: how many packets can be accumulated at a receiver and how much throughput can be decreased for the algorithm to remain in accumulated ACK mode.

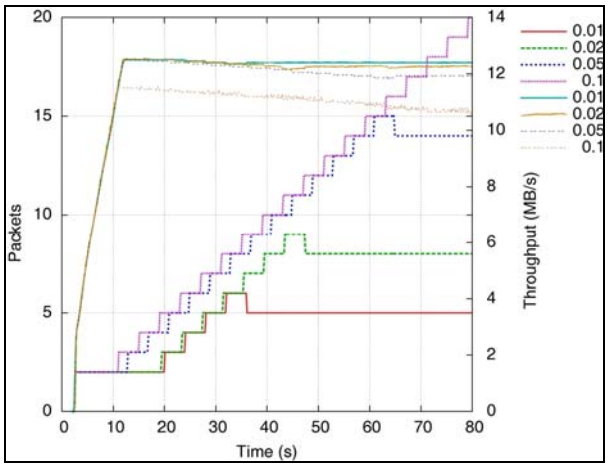


Fig. 4: The Effect of Threshold

Figure 4 shows how the threshold value affects overall performance. Stair-like curves represent the learning mode of how many packets can be accumulated without any throughput change more than the threshold value. Because a threshold value of 1% is tighter than any other values, the learning process ends its process early at just 5 accumulated data packets per ACK. However, threshold value of 10% allows the learning process to increase the maximum number of accumulation beyond 20, indicating it has not finish within 80 seconds. Therefore, with larger threshold value, more ACK packets can be reduced.

Four lines at the top of Figure 4 indicate how much throughput was achieved over time. With a threshold value of 1%, our TCP protocol achieved 12.4MB/s throughput after the maximum number of accumulation was set. However, only 10.5MB/s throughput was obtained with threshold value of 10%. The obvious trend of this graph is that throughput is monotonically decreasing and the maximum number of ACK accumulation is monotonically increasing as threshold value increases.

4.6. Various Network Environment

Figure 5 shows throughputs under various network bandwidth sizes. A fixed latency of 10ms is used for this experiment and various bandwidths are set for links. The Y-axis is a log 2 scale. It is easily seen that two lines are overlapped, one for normal TCP and the other for our TCP, which means there are little difference between normal TCP and our TCP mechanism in terms of throughput under low latency network. However, currently our algorithm is not showing similar graphs to Figure 5 when latency increases up to 100ms. Although point-to-point channel with 100ms is hard to be constructed in real world, it is worthwhile to see how it works under very long latency links. Unfortunately, our algorithm shows irregular throughput trends above 100 ms latency. In a 10 Mbit/s link, throughput is only 0.6 MB/s whereas with normal TCP throughput increases up to maximum bandwidth. In 100 Mbit/s and 1000Mbit/s, throughput converges to 0.4 MB/s.

[2] In general. See section 3.2 for the exact timer interval.

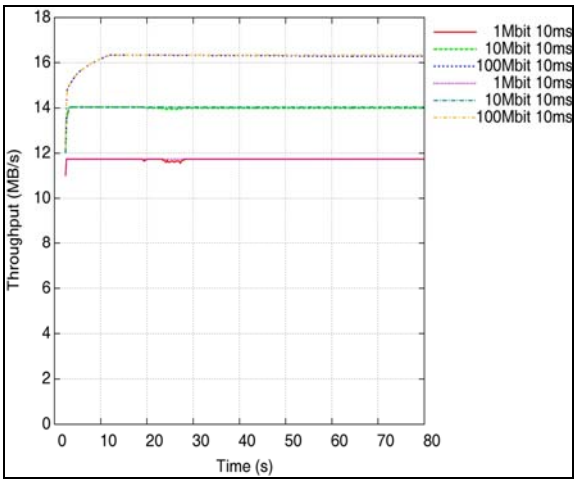


Fig. 5: Various Network Environments

5. Experimental Results

To validate results gained with the network simulator and to further gain insights into real-world implementations of TCP the algorithm described in Section 3 was implemented in the TCP stack of the at that time most recent Kernel obtained from www.Kernel.org in Version 2.6.15.4.

5.1. Implementation

Changes directly in the kernel source code are very difficult to validate, as modified parts have to be recompiled and a full relinking has to occur. In order for the changes to take effect, a system reboot is required. In case of a failure (Kernel panic) an old kernel has to be loaded and the bug has to be found with only limited and just static debugging information, as just the trace from the kernel panic is available. After these fixes, the kernel has to be recompiled, relinked, reinstalled and rebooted again. We therefore use Loadable Kernel Modules (LKM's)[3] to dynamically inject kernel space code in to a running system. The kernel patching is then only limited to (1) definition of a global kernel-space variable that tells whether the module has been loaded[4], (2) definition of global kernel-space function pointers that will be overwritten by the module at startup and (3) calls to the function pointers at the beginning of every relevant function that is executed if the module has been loaded. If the module is loaded, it does not execute the rest of the function in the kernel, thereby rerouting functionality from the kernel to the pluggable kernel module.

[3] See [12] for details about LKMs. For hooking also refer to the CSE222a class paper [13].

[4] Strictly speaking, this is redundant, as this check can be performed if the function pointer variables have already been overwritten

5.2. Code size

As the modified code portions are executed very frequently, it is crucial that the additional per-packed cost imposed by our algorithm is as small as possible, especially when deployed on a large scale. Without the overhead of hooking, i.e. in a plain kernel-implementation our additional per packet cost is in most cases limited to only 9 instructions. These consist of 2 assembler instructions for the update of the bandwidth counter and 7 for the check if more than 100ms have elapsed since the last execution. If the RTT exceeds 100ms, the RTT is used for this timer. In a 100Mbit LAN environment, this timer will fire every 1600 packets, which will then cause additional computation of around 100 instructions, which is negligible.

The actual size is the module is around 500 lines of C code, mainly consisting of debugging information and code necessary to support hooking. An actual kernel implementation would be less than 100 additional lines of code. For a comparison, the current TCP stack of the linux kernel that was worked on, not including header files consists of 58.000 lines of C source code.

5.3. Test Setup

The test setup consisted of 2 machines with one router in between in a LAN. All links were 100 Mbit full duplex and the router backplane was large enough to not interfere with the experiments. There was no background traffic during execution and we assume due to the time the experiment was done that the router queue was not influenced by other traffic on other networks it served. Both machines used Red Hat Enterprise Linux 4 AS. The sender used an unmodified stock kernel with TCP Vegas enabled whereas the receiver used our modified kernel. The application on top was a chargen server on TCP port 19 which just outputs characters as fast as possible[5], thereby assuring us no disk access overload etc. would influence results. The client application was a simple retrieval tool programmed for the second assignment of CSE222a that stops execution after a predefined number of bytes have been transferred. This environment enables transfers of more than 4GB, which was a problem we faced in prior test setups. Test results were gathered with tcpdump running on the sender listening on the specific port to ensure valid and unbiased results.

5.4. Test Results

As described in Fig. 6 we see that the amount of data packets sent is almost the same as the difference is less than 1%. As the data packet size is constant, this difference directly relates to achieved throughput. The amount of ACK packets received by the tcpdump is significantly different though: Whereas the regular TCP Vegas implementation uses roughly one ACK for every other data packet, the new implementation only uses one for every 45 packets, resulting in a decrease of 93% of the acknowledgements. This value of 45 data packets for every acknowledgement is broken down in Figure 7, which displays this value over time. During the first 100 seconds, the

[5] See [RFC 429] and [RFC 864] for further information on chargen.

algorithm is continuously in learning mode, trying to increase the amount of packets until it notices a decrease in throughput of at least 5%. Every attempt to re-evaluate the optimal value every 10 seconds fails, i.e. 46 packets per ACK cause a decrease of throughput of more than 5%. An aggregation up to 45 packets however does not affect throughput at all.

Results with TCP Reno show very similar results in terms of throughput, but the optimal amount of accumulated packets is increased from 45 to about 60. The graph with the comparison is in Appendix D.

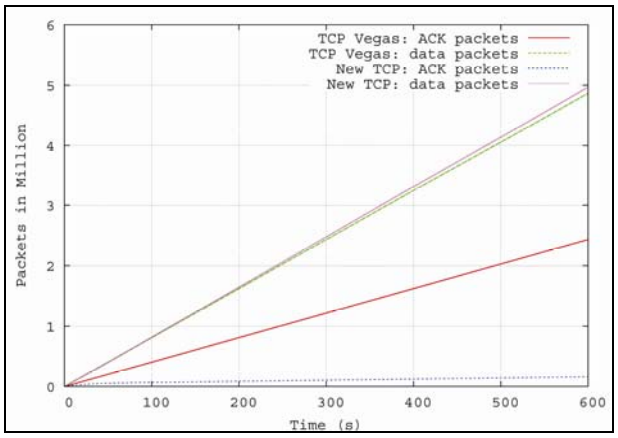


Fig. 6: Number of TCP ACKs and data packets

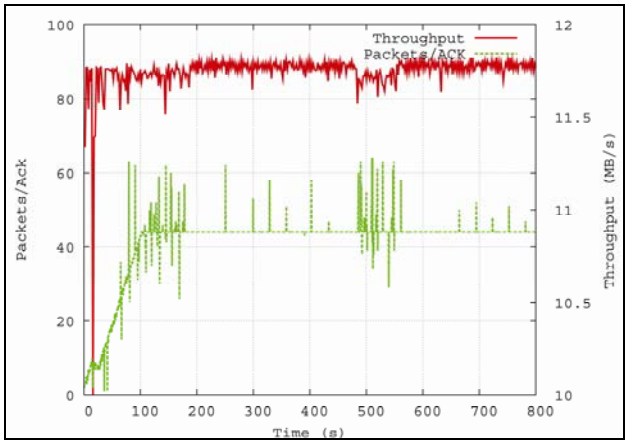


Fig. 7: Throughput and Data-Packet/Ack ratio

5.5. Interpretation

To analyze why TCP is so unaffected by accumulated acknowledgements we systematically go through all entities capable of throttling throughput in the current implementation of the TCP protocol in the Linux kernel.[6]

1. **TCP Nagle test, Urgent Data:** During our tests these checks were never relevant.
2. **Congestion Window Size (cwnd):** This value is obviously not throttling TCP if less than 45 packets are accumulated. This also means that late acknowledgements did not trigger timeouts, which would have decreased the congestion window size. Further is surprising that the window is not even close to being exceeded, as it can take up additional 43 packets before throughput is decreased. In normal TCP connections, we expect cwnd to be responsible for throttling of throughput. In our LAN environment, this is not even closely true, as it is able to take up at least 43 Packets *1448 MSS = 62 KB before it could possibly affect throughput. The reason cwnd is so high is because it never is cut down: The network device only puts data on the wire at a rate of 100Mbit/s, and the rest of the network can fully sustain this flow. Therefore, TCP can never send at a rate at which it induces losses or causes queuing delay, i.e. cwnd is never decreased, and is therefore irrelevant for our specific setup.
3. **Receiver's adv. window size (rwnd):** Our packet log files show that Linux advertises a window size of 32KB, which can be seen in Appendix A. However, as we accumulate 45 packets with 1448 bytes of data each, we have 64KB of outstanding data. In the TCP header this field has 2 bytes, resulting in a maximum advertised window size of 64KB. We do not understand why there is no decrease in throughput after 22 = 32.000/1440 packets, as the window size should then prevent sending of more packets. As also the latency of the acknowledgement has to be taken into account, the actual amount of packets should be even less. In our sample excerpt of the tcpdump trace in Appendix A this spans 22 packets. In this excerpt, the sender has almost 100KB outstanding before he starts waiting for a few microseconds for the next ack packet. Further surprising is that this small wait does not affect the throughput.

Just from the observations in Appendix A, we know that packets were sent even though there was almost 100KB of data outstanding and therefore filling the windows. Thus, rwnd (and cwnd) must have been at least 100 KB in this state. As this is obvious for cwnd, this is not possible for the adv. window size that cannot grow past 64 KB. We do not know why this is the case, as the standard Linux TCP Vegas implementation specifically checks for this value. When default TCP Reno is used, the maximum amount of outstanding data is even increased to about 140 KB. A hacked TCP version advertising a larger window would most likely allow us to accumulate even more packets without loss in transfer rate.

4. **Network card:** The network interface card imposes a hard sending limit of 100MBit/s. As we know cwnd and rwnd are not limiting the sending rate up to the 45th (+22 in flight, see above) received data packet, it must be the network device.

These results show us that in an ideal condition in which neither cwnd nor rwnd are limiting the sending rate standard TCP implementations deal very well with accumulated acknowledgements. However, further research is crucial to investigate the effects of a TCP connection bounded by cwnd. After all, this is TCP's purpose: Flow control and congestion control, and neither of them are affecting our results in this condition.

Solving flow control will be the smaller of the problems, as it is advertised by the receiver. Congestion control however will highly depend on the TCP protocol used.

[6] Function 'tcp_snd_test' in net/ipv4/tcp_output.c

Furthermore noticeable is the 'hard' limit of 45 packets that are accumulated. As described in the end of section 5.4, the algorithm re-evaluates the optimal value every 10 seconds, but an accumulation of 46 packets results in a throughput decrease of more than 5%. This is because 45 packets (+22 in flight, see above) cause TCP to run into the minimum of cwnd and rwnd, waiting for packets to leave the network before in can inject new ones. However, the receiver will not send an acknowledgement unless it gets the 46th packet. As no ACKs are in transit, as only 22 packets can be transmitted in one RTT, the 46th packet is never sent, and the 200ms timeout for the first unacknowledged packet fires, causing an acknowledgement to be sent. Only after this acknowledgement was received the sender continues injecting packets into the network. This results in a severe performance loss of at least 5%, telling the algorithm to go back to 45 ACKs.

5.6. Comparison with NS

To validate the results gained with NS, we compare results with the same network characteristics of NS and the Linux kernel. Results show us that NS by far does not handle accumulated ACKs as well as Linux.

	NS (our TCP/norm. TCP)	Linux (our TCP/norm. TCP)
Throughput:	11.75 / 11.74 (<1%)	11.71 / 12.06 (2.9%)
Data packets /Ack:	5	45

The main reason lies in the architecture of NS. Whereas the reason for the good performance of Linux is that rwnd and cwnd never throttle bandwidth, this very well is the case for NS. In NS, the network device cannot throttle bandwidth, i.e. TCP slow start figures out at which rate to send. In our experiments with NS, we specifically set rwnd to several MB to guarantee it will never influence results. Therefore, we know it is cwnd which is limiting bandwidth, which is the inherent difference of NS results compared to Linux. Once more results are gained with the Linux kernel it is very likely that the results will match up more.

Furthermore, NS2 has a large variety of parameters for setting configuration and parameters for TCP/IP. Even though initial default values should reflect common network configurations, this must not necessarily be true, as it was for the advertised window size, described in the end of section 4.1.

6. Future work

Our TCP has been shown that it works extremely well with TCP Vegas and even with TCP Reno in Linux kernel implementation if neither the congestion nor the receiver's window size is limiting throughput. It is crucial to evaluate how our approach once this restriction is relaxed. We also hope this closes the gap to NS2, thereby supporting and validating its results.

It would further be interesting if it is possible to combine our work with more TCP protocols, such as Fast TCP or XCP. These protocols are designed for high bandwidth-delay product networks and use congestion avoidance mechanisms, eliminating TCP's saw-tooth transmission behavior.

So far, our experiments are done under, one way or another, ideal network configuration such as very low loss rate and no contention with other TCP flows. Only basic contention tests (section 4.4) and simple loss tests (section 4.3) have been conducted. To overcome this limitation, extensive experiments and analysis even over Wide Area Network environments or Internet-wide networks are required. In addition, as mentioned in Section 4.6, in long-latency networks, it is not clear how well our TCP works. Therefore, more investigation on long latency network needs to be done.

Modelnet can and will be a useful tool in this research, as it provides a flexible testing environment.

Final possible future work would be theoretical modeling. Rather than using latency and ACK cost as an objective function, we can remodel the objective function so that it can reflect, as parameters, throughput changes and total number of reduced ACK in the objective function.

7. Conclusion

We have introduced a new TCP modification, combining advanced, queuing delay based TCP implementations with our TCP acknowledgement mechanism that accumulates redundant ACKs with negligible throughput decrease.

In simulation results, we achieved equal fairness, while cutting down acknowledgements by about 70% uncongested, high bandwidth, low latency environments and still maintaining 97% of the throughput. By setting our activation threshold the user can define how much bandwidth to sacrifice for how many acknowledgements. We were able to detect lossy and unstable network environments and to use this information to fall back to a regular acknowledgement policy. As this can be achieved without a modification on the sender, gradual deployment in real environments is possible.

In Linux Kernel implementations we were able to show that in local area network environments 93% of all acknowledgement can be eliminated while not affecting throughput at all (<1%). Further experiments will provide more information about performance in environments in which the TCP congestion window and/or the advertised window size is limiting throughput. We were further able to show that these modifications require negligible additional computation cost per packet, enabling wide deployment.

8. Acknowledgements

We would like to thank Jeff LeFreve, Robert Chen and To-Ju Huang for support with Linux Kernel hacking and the Linux TCP stack.

Further Mikhail Afanasyev is to thank for his relentless help during all hours of the night with the Linux Kernel and general architecture advice.

Finally, Calivin Hubble was very eager to help with Modelnet for Linux. Even though it turned out after several hours that we will not have enough time to complete the test runs, the knowledge gained will be useful for the upcoming tests with modelnet.

9. References

1. C. Jin, D.X. Wei, S.H. Low. *"FAST TCP: Motivation, Architecture, Algorithms, Performance"* in Proceedings of IEEE INFOCOM, March 2004. http://citeseer.ist.psu.edu/jin04fast.html

2. D. Katabi, M. Handley, C. Rohrs. *"Congestion control for high bandwidth-delay product networks"* in Proc. ACM Sigcomm, August 2002.

3. L. S. Brakmo, L. L. Peterson. *"TCP Vegas: End to end congestion avoidance on a global internet."* IEEE Journal on Selected Areas in Communication, 13(8):1465--1480, October 1995.

4. V. Jacobson. *"Congestion Avoidance and Control"*, Proc. of SIGCOMM '88, ACM, Aug. 1988.

5. A.R. Karlin, C. Kenyon and D. Randall. "*Dynamic TCP acknowledgement and other stories about e/(e-1)*". Proc. 31st ACM Symposium on Theory of Computing , 502-509, 2001.

6. D.R. Dooly, S.A. Goldman, and S.D. Scott. "*TCP dynamic acknowledgement delay: Theory and practice*." Proc. 30th Annual ACM Symposium on Theory of Computing , 389-398, 1998.

7. J. Edmonds, S. Datta, and P. W. Dymond. "*Tcp is competitive against a limited adversary*." In Proc. 15th Symp. on Parallel Algorithms and Architectures (SPAA), pages 174--183. ACM, 2003.

8. Brown, K. and Singh, S. 1997. M-TCP: TCP for mobile cellular networks. SIGCOMM Comput. Commun. Rev. 27, 5 (Oct. 1997), 19-43 DOI= http://doi.acm.org/10.1145/269790.269794.

9. Wetherall, D. *"Active network vision and reality: lessons from a capsule-based system."* 17th ACM Symposium on Operating Systems Principles, Operating Systems Review, 34(5):64-79, 1999.

10. M. Mathis, J. Mahdavi, S. Floyd, "*TCP Selective Acknowledgment Options*", RFC 201.

11. Braden, R., "Requirements for Internet Hosts -- Communication Layers," STD 3, RFC 1122, October 1989.

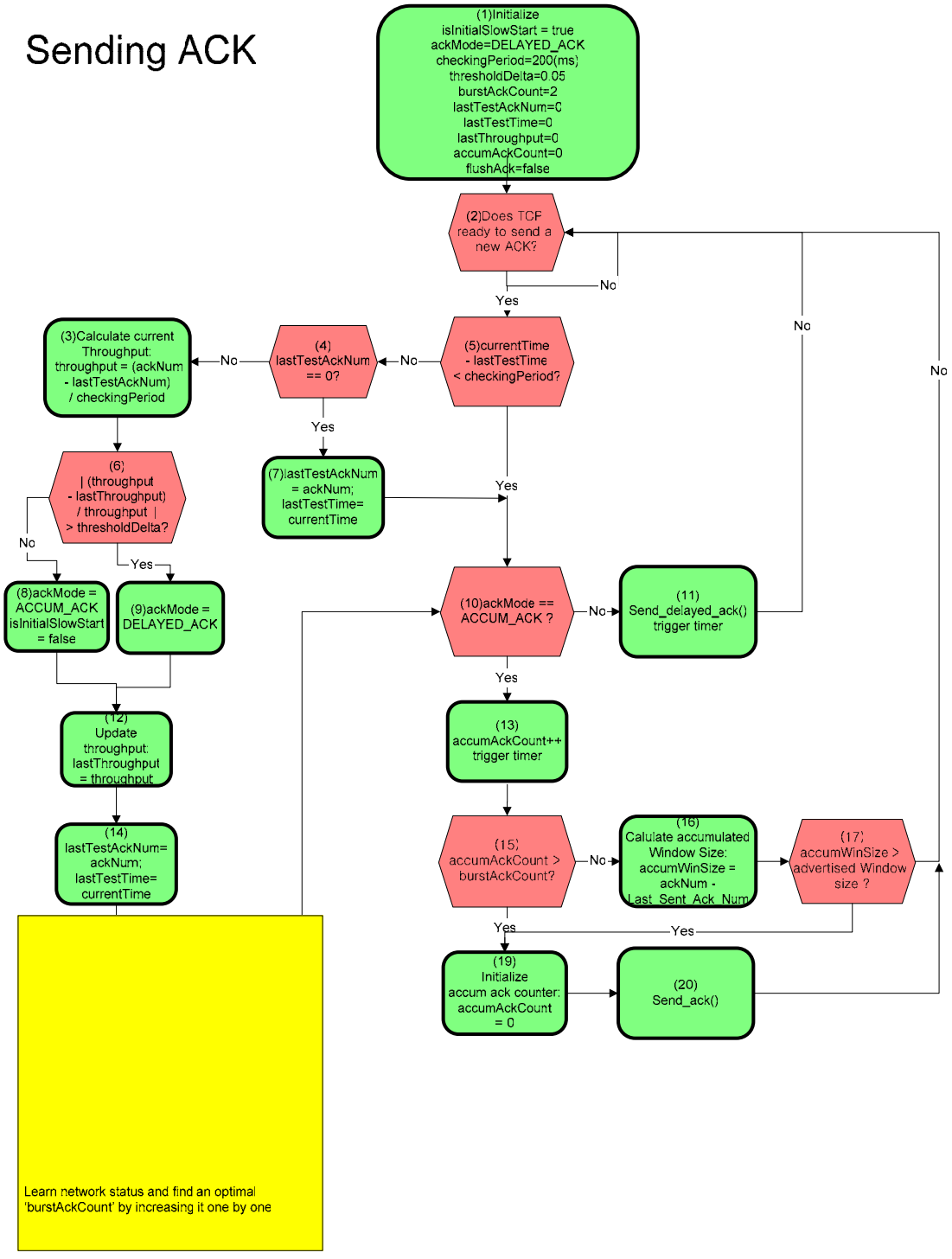
12. The Linux Documentation Project, Loadable Kernel Modules. http://tldp.org/HOWTO/Module-HOWTO/

13. Robert Chen, To-Ju Huang and Jeff LeFreve, "TCP for Wireless Connections", CSE222a class project, Winter 2006. University of California, San Diego.

Appendix A: Sample `tcpdump` trace file. Traffic recorded at the sender.

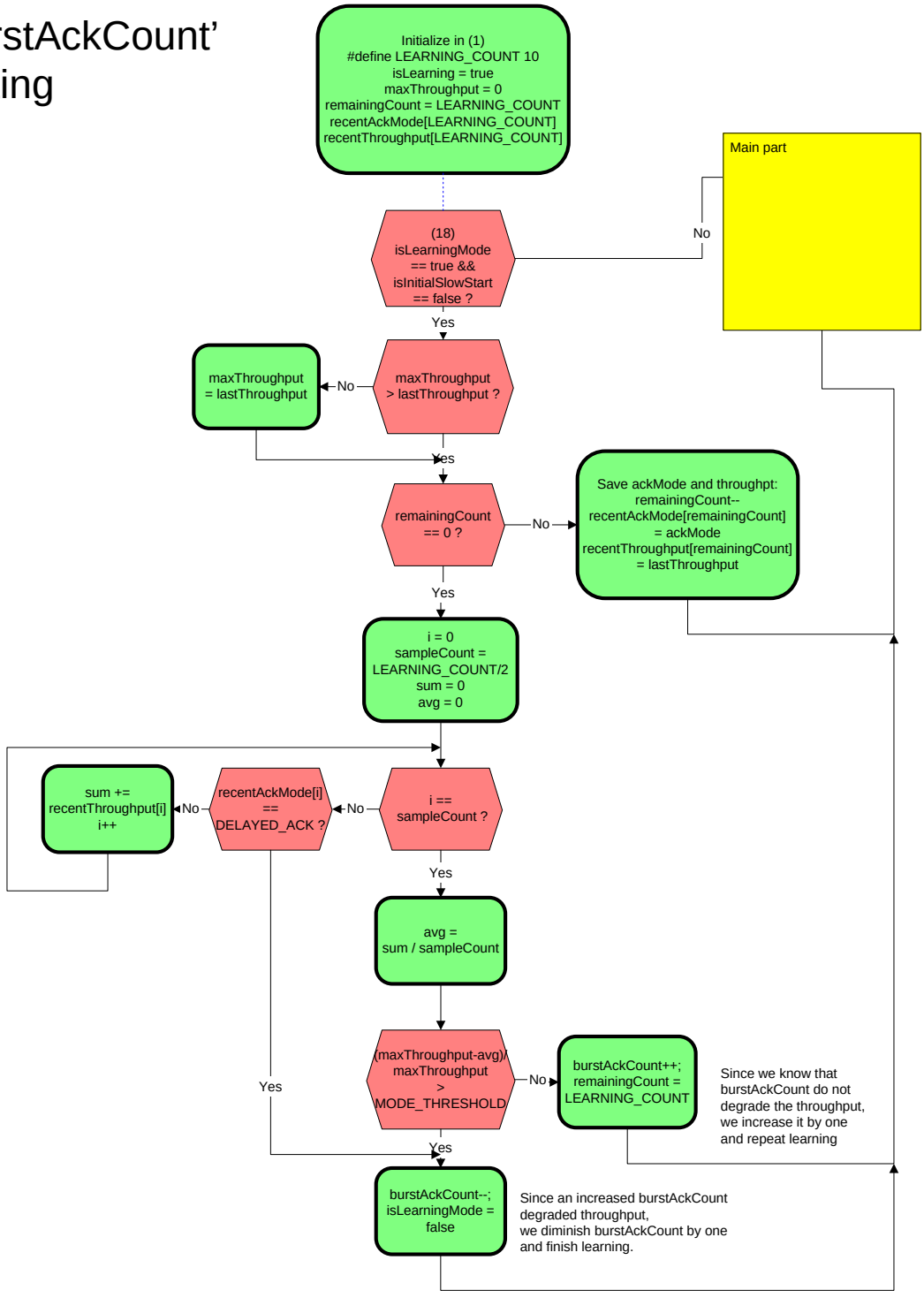
```
//Format: Timestamp(hh:mm:ss) IP source > dest tcp-flags <data-range (data)> packed-acked adv.window-size <tcp options>
05:02:47.433619 IP sender.chargen > receiver.50396: P 248803:250251(1448) ack 32 win 1448 <nop,nop,timestamp 31911526 2966386>
05:02:47.433657 IP sender.chargen > receiver.50396: . 250251:251699(1448) ack 32 win 1448 <nop,nop,timestamp 31911526 2966386>
05:02:47.433696 IP sender.chargen > receiver.50396: . 251699:253147(1448) ack 32 win 1448 <nop,nop,timestamp 31911526 2966386>
05:02:47.437353 IP receiver.50396 > sender.chargen: . ack 225635 win 31132 <nop,nop,timestamp 2966387 31911520>
05:02:47.437386 IP sender.chargen > receiver.50396: P 258939:258983(44) ack 32 win 1448 <nop,nop,timestamp 31911530 2966387>
05:02:47.437426 IP sender.chargen > receiver.50396: . 258983:260431(1448) ack 32 win 1448 <nop,nop,timestamp 31911530 2966387>
05:02:47.437465 IP sender.chargen > receiver.50396: . 260431:261879(1448) ack 32 win 1448 <nop,nop,timestamp 31911530 2966387>
05:02:47.437502 IP sender.chargen > receiver.50396: . 261879:263327(1448) ack 32 win 1448 <nop,nop,timestamp 31911530 2966387>
05:02:47.437541 IP sender.chargen > receiver.50396: . 263327:264775(1448) ack 32 win 1448 <nop,nop,timestamp 31911530 2966387>
05:02:47.437583 IP sender.chargen > receiver.50396: . 264775:266223(1448) ack 32 win 1448 <nop,nop,timestamp 31911530 2966387>
05:02:47.437623 IP sender.chargen > receiver.50396: . 266223:267671(1448) ack 32 win 1448 <nop,nop,timestamp 31911530 2966387>
05:02:47.437660 IP sender.chargen > receiver.50396: . 267671:269119(1448) ack 32 win 1448 <nop,nop,timestamp 31911530 2966387>
05:02:47.437699 IP sender.chargen > receiver.50396: . 269119:270567(1448) ack 32 win 1448 <nop,nop,timestamp 31911530 2966387>
05:02:47.437737 IP sender.chargen > receiver.50396: . 270567:272015(1448) ack 32 win 1448 <nop,nop,timestamp 31911530 2966387>
05:02:47.437777 IP sender.chargen > receiver.50396: . 272015:273463(1448) ack 32 win 1448 <nop,nop,timestamp 31911530 2966387>
05:02:47.437819 IP sender.chargen > receiver.50396: . 273463:274911(1448) ack 32 win 1448 <nop,nop,timestamp 31911530 2966387>
05:02:47.437857 IP sender.chargen > receiver.50396: . 274911:276359(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.437896 IP sender.chargen > receiver.50396: . 276359:277807(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.437933 IP sender.chargen > receiver.50396: . 277807:279255(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.437997 IP sender.chargen > receiver.50396: . 279255:280703(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.438044 IP sender.chargen > receiver.50396: . 280703:282151(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.438081 IP sender.chargen > receiver.50396: . 282151:283599(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.438119 IP sender.chargen > receiver.50396: . 283599:285047(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.438156 IP sender.chargen > receiver.50396: . 285047:286495(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.438221 IP sender.chargen > receiver.50396: . 286495:287943(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
//This is the last packet that the ACK refers to
05:02:47.438258 IP sender.chargen > receiver.50396: . 287943:289391(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.438297 IP sender.chargen > receiver.50396: . 289391:290839(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.438336 IP sender.chargen > receiver.50396: . 290839:292287(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.438373 IP sender.chargen > receiver.50396: . 292287:293735(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.438413 IP sender.chargen > receiver.50396: . 293735:295183(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.438451 IP sender.chargen > receiver.50396: . 295183:296631(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.438489 IP sender.chargen > receiver.50396: . 296631:298079(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.438526 IP sender.chargen > receiver.50396: . 298079:299527(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.438564 IP sender.chargen > receiver.50396: . 299527:300975(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.438631 IP sender.chargen > receiver.50396: . 300975:302423(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.438668 IP sender.chargen > receiver.50396: . 302423:303871(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.438707 IP sender.chargen > receiver.50396: . 303871:305319(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.438743 IP sender.chargen > receiver.50396: . 305319:306767(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.438781 IP sender.chargen > receiver.50396: . 306767:308215(1448) ack 32 win 1448 <nop,nop,timestamp 31911531 2966387>
05:02:47.438824 IP sender.chargen > receiver.50396: . 308215:309663(1448) ack 32 win 1448 <nop,nop,timestamp 31911532 2966387>
05:02:47.438862 IP sender.chargen > receiver.50396: . 309663:311111(1448) ack 32 win 1448 <nop,nop,timestamp 31911532 2966387>
05:02:47.438899 IP sender.chargen > receiver.50396: P 311111:312559(1448) ack 32 win 1448 <nop,nop,timestamp 31911532 2966387>
05:02:47.438938 IP sender.chargen > receiver.50396: . 312559:314007(1448) ack 32 win 1448 <nop,nop,timestamp 31911532 2966387>
05:02:47.438976 IP sender.chargen > receiver.50396: . 314007:315455(1448) ack 32 win 1448 <nop,nop,timestamp 31911532 2966387>
05:02:47.439038 IP sender.chargen > receiver.50396: . 315455:316903(1448) ack 32 win 1448 <nop,nop,timestamp 31911532 2966387>
05:02:47.439077 IP sender.chargen > receiver.50396: . 316903:318351(1448) ack 32 win 1448 <nop,nop,timestamp 31911532 2966387>
05:02:47.439113 IP sender.chargen > receiver.50396: . 318351:319799(1448) ack 32 win 1448 <nop,nop,timestamp 31911532 2966387>
//At the time this following packet is transmitted, the sender has 22+46 = 68 packets and 96KB outstanding.
05:02:47.439151 IP sender.chargen > receiver.50396: . 319799:321247(1448) ack 32 win 1448 <nop,nop,timestamp 31911532 2966387>
//Ack comes in for data #. In the meantime, 22 more packets have been transmitted.
05:02:47.442822 IP receiver.50396 > sender.chargen: . ack 289391 win 31132 <nop,nop,timestamp 2966388 31911526>
05:02:47.442852 IP sender.chargen > receiver.50396: P 321247:321291(44) ack 32 win 1448 <nop,nop,timestamp 31911536 2966388>
05:02:47.442900 IP sender.chargen > receiver.50396: . 321291:322739(1448) ack 32 win 1448 <nop,nop,timestamp 31911536 2966388>
```

Appendix B


Sending ACK
(1)Initialize
isInitialSlowStart = true
ackMode=DELAYED_ACK
checkingPeriod=200(ms)
thresholdDelta=0.05
burstAckCount=2
lastTestAckNum=0
lastTestTime=0
lastThroughput=0
accumAckCount=0
flushAck=false
(2)Does TCP ready to send a new ACK?
No
Yes
(5)currentTime - lastTestTime < checkingPeriod?
No
(4) lastTestAckNum == 0?
No
(3)Calculate current Throughput: throughput = (ackNum - lastTestAckNum) / checkingPeriod
Yes
(7)lastTestAckNum = ackNum; lastTestTime= currentTime
Yes
(6) | (throughput - lastThroughput) / throughput | > thresholdDelta?
No
Yes
(8)ackMode = ACCUM_ACK isInitialSlowStart = false
(9)ackMode = DELAYED_ACK
(10)ackMode == ACCUM_ACK ?
No
(11) Send_delayed_ack() trigger timer
No
Yes
(12) Update throughput: lastThroughput = throughput
(13) accumAckCount++ trigger timer
(14) lastTestAckNum= ackNum; lastTestTime= currentTime
(15) accumAckCount > burstAckCount?
No
(16) Calulate accumulated Window Size: accumWinSize = ackNum - Last_Sent_Ack_Num
(17) accumWinSize > advertised Window size ?
No
Yes
Yes
(19) Initialize accum ack counter: accumAckCount = 0
(20) Send_ack()
Learn network status and find an optimal 'burstAckCount' by increasing it one by one

Appendix C

‘burstAckCount’ setting



Appendix D

Comparison of a TCP Reno and TCP Vegas sender and a modified receiver; Test settings as described in 5.3.