Solving Ack Inefficiencies in 802.11 Networks

David Murray
Murdoch University
D.Murray@murdoch.edu.au

Terry Koziniec
Murdoch University
T.Koziniec@murdoch.edu.au

Michael Dixon
Murdoch University
M.Dixon@murdoch.edu.au

***Abstract*—The founding idea behind this study was that 802.11 acks and TCP acks are substantial contributors to 802.11 overheads, yet, they both provide the same functionality; reliability. Initial experiments suggest that 802.11 acks contribute to over 20% of the overhead in 802.11 networks. Unfortunately, without 802.11 acks, paths with RTTs greater than a millisecond are unable to utilise this additional performance because lost packets, which occur frequently in unacknowledged (NoAck) 802.11, are interpreted as congestion. This study experiments with a range of PEPs (Performance Enhancing Proxies) which retransmit lost packets. A new proxy, known as D-Proxy, designed to solve the shortcomings of previous I-TCP and Snoop proxies, is experimentally developed and tested in Linux. D-Proxy is a distributed, proactive proxy that caches, analyses and resends packets based on TCP sequence numbers. The results suggest that D-Proxy can substantially improve 802.11 throughputs.**

I. Introduction

This study presents a cross layer approach to increasing performance in multi-hop ad-hoc networks. It proposes that the reliability function provided by 802.11 acks is replicated by TCP's end-to-end reliability. The potential performance gains are demonstrated through modifications to the MadWiFi driver. However, these gains diminish with increasing RTTs necessitating other mechanisms to hide end-to-end losses.

A. 802.11 Overheads

To justify this work we initially wish to show the extent of inefficiencies imposed by current ack mechanisms in 802.11. 802.11a/g can transmit data at 54 Mb/s. Unfortunately, real world TCP throughputs are roughly half this number; 27 Mb/s. Actual throughputs are approximately half the data rate because of a combination of MAC layer contention delays, packet headers and TCP acks. This study firstly determines how much of this overhead is consumed by MAC layer acks.

In wired networks, transmissions are reliable and packet loss usually only occurs as a result of congestion. However, 802.11 is inherently unreliable, and consequently, every packet is acknowledged by an 802.11 ack. We use the terms MAC layer, link layer and 802.11 ack interchangeably. These acknowledgements provide a reliable 'Ethernet like' MAC layer and prevent adverse reactions from TCP. Currently, both the TCP data segment and TCP ack are individually acknowledged by MAC layer acks. The reliability replication that occurs between TCP acks and 802.11 acks is the starting point of our investigation.

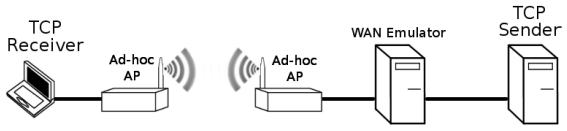

Fig. 1. Testing setup

B. Real World NoAck Performance

Physical experiments were performed to determine the effect of removing MAC layer acks from 802.11. Using Linux, we were able to modify the MadWiFi driver (version 9.3.3) such that the interface would neither transmit an ack nor wait for an ack following the reception of a data frame. With newer 802.11e capable drivers, removing acks is simple and standards compliant with the 802.11e NoAck mechanism.

Standard 802.11 and NoAck 802.11 were initially compared over a single 802.11 link. The results suggest that NoAck 802.11a allows a peak transmission capacity of 33.5 Mb/s while standards based 802.11a had a throughput of 27.4 Mb/s. These results show that the overhead of 802.11 acks is 22%. This additional performance is available because MAC layer acks are no longer sent, providing additional time for data transmission.

Unfortunately, these results are somewhat misleading and link layer acks were implemented in 802.11 for good reason. TCP, which provides flow control and error recovery, interprets packet loss as congestion. Each lost packet results in the congestion window being halved. Therefore, a single packet loss can have a significant affect on the TCP congestion window. The afore mentioned results were obtained using two directly connected nodes with a sub 1 ms end-to-end latency. To add latency we inserted a WAN emulator and shifted the TCP sender/receiver off the APs and onto dedicated machines. We then performed the same tests over a range of latencies using the testing topology shown in Fig 1. This topology was used for all of the subsequent tests in this study.

The results obtained from this second round of tests showed that increasing end-to-end latency sharply degrades the throughput of unacknowledged or NoAck 802.11 transmissions. Running TCP over lossy unacknowledged 802.11 is so delay sensitive that simply moving the TCP endpoints from directly connected APs and onto separate machines and adding a 100 Mb/s Ethernet bridge was enough to reduce TCP throughputs from 33.5 Mb/s to 25.5 Mb/s. When additional latency was added, throughputs quickly dropped to a few

978-1-4244-4793-0/09/$25.00 ©2009 IEEE

megabits per seconds. While NoAck 802.11 transmissions may be efficient in terms of channel usage, requiring less channel transmission time, transfers complete more slowly due to underutilisation.

TCP throughputs degrade so rapidly with increasing latency because dropped packets are interpreted as congestion, causing the TCP sender to half the congestion window. Large RTTs are the catalyst for two reasons. Firstly, larger RTTs will require more packets to be released unacknowledged from the TCP sender. Secondly, paths with larger RTTs, will recover lost packets more slowly. As a result, the congestion window will transmit all of the packets allowed to be outstanding and the missing packet will take longer to be recovered. Based on these results, link layer reliability is a necessary overhead for TCP networks. Perhaps an idyllic solution to this problem is to move from the current system of positively acknowledging successful packets to negatively acknowledging unsuccessful packets. Such a system would provide reliability and incur minimal overhead.

C. Positive and Negative Acknowledgements

Currently, the 802.11 ack system is a positively acknowledging system. For every successful transmission, whether TCP data or TCP ack, a MAC layer ack is sent. In the existing positive ack scheme, nodes use the absence of an ack to indicate a loss. A better approach, if possible, would be to negatively acknowledge packets. Put simply, instead of transmitting acks for every packet correctly received, why not instead, transmit a negative ack for every packet not correctly received.

Currently, this is impossible in 802.11 because the radios cannot send and receive simultaneously. This means that a radio cannot hear if its transmission is being corrupted as it is sent. Furthermore, packet corruption generally happens at the receiver side not the sender side and therefore, even if the sender could send and receive simultaneously, its assessment of transmission success may differ from the receiver's transmission success.

This study investigates solutions to the ack inefficiency of 802.11 with a focus on multi hop ad hoc networks. We consider solutions from multiple networking layers. Although there is no obvious way to turn 802.11 into a negatively acknowledging system at the MAC layer, IEEE standards based work is addressing inefficiencies in the existing ack scheme. These MAC layer efficiency mechanisms are our next topic of discussion.

II. MAC Layer Solutions

A. 802.11e BlockAck

The IEEE 802.11e BlockAck function enables senders to transmit multiple packets before requiring a MAC layer ack. When packets are acknowledged, a single BlockAck is sent rather than an individual acknowledgement for each packet, significantly reducing the ack overhead. While there are two types of block acknowledgements, immediate BlockAck and delayed BlockAck, for brevity we only consider the more efficient immediate BlockAck. Using BlockAcks has a number of benefits. The reduction in the number of acknowledgements transmitted is the most obvious. Fig 2 demonstrates the difference between standard 802.11 and BlockAck 802.11. Note that the IFS (Inter-Frame Spacing) between the data frames in BlockAck in Fig 2 is SIFS (Short IFS) rather than DIFS (DCF IFS). Subsequently, the use of BlockAck saves on IFS as well as the number of transmitted acks.

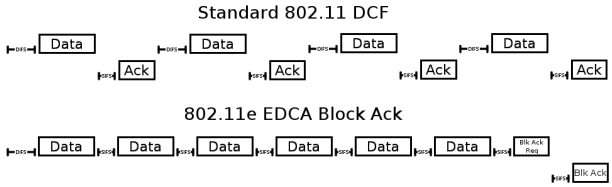

Fig. 2. Standard 802.11 DCF vs 802.11e BlockAck

The protocol operation is relatively simple; the number of frames that can be sent in a block is defined by the AP during the BlockAck transmission setup. In the case of immediate BlockAck, when the sender has finished transmitting a block of packets, it will send a BlockAck request to the receiver. The receiver will then respond with a BlockAck to acknowledge the successfully received frames. Any frames sent, but not acknowledged will be resent as part of the next block of transmissions. If a frame is lost, the receiver will buffer all the received packets until the lost frames have been recovered. Buffering of packets is done so that packets can be passed to the upper layers in order. Some studies suggest that for bulk transfers with large packet sizes, the performance benefit is approximately 10% [1].

1) 802.11 BlockAck: Delay vs Efficiency: Most academic work suggests that BlockAcks increase the throughput but also express concern over the number of packets transmitted in a block because of the affect on time sensitive traffic [1], [2], [3], [4]. Obviously it is more efficient to transmit a larger number of packets in a block because larger blocks mean that fewer layer 2 acks must be transmitted. However, the problem with transmitting larger blocks is delay. The larger block sizes, which are most efficient, may not facilitate fair and jitter free networks. Cabral et al [1] believes that due to the delays added by BlockAcks the optimal block size is 12 to 16 packets. Prior studies [1], [2], [3], [4] have all either mathematically modelled or simulated BlockAck, providing extensive understanding of link layer performance benefits. To our knowledge, no study has practically investigated the protocol on a real network.

2) 802.11 BlockAck: An Unexplored TCP Interaction: TCP has a self clocking mechanism whereby returning TCP acknowledgements prompt the release of new data segments onto the medium. TCP's goal is to recover errors and facilitate the fastest and fairest data transport. To do this, the release of TCP data packets onto the medium should be smooth and paced. Excessive bursts are problematic as they may suddenly increase router queue sizes causing packet drops. A lot of work has gone into reducing the natural traffic bursts caused by TCP

[5], often referred to as ack compression [5]. Sundaresan [6] made specific reference to this problem with TCP and expects it to worsen in ad hoc networks where media contention is greater.

We also question the extent to which the use of BlockAck may further degrade this natural effect by compressing data packets arriving in bunches. When TCP receivers experience large numbers of back-to-back TCP segments in blocks, the subsequent TCP acks will also be generated and transmitted in blocks. Rather than the natural multiplexing that occurs in current DCF where after every two TCP data segments received a TCP ack is transmitted, with large block sizes, many data packets will be received which may be replied to by blocks of back-to-back acks. This will increase ack compression which may degrade end-to-end performance in high bandwidth high RTT environments. TCP senders receiving compressed TCP acks can either dump large numbers of packets onto the network, suddenly increasing router buffers and the likely-hood of queuing losses, or, pace the transmission of data packets more evenly and smoothly, but at the price of additional delay. We believe that this issue is worthy of further study. To conclude, 802.11 BlockAck may increase performance by 10% [1], however, these efficiency gains may come at the cost of other network goals.

III. PEP (Performance Enhancing Proxies)

An alternative to using MAC layer acks is to use a PEP (Performance Enhancing Proxy) to provide reliability. PEPs are designed to mitigate link related degradations and are discussed in a dedicated RFC 3135 [7] as well as RFC 3449 [5]. In this section we will review I-TCP and Snoop as well as our own distributed PEP specifically designed for the goal of ack efficiency in 802.11 multi hop ad hoc networks.

A. I-TCP

Indirect TCP [8], also known as a split TCP [9], splits a TCP connection in two. By capturing the SYN and SYN-ACK TCP segments, the proxy can imitate each side of the transaction. The main advantage being a large reduction in TCP perceived RTTs. Real end-to-end latencies will be the same, however, by splitting the link, the TCP sender can receive acks more quickly, building the congestion window faster and thereby completing faster. In satellite networks, large RTTs impair performance due to the time required to build the TCP congestion window and therefore, I-TCP is often deployed on the TCP sender side of a satellite link to provide faster growth.

As shown in Fig 3 we deploy an I-TCP proxy on the reliable side of the 802.11 link. With the PEP deployed close to the TCP receiver, the affect of packet loss on TCP congestion windows become irrelevant because even small TCP congestion windows will likely fill the link. The use of I-TCP in this manner will also hide link losses from the real TCP sender. Daniele Lacnamara's [9] PEPsal provided an implementation to test Split TCP. One problem, discussed later in more detail, is that I-TCP breaks TCP end-to-end semantics.

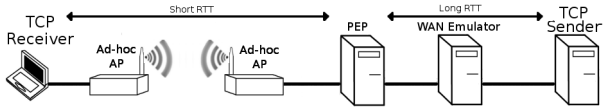

Fig. 3. Proposed use of PEPs in WLANs

B. Snoop

The Snoop proxy idea [10], [7], [11], [12], [13], [14], first envisaged by Balakrishnan [10], was that an intermediate device could cache packets and retransmit them upon reception of TCP dup acks. While TCP ack numbers are increasing, Snoop assumes that the transfer is in good state. When TCP dup acks are seen, the Snoop proxy searches its local cache and if found, retransmits the lost segment. Dup acks are filtered to hide losses from the TCP sender

After the lost packet is replayed, the sequence can continue. While Snoop proxies do not break TCP semantics, they miss a number of benefits provided by split-TCP proxies. Firstly, the TCP congestion window is still end-to-end and therefore must grow larger than I-TCP where window sizes are insignificant given a small enough RTT on the unreliable side. Thus Snoop proxies are dependent on their ability to hide link losses from the sender, however, there are a number of studies that suggest issues with SACK and Snoop [14], [12], [13]. These studies suggest that Snoop is unable to completely hide losses from the TCP receiver, especially in the presence of burst losses.

C. D-Proxy

D-Proxy is a new proactive distributed TCP proxy designed to overcome the limitations of Snoop and I-TCP. D-Proxy is distributed because it uses a proxy either side of the lossy link. It is proactive because instead of waiting for TCP acks to confirm packet loss, the proxy on the lossy side of the data stream analyses TCP data sequence numbers. If the sequence number received is greater that the sequence number expected then it is assumed that there may be one or more missing packets. A request is then sent to the previous proxy to ask for a retransmission. Fig 4 shows the basic operation of D-proxy. Note that the missing packet was discovered because 5792 was sent when 4344 was expected. When a loss is detected, D-proxy buffers frames until the lost segment can be replayed and reorganises them such that they are put back in sequence.

D-Proxy maintains TCP state information and each flow is differentiated based on source IP, destination IP, source port and destination port. The individual packets being cached are identified within their flow based on sequence number. We implemented D-Proxy in Linux using the ip_queue library which passes packets from kernel space to user space for processing.

1) Inter proxy communication: The speed at which, D-proxy can recognise and reliably resend lost packets is of utmost importance. It is anecdotally recognised that wireless losses occur in bursts which makes inter proxy communication troublesome. What makes this problematic is the likelihood

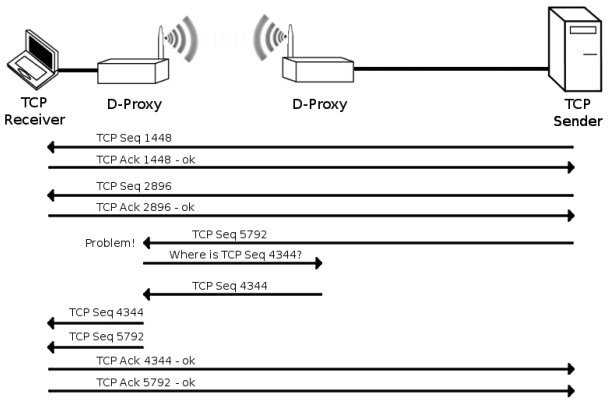

Fig. 4. Basic D-Proxy example

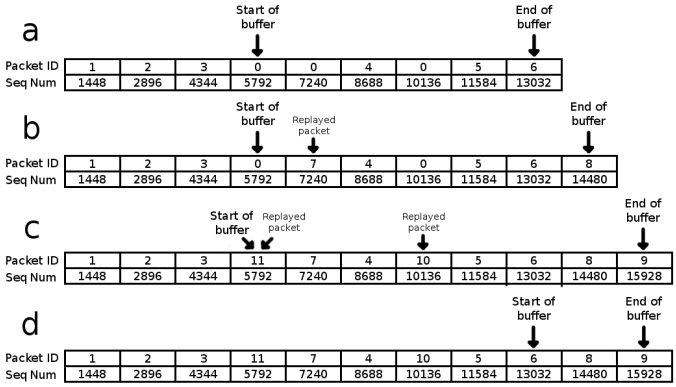

Fig. 5. Packet recovery example

that retransmission requests or the data segment being retransmitted might be lost. For these reasons, the mechanism used to request the retransmission of lost packets was critical.

We opted to use UDP sockets to re-request lost packets because they have less latency than TCP sockets. UDP may appear an odd choice because the reliability of the retransmission request messages are critical, however, we found TCP error recovery too slow for our purposes where errors must be detected and recovered on a millisecond timescale. For this reason, we decided to use UDP and implement our own fast reliability mechanisms. Note, the mechanism must provide fast two way reliability for both the UDP retransmission request and the actual data segment being sent. Two mechanisms were used to infer that either the UDP retransmission request, or, the retransmitted TCP data segment had been lost.

2) Timeouts: One mechanism was timeout based whereby if timer is exceeded the packet is re-requested. Packets will be re-requested after two, three and four times the average retransmission time. On the fourth time, the packet has been re-requested, D-Proxy will give up waiting to receive the requested packet.

The amount of time to wait for a packet to be resent before requesting retransmission is a trade-off. If a replayed packet is not lost but merely waiting in buffers, or waiting for media access, sending further retransmission requests will only create additional delays and superfluous retransmissions. Equally, waiting too long before sending further retransmission requests only increases the number of packets being buffered and the likelihood of TCP timing out.

Using the timeout based retransmission mechanism above provided benefits, however, the performance using this mechanism alone still left NoAck 802.11 below standard 802.11. The problem was that retransmission requests as well as the actual replayed TCP packets were being lost. These losses were common due to the burst loss nature of 802.11. The timeout mechanisms took too long to re-request the lost packet. Trying to reduce the timeouts further was counter productive because often packets were simply late (not lost) because they were sitting in buffers. Reducing timeout values only created additional unnecessary retransmissions. The reliability mechanism used to overcome this problem, retransmission order, is described in detail as it was the crucial feature enabling NoAck 802.11 performance with D-Proxy to surpass standard 802.11.

3) Retransmission Order: The operation of retransmission order is best understood with an example. Fig 5a shows the APs buffer on the unreliable side of the wireless link. The first three packets, 1448, 2836 and 4344 are all in sequence and are passed back to the kernel for transmission. Following 4334 there are holes between 4344-8688 and 8688-11584. Based on the premise that the next sequence number equals the previous plus the packet size, D-Proxy can ascertain that 5792 is missing and also that 10135 is missing. Also, because of the size of the hole between 4344-8688, based on the packet sizes in the flow, we can predict that 7240 is also missing. Each missing packet generates its own retransmission request. So three individual retransmission requests will be sent containing a TCP flow ID and the missing sequence number. The previous proxy will replay these cached packets.

In Fig 5b the replayed packet with seq number 7240 is received and then a new packet 14480 is received. The fact that the replayed packet 7240 was received before 5792 is used as an indicator that either the UDP request or the replayed 5792 packet was lost. As stated earlier, each missing packet is re-requested with an individual UDP packet and thus the individual retransmission requests are sent in order. If a retransmitted segment such as 7240 is received ahead of an unfilled hole such as 5792, it indicates that either the retransmission request or the retransmitted data segment was lost and a new retransmission request is generated for 5792. If packet loss was randomly distributed then this mechanism would be less effective, however, as packet losses frequently occurs in bursts, we found that this method of packet recovery was significantly faster than the timeout based system.

Similar to Fig 5a, at the current point in time, Fig 5b, no more packets are being passed back to the kernel for transmission. It is critically important to maintain the sequence order to prevent congestion avoidance at the sender.

In Fig 5c, a new packet (15928) is added to the sequence, then packet 10136 is replayed. Finally, packet 5792 which had

to be re-requested for a second time is replayed. With 5792 now in sequence, the packets can be sent to the kernel for transmission.

4) Staggered TCP catch-up: In Fig 5d, note that the buffer has only moved to packet 13032. This is done to avoid packet bursts. For example, large number of packets can be queued waiting on the retransmission of one packet. When the missing packet is replayed we want to avoid replaying 30 or more straight packets from the same TCP flow as this could be to the detriment of other flows waiting in the queue. Instead we send a maximum of five packets before rechecking the input buffer. The purpose of this mechanism is to stagger the catch-up such that other TCP flows are not adversely affected.

5) Variable per flow buffer size: Similar to normal router buffers, more buffering increases latency, slowing the reaction to congestion. D-Proxy has a buffer size of 150 packets. If there is only one data flow, that flow will have a buffer size of 150 packets. With the addition of more flows, the buffer size is divided equally between all flows. Therefore, if there are 5 flows, each flow will have a buffer of 30 packets. Flows may exceed their buffer size, for example, if there is one flow utilising a buffer of 150 packets and then 2 additional flows are quickly added reducing the buffer size to 50 packets per flow, the 100 packets that exceed the buffer size will continue being processed. The holes within this extended buffer will not be filled and retransmitted segments will not be re-sequenced until the buffer size has dropped below the allowed level. Retransmission requests will have already been sent for the holes, however, D-Proxy will no longer wait to reorder packets exceeding the buffer size. This function fortuitously causes congestion avoidance at the sender and TCP will slow down to accommodate the two new flows.

D. Results

Fig 6 and Fig 7 present the results of the reliable and unreliable link tests. Each test used five TCP flows which downloaded a 144MB file from an Apache web server. The fair link was created by locking the speed at 54 Mb/s and moving the APs past the point where they would normally rate shift. Single flow tests were also performed however the results were similar to the multi flow tests so they have been excluded for brevity.

1) Snoop Results: Snoop proxies increase performance in standard 802.11 networks, however, both our study and numerous other recent studies [14], [12], [13] have suggested that Snoop cannot completely hide losses from TCP senders. Our results, Fig 6 and Fig 7, suggest that Snoop is not suitable as a performance enhancement with NoAck 802.11. Furthermore the implementation of Snoop in multi hop ad hoc networks is problematic. For example, along a five hop wireless path, a packet could be lost at any stage. If packet loss occured on the first hop, the packet will not be recovered until the dup ack is re-received on the first wireless router. This will substantially delay the detection of loss and increase the incidence of TCP timeouts.

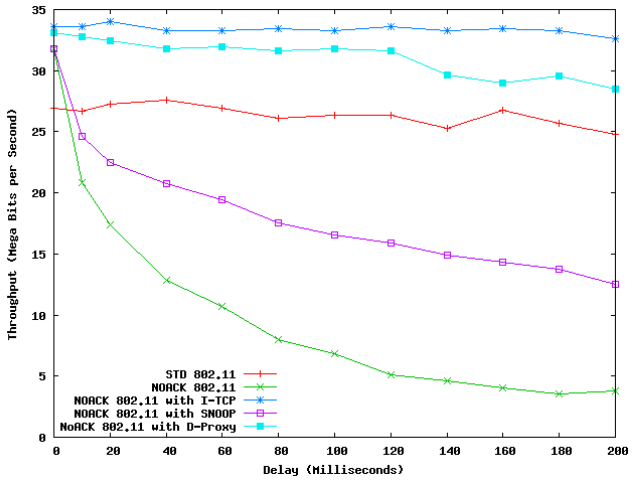

Fig. 6. Reliable 802.11 link

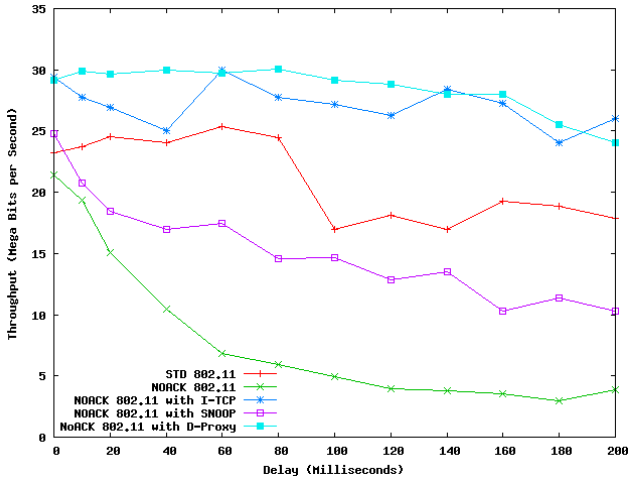

Fig. 7. Unreliable 802.11 link

2) I-TCP Results: The I-TCP results show that the performance of I-TCP with NoAck 802.11 exceeds standard 802.11. With an I-TCP proxy deployed as close as possible to the TCP receiver, large congestion windows are unnecessary because of the low RTT. On the long RTT side of the proxy, the wireless losses are shielded from the real TCP sender. The use of a I-TCP in this manner achieves our goal of a negatively acknowledging system as dup acks are the only feedback mechanism used to indicate loss.

As I-TCP has been in use for over a decade, it is important to question why it is not standard in WiFi implementations. I-TCP is only recommended for specific cases because it breaks TCP's end-to-end semantics. When a TCP ack is received to a TCP data segment, the TCP sender assumes that the data has been received correctly. I-TCP may ack segments before the real TCP receiver has received the data. In the case of a link failure between TCP receiver and I-TCP, the TCP sender may have received acknowledgements for data that was never actually received. This is the main argument against I-TCP; it

breaks TCP's end-to-end semantics. Many security protocols such as SSH do not work through I-TCP. A more thorough treatment of these issues is given in RFC 3135 [7].

Implementations for ad-hoc networks are also problematic. Mindful that low RTTs are quintessential to performance over the unreliable link, in multi hop ad hoc networks many I-TCP proxies will be required throughout the network. As so many nodes are "pretending" to be endpoints, end-to-end transactions will fail with a single path/routing change.

3) D-Proxy Results: The inapplicability of I-TCP and Snoop proxies in multi hop ad hoc networks, provided the impetus for a fresh approach. We started with the notion that an ideal proxy should not break end-to-end TCP semantics but should also be more proactive than Snoop in its approach to detecting packet losses. The solution, D-Proxy, has been shown to provide superior performance to standard 802.11 without the drawbacks of I-TCP or Snoop.

D-Proxy does not break TCP end-to-end semantics but does hide link losses from the TCP receiver and can be implemented in multi-hop ad hoc networks. There are two current drawbacks of D-Proxy, Firstly, transport layer security mechanisms such as IPSec will have to bypass the proxy and secondly, in the current implementation, CPU utilisation for the packet caching side of the proxy is too high to run on embedded systems. We believe that this can be solved by implementing D-Proxy as a kernel module.

IV. Conclusion

This study started by questioning the need for 802.11 acks. It was shown that without 802.11 acks, links with RTTs greater than 1 ms were underutilised by TCP because packet loss was interpreted as congestion. We then explained and explored IEEE standards based work attempting to address this problem. Experiments with PEPs revealed that I-TCP provided excellent performance, but had a number of limitations and implementation problems for ad hoc mesh networks. It was also shown that Snoop proxies were not suitable for hiding the link layer losses of unacknowledged 802.11. This prompted us to create a new PEP specifically designed to address the limitations in previous proxies.

The results show that D-Proxy provides good performance and can be implemented in ad hoc networks. The question remains; is performance superior to the 802.11e BlockAck function? Simulations claim that 802.11e BlockAck can improve efficiency in bulk transfers by 10% [1]. Our real life wireless tests revealed that the combination of NoAck 802.11 and D-Proxy improves transfer capacity by 21.5%.

The key difference is that BlockAck is a positively acknowledging proxy, providing reliability by acknowledging packets that have been received. D-Proxy is a negatively acknowledging proxy that uses TCP sequence numbers to decipher what is missing rather than relying on positive acknowledgements. The drawback of D-Proxy is complexity. D-Proxy must analyse, cache and reorder packets. However, BlockAck has similar requirements, somewhat nullifying this argument.

A disadvantage of BlockAck is that it groups TCP data segments and TCP acks into blocks potentially causing traffic bursts and ack compression. D-Proxy maintains the natural multiplexing of two data packets for every TCP ack, steadying the delivery of TCP acks. Also, D-Proxy groups packets based on TCP flows, not MAC layer senders so the loss of one packet will still allow packets from other flows to pass unhindered. D-Proxy also investigates whether a packet is missing after every packet whereas BlockAck performs this function at the end of every block of packets, delaying the reaction to loss. Finally, unlike BlockAck, D-Proxy does not require lengthy medium reservation and thus D-Proxy may facilitate fairer QoS in the presence of multiple transmitting nodes. We conclude that D-Proxy offers better performance, fairer QoS, is more TCP friendly, and reacts faster to lost packets. The code used for Snoop and D-Proxy will be made available at www.bridgingthelayers.org.

References

[1] Orlando Cabral, Alberto Segarra, and Fernando J Velez, "Implementation of multi-service ieee 802.11e block acknowledgement policies", *IAENG International Journal of Computer Science*, vol. 36:1, pp. 1, June 2009.

[2] Guido R Hiertz, Lothar Stibor, Jrg Habetha, Erik Weiss, and Stefan Mangold, "Throughput and delay performance of ieee 802.11e wireless lan with block acknowledgments", in *11th European Wireless Conference*, 2005.

[3] Tianji Li, Qiang Ni, and Yang Xiao, "Investigation of the block ack scheme in wireless ad hoc networks: Research articles", *Wireless Communications & Mobile Computing*, vol. 6, no. 6, pp. 877–888, 2006.

[4] Arun Ranjitkar and Young-Bae Ko, "Performance enhancement of ieee 802.11s mesh networks using aggressive block ack scheme", in *The International Conference on Information Networking*, 2008.

[5] H Balakrishnan, V N Padmanabhan, G Fairhurst, and M Sooriyabandara, "Tcp performance implications of network path asymmetry", IETF RFC 3449, December 2002.

[6] Karthikeyan Sundaresan, Vaidyanathan Anantharaman, Hung-Yun Hsieh, and Raghupathy Sivakumar, "Atp: A reliable transport protocol for ad hoc networks", *IEEE Transactions on Mobile Computing*, vol. 4, no. 6, pp. 588–603, 2005.

[7] J Border, M Kojo, J Griner, G Montenegro, and Z Shelby, "Performance enhancing proxies intended to mitigate link-related degradations", IETF RFC 3135, June 2001.

[8] Ajay Bakre and B R Badrinath, "I-tcp: Indirect tcp for mobile hosts", in *15th International Conference on Distributed Computing Systems*, 1995, pp. 136–143.

[9] Carlo Caini, Rosario Firrincieli, and Daniele Lacamera, "Pepsal: A performance enhancing proxy for tcp satellite connections", in *IEEE Aerospace and Electronic Systems, Internetworking and Resource Management in Satellite Systems Series*, Aug. 2007, vol. 22, pp. B–9–B–16.

[10] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H. Katz, "Improving tcp/ip performance over wireless networks", in *1st Annual International Conference on Mobile Computing and Networking*, New York, NY, USA, 1995, pp. 2–11, ACM.

[11] Chi Ho Ng, Jack Chow, and Ljiljana Trajkovic, "Performance evaluation of tcp over wlan 802.11 with the snoop performance enhancing proxy", in *In Proceedings of OPNETWORK*, 2002, pp. 134–142.

[12] Sarma Vangala and Miguel A. Labrador, "The tcp sack-aware snoop protocol for tcp over wireless networks", in *IEEE Vehicular Technology Conference*, 2002.

[13] Fanglei Sun, Victor O.K. Li, and Soung C. Liew., "Design of snack mechanism for wireless tcp with new snoop", in *Wireless Communications and Networking Conference*, March 2004, vol. 2, pp. 1051–1056.

[14] Jaehoon Kim and Kwangsue Chung, "C-snoop: Cross layer approach to improving tcp performance over wired and wireless networks", *International Journal of Computer Science and Network Security*, vol. 7(3), pp. 131–137, 2007.