HACK: Hierarchical ACKs for Efficient Wireless Medium Utilization

Lynne Salameh, Astrit Zhushi, Mark Handley, Kyle Jamieson, Brad Karp
University College London

Abstract

WiFi's physical layer has increased in speed from 802.11b's 11 Mbps to the Gbps rates of emerging 802.11ac. Despite these gains, WiFi's inefficient MAC layer limits achievable end-to-end throughput. The culprit is 802.11's mandatory idle period before each medium acquisition, which has come to dwarf the duration of a packet's transmission. This overhead is especially punishing for TCP traffic, whose every two data packets elicit a short TCP ACK. Even frame aggregation and block link-layer ACKs (introduced in 802.11n) leave significant medium acquisition overhead for TCP ACKs. In this paper, we propose TCP/HACK (Hierarchical ACKnowledgment), a system that applies cross-layer optimization to TCP traffic on WiFi networks by carrying TCP ACKs within WiFi's link-layer acknowledgments. By eliminating all medium acquisitions for TCP ACKs in unidirectional TCP flows, TCP/HACK significantly improves medium utilization, and thus significantly increases achievable capacity for TCP workloads. Our measurements of a real-time, line-speed implementation for 802.11a on the SoRa software-defined radio platform and simulations of 802.11n networks at scale demonstrate that TCP/HACK significantly improves TCP throughput on WiFi networks.

1 INTRODUCTION

In today's WiFi wireless networks, each time a sender wishes to transmit, it must first sense the medium to be idle for a randomly chosen interval. These random delays desynchronize would-be concurrent senders. To use a concrete example, Enhanced Distributed Channel Access (EDCA) in 802.11n [1] enforces an average idle period of 110.5 μs before a frame's transmission, whereas a 1500-byte payload itself lasts only 80 μs at 150 Mbps. Each frame's link-layer acknowledgment (LL ACK) consumes further channel capacity. As the physical-layer bit-rate increases but the pre-transmission idle period remains the same, this inefficiency only worsens. If a 600 Mbps 802.11n sender sent single frames in this fashion, it would only achieve 9% of the theoretical channel capacity. Moreover, WiFi senders back off exponentially after a failed transmission, and so incur even longer mean pre-transmission idle periods under contention, further reducing medium efficiency.

In an effort to amortize the significant overhead of medium acquisition over multiple data frames, 802.11n's MAC protocol batches multiple data frames into a single aggregate MAC protocol data unit (A-MPDU), and incurs only a single medium acquisition for each such batch. 802.11n further aggregates the LL ACKs for the data packets in a received A-MPDU into a single LL Block ACK. While batching helps one sender, TCP traffic is inherently *bidirectional:* a TCP receiver typically transmits a single TCP ACK packet for every pair of TCP data packets it receives. Not only do TCP ACKs incur further expensive medium acquisitions by the TCP receiver—they run the risk of colliding with the TCP data sender's transmissions as well.

WiFi's data frames elicit LL ACKs that the receiver sends without contending for the medium, as other would-be senders defer for an ACK frame's duration after hearing a data frame. We observe that this LL ACK is an ideal vessel for carrying TCP ACK information on the reverse path without incurring a costly medium acquisition. We name this overall cross-layer approach—in which a single transmission of feedback by a lower-layer protocol additionally carries feedback from a higher-layer protocol—Hierarchical ACKnowledgment (HACK). Though applying HACK to carry TCP ACKs in LL ACKs is conceptually quite simple, a robust design to do so must address several systems challenges. In this paper, we describe and evaluate such a design, TCP-over-HACK (TCP/HACK). Our contributions in this work include:

- We offer an analysis of the capacity of the 802.11n MAC protocol for TCP traffic as function of bit-rate, and the throughput gains theoretically achievable by avoiding medium acquisitions for TCP ACK packets.
- We describe TCP/HACK, a scheme that increases the WiFi MAC's efficiency by encapsulating TCP ACK information in WiFi LL ACKs. TCP/HACK fully supports 802.11n's batching of data packets and use of LL Block ACKs.
- We show how to efficiently encode the full range of TCP ACK information (*e.g.,* timestamp options, receiver window changes) within LL ACKs.
- We identify potential pathological interactions between TCP's reliability and congestion control mechanisms and WiFi's LL reliability protocol that would limit system throughput, and ensure that TCP/HACK avoids such interactions, without any changes to any node's

The research leading to these results has received funding under the EU's 7th Framework Programme, grant n° 317756, n° 287581, and European Research Council grant n° 279976. We gratefully acknowledge a hardware donation from the Microsoft Research Software Radio Academic Program.

TCP implementation.

- We offer an interface between the network device driver software and the network interface card (NIC) hardware that minimizes complexity in the NIC while allowing prompt sending of TCP ACK information in WiFi LL ACKs generated in response to WiFi data packets.
- Through an evaluation in simulation of up to 10 competing TCP flows on a 150 Mbps 802.11n network, we illustrate that TCP/HACK improves aggregate throughput up to 22% over TCP on "stock" 802.11n.
- Through an evaluation of a prototype online, wire-speed implementation of TCP/HACK for 802.11a on the SoRa software-defined radio platform, we illustrate that TCP/HACK improves aggregate throughput up to 32% over TCP on "stock" 802.11a.

2 Problem and Design Goals

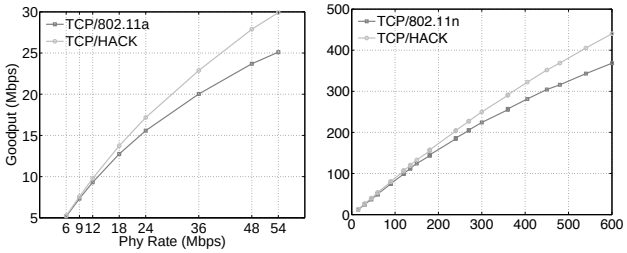
There are two distinct facets to improving the efficiency of the WiFi MAC layer for TCP transfers at fast bit-rates. First, we must understand the overhead of medium acquisition in WiFi 802.11a and 802.11n networks. How inefficient is the status quo, and what potential performance gains can one achieve by reducing the number of medium acquisitions? Second, we must articulate goals for our design to ensure that it meets the practical challenges of carrying feedback from a higher-layer protocol in a lower-layer one, as we propose to do in HACK. Such challenges arise because of the vagaries of wireless links (*e.g.,* frequent packet losses on links with poor signal-to-noise ratios), the potential for pathological interactions between TCP and the WiFi MAC protocol when optimizing across layers, and the constraints of real-world protocol stacks, network device drivers, and NICs. We now consider these two facets—medium acquisition overhead and practical design goals—in turn.

2.1 WiFi MAC Overhead

Consider a typical WiFi use scenario, where a single 802.11a or 802.11n client downloads a large file from a remote TCP sender. We assume throughout that the TCP receiver uses delayed ACK, and thus generates one TCP ACK packet for every two TCP data packets it receives.[1]

In Figures 1(a) and 1(b), the curves labeled "TCP 802.11{a,n}" show analytical predictions of the throughput a single TCP downloader achieves as a function of physical-layer bit-rate on lossless 802.11a and 802.11n networks, respectively. These analytical predictions are based on the parameters of the 802.11a and -n MACs. A detailed derivation of the capacity of the 802.11n MAC

[1]Note that this assumption is the best case for the efficiency of the status quo WiFi MAC—were delayed ACK not used, a TCP receiver would generate twice as many ACK packets, and the WiFi MAC would incur significantly more medium acquisitions.



(a) (b)

Figure 1: Theoretical goodput for 802.11a (a) and 802.11n (b) rates. In (b), theoretical TCP/HACK achieves an 8% improvement on average over TCP/802.11n for physical rates lower than 100 Mbps.

layer may be found in [6]; we do not repeat it here. The calculation for 802.11a is similar. (Figures 1(a) and 1(b) also show the improved throughput achieved by HACK, our modified 802.11 MAC protocol that carries TCP ACKs in link-layer ACKs, which we describe in Section 3.)

Note that for "stock" 802.11a and -n, the achievable TCP throughput is a progressively smaller fraction of the physical layer bit-rate as the latter increases. Time spent on non-payload overhead for each medium acquisition is to blame. In 802.11a, these overheads include the durations of DIFS and the contention window (both before a data frame's transmission), the data frame's preamble, the SIFS interval between data frame and LL ACK, and the LL ACK itself.[2]

As noted earlier, 802.11n aggregates data frames into A-MPDUs so as to amortize medium acquisition overhead over many frames, and combines multiple LL ACKs into Block ACKs in response. The results in Figure 1(b) include the application of these techniques, and show that while they reduce 802.11a's overhead, TCP still suffers progressively greater throughput limitations *vs.* the physical-layer rate because of the overhead of medium acquisitions for TCP ACKs.

2.2 Design Goals

To work robustly in practice, TCP/HACK must meet several demands that arise from the constraints of a modern wireless host's networking software and hardware, some of which are particularly unforgiving.

Hard real-time deadlines A WiFi receiver must reply to a data packet with an LL ACK within SIFS, an interval defined in the 802.11a specification (for example) as 16 μs. That deadline is of course far too short to meet in host software, so WiFi NICs validate received frames and generate LL ACKs in hardware. TCP/HACK must comply with these same LL ACK deadlines imposed by today's

[2]802.11n's parameter names and values differ slightly (*e.g.,* AIFS instead of DIFS); the overall scheme of per-medium acquisition overhead does not.

WiFi MAC. But if TCP/HACK is to enclose TCP ACK information in LL ACKs, the host TCP implementation cannot possibly generate a TCP ACK for a newly received TCP data packet within SIFS. To accommodate typical host protocol stack processing delays, TCP/HACK must allow the TCP ACK for a newly received TCP data packet to be enclosed within the LL ACK for a *different* TCP packet received later. Yet it mustn't unduly delay the return of an ACK to the TCP sender (see "cross-layer nuances" below).

Efficient encoding of general TCP ACK information The WiFi MAC reserves time on the wireless medium for a LL ACK to return after a data packet, so that other senders' transmissions do not collide with the LL ACK. It is important that TCP/HACK encode TCP ACKs in LL ACKs efficiently, to minimize the period of medium occupancy for these lengthened LL ACKs. The encoding for TCP ACKs must be compact yet allow the full generality of information that may potentially be found in a TCP ACK, (*e.g.,* TCP timestamp options, changes in receiver's advertised window, *&c.*) all of which is important to the correct and efficient operation of TCP.

Simplicity of NIC modifications TCP/HACK should not require any in-NIC intelligence about TCP packet headers or other TCP protocol details. Both at clients and APs, all TCP-aware processing must occur in the host software. We set this goal to minimize the complexity and thus the cost of the NIC, but also because we would like HACK to generalize to other higher layers than TCP such as SCTP [10] or DCCP [5]: if the NIC treats the feedback to be appended to an LL ACK as opaque bits that it needn't understand, then HACK should generalize in this way.

No changes to TCP TCP changes are difficult to standardize and difficult to deploy, as many widely used OSes ship with a single closed-source TCP implementation. Both at clients and APs, HACK-related functionality should be confined to the WiFi NIC's device driver (which is bound to the NIC's hardware design—*i.e.,* NIC hardware that supported HACK would routinely ship with a driver supporting HACK).

Avoid pathological cross-layer interactions Finally, it is important to note that TCP relies on a stream of TCP ACKs reaching the sender to maintain steady packet transmissions by the sender (and thus high throughput). TCP/HACK must not disrupt the timely return of correct TCP ACKs to the sender.

3 HACK Design

We first offer an overview of TCP/HACK's design. We then explore nuances of the cross-layer interactions between TCP and 802.11n, which motivate refinements to TCP/HACK that improve robustness and performance. Finally, we consider the constraints of real-world systems software and NIC hardware, as well as of lossy wireless links, and flesh out the design of TCP/HACK into a fully practical system.

In the interest of brevity, we describe the design of TCP/HACK in the context of an 802.11 client acting as a TCP receiver while downloading via an 802.11 AP. Throughout, we refer to this downloader as the "client." Note, however, that TCP/HACK is a fully symmetric design—both the design and our implementation of it also work on TCP uploads by an 802.11 client.

3.1 HACK in Overview

Let us first consider how TCP/HACK works in the simpler case of 802.11a, without batching of packets into A-MPDUs. When a regular TCP client receives a TCP data packet, its network stack generates a TCP ACK and enqueues it for transmission by the WiFi NIC.

Under TCP/HACK, a client does not immediately enqueue a TCP ACK for transmission. Instead, the client compresses each TCP ACK and appends them to a compressed frame that it builds. When the next data packet from the AP arrives, the client encapsulates the compressed TCP ACK frame within the returning LL ACK, *effectively avoiding all medium acquisitions for the corresponding TCP ACKs.* The AP recognizes an "augmented" LL ACK, which it decompresses, reconstitutes the encoded TCP ACKs, and forwards them upstream.

Now let us consider 802.11n, where data packets can be aggregated into a single batched A-MPDU, and link-layer ACKs take the form of a Block ACK that includes a bitmap indicating which packets from the A-MPDU were received. On "stock" 802.11n during a TCP download the normal repeating pattern will then be:

1. one A-MPDU from AP to client containing TCP data
2. one Block ACK from client to AP
3. one A-MPDU from client to AP containing TCP ACKs
4. one Block ACK from AP to client

To eliminate medium acquisitions for TCP ACKs in 802.11n, we would like a TCP/HACK client to encapsulate all the TCP ACKs generated in step 3 in the Block ACK sent in step 2, and thus avoid step 4 entirely. In practice, the arrival of an A-MPDU containing a batch of TCP data packets will cause the client's OS to generate a burst of TCP ACK packets in step 3 *after* the Block ACK has departed for that A-MPDU. These TCP ACKs arrive at the client's transmit queue where they are compressed and concatenated, waiting for the arrival of the *next* A-MPDU from the AP. The client will then append this compressed frame to the Block ACK it sends the AP in step 2.

Although the description above is for downloads, the design is in fact symmetric; we envisage TCP/HACK as especially useful for wireless backup to LAN-attached

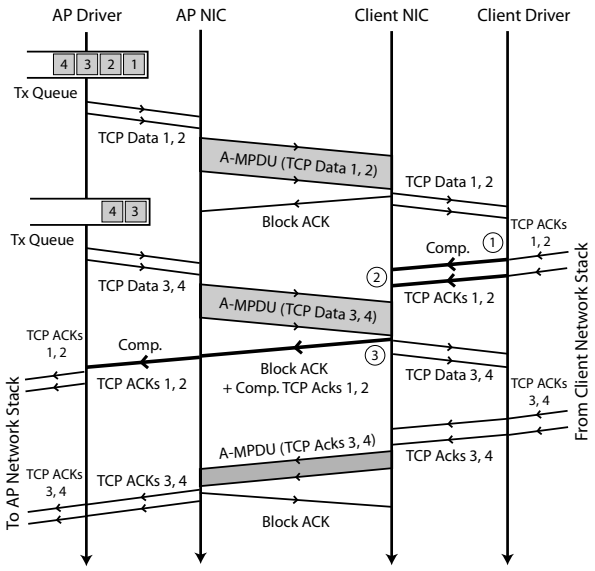


Figure 2: Interaction between A-MPDUs, Block ACKs and encapsulated HACK packets

storage, such as a Time Capsule.

3.2 Cross-Layer Nuances

We now refine our design to handle the subtle cross-layer interactions that arise between TCP and 802.11.

In principle, we would like to encapsulate TCP ACKs on the link-layer ACKs of the TCP packets they acknowledge. For example, if a batch containing TCP packets 1-64 arrives, the client would like to piggyback the TCP ACKs for packets 1-64 on the Block ACK for that batch. However, the 16μs SIFS interval between receiving data and sending the link-layer ACK or Block ACK is too short for the host's TCP stack to turn around the TCP ACKs, compress them, and DMA them to the NIC. For HACK to be practical, the compressed ACKs will have to wait until the next data arrives, and piggyback on its ACK or Block ACK. It turns out that this significantly complicates the dynamics of TCP/HACK and we will explore the consequences.

Figure 2 illustrates this process[3]. In response to a batch containing TCP packets 1 and 2, TCP ACKs 1 and 2 arrive at the client transmit queue too late to be carried on that batch's Block ACK. Instead, the TCP ACKs are compressed but not yet sent. When the next batch carrying TCP packets 3 and 4 arrives, its Block ACK can now carry the compressed frame with TCP ACKs 1 and 2. The AP then reconstitutes the full TCP ACKs and passes them up the network stack.

So long as TCP data packets continue to arrive, there is a steady stream of Block ACKs on which to piggyback compressed TCP ACK frames: all TCP ACKs are carried as HACK packets and no vanilla TCP ACK packets need to be sent. But what happens if no further data packets arrive? The client cannot retain the TCP ACKs for too long, or it will cause the TCP sender to time out and retransmit. Thus, after some time period, the client must send uncompressed vanilla TCP ACKs in the normal way. In Figure 2, TCP ACKs 3 and 4 meet this fate, and are in turn Block-ACKed.

Figure 1 summarizes the theoretical upper bound on TCP/HACK throughput on 802.11a (Figure 1(a)) and 802.11n (Figure 1(b)). The curves assume that the sender transmits the largest possible A-MPDUs,[4] that HACK manages to encapsulate all TCP ACKs in TCP Block ACKs, and that the compression is performed using the algorithm in Section 3.3. As the bit-rate increases, TCP/HACK significantly improves capacity, with a 20% improvement seen at 600 Mbps on 802.11n. In reality, the improvement can actually exceed that shown in the figure, as TCP/HACK can get closer to its bound than vanilla TCP can. This is due to collisions between TCP data packets and vanilla TCP ACK packets, a problem HACK sidesteps.

To HACK or not to HACK?

To maximize the benefits, TCP/HACK packets should be used whenever possible. But TCP ACKs must not be delayed when no more TCP data packets will arrive. How long should the client retain these TCP ACKs before giving up and sending them natively?

There are several reasons no more packets may arrive, including that the sender has stopped sending, but with 802.11n, the principal reason is the adverse effect of A-MPDUs on TCP's ACK clock. On a busy AP or during slow start, it is common for the entire TCP congestion window to be queued at the AP and then to be sent to the client in a single A-MPDU. An entire congestion window of TCP ACKs is generated and compressed, and these now sit at the client, waiting for the arrival of another incoming data packet so they can be send on its Block ACK. As the congestion window is full, this next packet never arrives and the connection stalls until TCP's retransmit timer fires. On 802.11a, which lacks aggregation, we don't often see this problem, but it is normal during slow start when 802.11n batching is used. We consider the following three different designs to address these concerns:

Explicit Timer A naive approach would be to have TCP/HACK time out and fall back to sending regular ACKs after a delay. In practice there is no good delay value that can be chosen, since the client cannot know the RTT and congestion window at the TCP sender, how the sender's packets will be spaced throughout the RTT, nor

[3]For simplicity it assumes that delayed TCP ACKs are disabled

[4]A-MPDU length is limited either by the 64 KByte A-MPDU bound or at lower bitrates by 802.11n's 4 ms transmit opportunity limit.

if the AP will suddenly start sending to another client.

Opportunistic HACK A more adaptive approach is not to explicitly delay TCP ACKs at all, but rather be opportunistic. When the wireless link is the bottleneck, the next downstream data batch will contend with the upstream TCP ACK batch. If the downstream batch wins, HACK can be used, but otherwise vanilla TCP ACKs will be sent. Such a design may often squander the opportunity to use HACK, but it has the virtue of seeming simple—until one considers the complexity of the NIC-network driver interface needed to implement it.

The MORE DATA Bit In Figure 2, initially there are four data packets queued at the AP. When the AP forms the first batch containing TCP data packets 1 and 2, it already knows more data will be sent to that client, as it already has packets 3 and 4 in its queue. So long as the AP has more packets queued than will fit in a batch, it knows that it is safe for the client to save up compressed ACKs waiting for the next batch. The AP simply tells the client that there is *more data* coming by setting the MORE DATA bit in the 802.11 header of the A-MPDU.[5] When the client sees this flag, it latches this state and will not transmit any more non-encapsulated TCP ACKs until the next data packet arrives, when it can use HACK to send them.

3.3 HACK in Practice

In the preceding section, we have presented a conceptual description of TCP/HACK, but several questions concerning the practicality of this conceptual design remain unanswered. First, how realizable is TCP/HACK given current systems and hardware? In particular, how should TCP/HACK's functionality be divided between a station's network interface card (NIC) hardware and NIC device driver? Finally, what manner of compression should TCP/HACK employ to reliably encode the TCP ACKs?

3.3.1 Driver and NIC Functionality

We realize TCP/HACK (including the MORE DATA mechanism) with very few changes to a station's 802.11 NIC. The main strategy is to implement the bulk of TCP/HACK within the NIC's driver, as we demonstrate using the example shown in Figure 2. Our discussion is in the context of a modern Linux wireless driver, such as the Atheros ath9k driver.[6]

AP (data transmission) The only modification needed to the AP when transmitting data packets is to set the MORE DATA flag when there are more packets remaining in the transmit queue for the same client.

[5]This bit exists in stock 802.11 to assist with power saving. HACK uses this bit irrespective of whether power saving is enabled.

[6]http://wireless.kernel.org/en/users/Drivers/ath9k

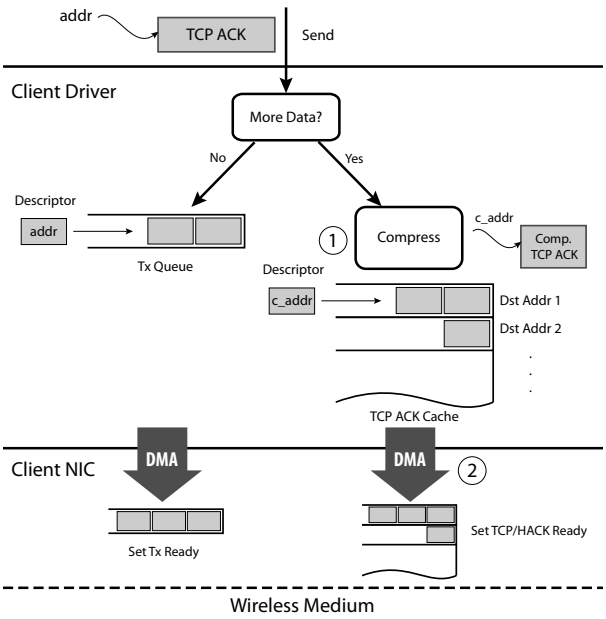


Figure 3: Client-side TCP/HACK compressing a TCP ACK, ready to be sent on the link-layer acknowledgment of the next frame.

Client The client's driver needs to determine when it can use TCP/HACK and when it must send TCP ACKs normally. In Figure 2, on receiving packets 1 and 2, the client's NIC also passes the MORE DATA state to the driver. The client TCP stack acknowledges the data, generating TCP ACKs 1 and 2, and puts them in the transmit queue at point (1).

Figure 3 shows what happens at points (1) and (2) from Figure 2 in more detail. If the driver is not in the MORE DATA state, it simply enqueues these ACKs normally. However, if MORE DATA is set, it compresses the arriving TCP ACKs and creates corresponding buffer descriptors. A separate buffer descriptor chain per destination address is needed to match compressed TCP ACKs with Block ACKs for that destination.

At point (2) the driver DMAs the buffer descriptor chain to the NIC. The NIC maintains this table of compressed TCP ACK descriptors separately from normal transmission descriptors. Finally, the driver sets a flag in the NIC to indicate that TCP/HACK is ready.

Figure 4 shows what happens when the next batch from the AP arrives at the client. If the TCP/HACK flag indicates "ready," the NIC uses the corresponding descriptors to DMA the compressed TCP ACK frames to the card. It concatenates these frames, and appends them to the returning Block ACK at point (3). Recall that the NIC normally fires an interrupt when it receives data packets. In this case, the interrupt must also indicate whether the NIC succeeded in sending the compressed ACKs.

This design also copes with the race condition where

the batch carrying packets 3 and 4 arrives with the MORE DATA flag not set before the driver has succeeded in conveying compressed TCP ACKs 1 and 2 to the NIC. In this case, the TCP/HACK "ready" check will fail. The NIC sends a normal Block ACK and signals to the driver a TCP/HACK failure in the receive interrupt. The driver now is free to re-enqueue the TCP ACKs on the transmit queue for normal transmission.

AP (ACK reception) Finally, the AP needs to recognize and decompress the "augmented" Block ACKs. The task of recognition falls to the AP's NIC, which extracts the compressed TCP ACK frame from the received Block ACK, adds it to the transmit complete report and interrupts to indicate transmit complete. The driver extracts the compressed TCP ACK frame, decompresses and reconstitutes the TCP ACKs, and forwards them upstream.

3.3.2 Compression

A critical component of the design is choosing a compression method for TCP ACKs. As 802.11a and -n transmit LL ACKs at one of the slower basic rates, *e.g.* 6 Mbps, it is desirable to minimize the size of the TCP ACK information appended to LL ACKs. Moreover, the 802.11a and -n MAC protocols' DIFS and AIFS intervals protect "stock" LL ACKs from collisions. Ideally, the compressed ACK information that HACK appends to LL ACKs should be short enough to fit within DIFS and AIFS, to avoid risking a collision.[7] We would like to leverage the redundancy within TCP and IP headers across consecutive TCP ACKs. Since most of the TCP/IP header fields remain static for a particular flow, they can be cached at the compression and decompression endpoints. To encode TCP and IP header fields reliably, TCP/HACK uses *Robust Header Compression (ROHC)* [8] to efficiently condense TCP/IP segments. ROHC supports the most popular TCP options like Timestamps and Selective Acknowledgments (SACK), and defines the notion of contexts, each with a particular identifier (CID). A context for TCP/HACK's purposes maps nicely to a particular TCP flow. In addition to caching static fields like the TCP/IP five-tuple at the endpoints, ROHC losslessly compresses the dynamic fields like the TCP Sequence and ACK numbers.

TCP/HACK-specific ROHC optimizations Since TCP/HACK applies ROHC in a specific context, we make the following simplifications:

1. We do not explicitly send Initialize-Refresh (IR) packets from the TCP client to the AP. To initialize a new context, the client can simply send uncompressed TCP ACKs outside of the TCP/HACK mechanism. The AP will consequently store the necessary state for the new context and assign it the correct CID.
2. The client and AP need not exchange any messages to agree upon a new CID for an emergent flow. Instead CIDs are computed independently at each endpoint. The client's driver on receiving a TCP ACK for a new flow computes the MD5 [9] hash over the ACK's 5-tuple and selects the lowest byte as the CID.
3. Compressed TCP ACK packets encapsulated within link-layer ACKs require a new mechanism to deal with losses outside of sending explicit ROHC feedback packets. We describe how TCP/HACK handles losses in Section 3.4.

With ROHC, a driver can shrink a TCP ACK to about 4 bytes, or even 3 bytes if the associated flow transmits a constant payload size (*e.g.* for large file downloads) [8].

[7] In our simulations in Section 4.3, we find that 98.5% of the LL ACKs carrying ROHC-compressed TCP ACKs fit within AIFS for best-effort traffic. For the few that don't fit, the sender may either split the compressed TCP ACKs across multiple LL ACKs (ensuring each LL ACK is fully protected by AIFS) or it may send them all on a single LL ACK (risking a collision with a hidden terminal). Our simulator does the latter; there are no hidden terminals in the scenarios we simulate.

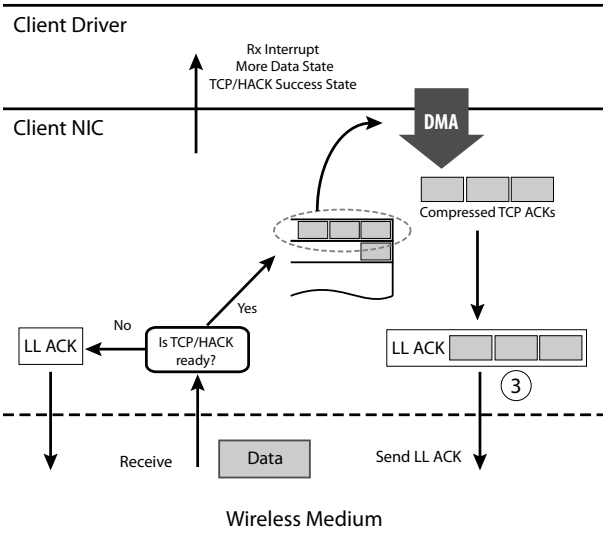


Figure 4: Client-side TCP/HACK receiving a batched frame from the air and including compressed TCP ACK frames in the corresponding link-layer acknowledgment.

3.4 Avoiding Cross-Layer Pathologies

The protocol we have described so far works well in a lossless environment. When applying HACK in low signal-to-noise ratio (SNR) regimes, decoding failures will cause packet drops. Any of the various packets sent by TCP/HACK may be dropped: TCP data packets, TCP ACKs, LL HACKs that contain LL Block ACKs and TCP ACK information, LL ACKs, *&c.* Under such losses, several concerns arise. To decompress headers correctly, ROHC requires that compression state at sender and receiver remain synchronized. Packet losses may cause loss of synchronization of this state, and in turn cause CRC failures on decompressed TCP ACK packets. Such loss of synchronization must not be persistent. We now describe

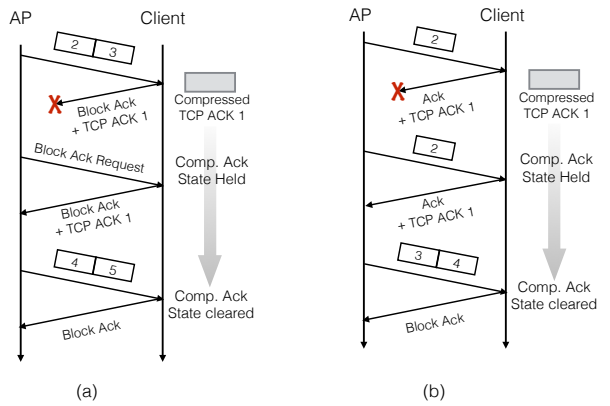


Figure 5: Coping with loss of (a) Block ACKs and (b) single LL ACKs by retaining TCP ACK state.

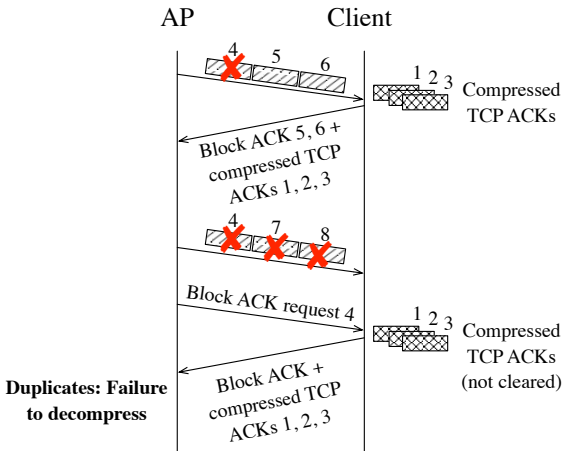


Figure 6: Retaining state: gap in sequence space.

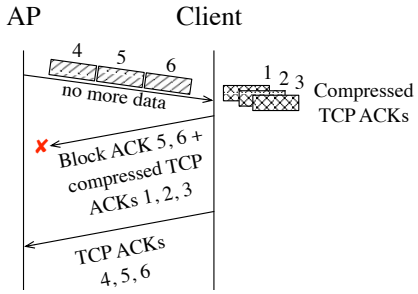


Figure 7: Flushing state: HACK-to-TCP ACK transition.

to restore lost synchronization quickly, to preserve the flow of TCP ACKs to the TCP sender.

Loss of LL ACK. First, consider the scenarios in Figures 5(a) and 5(b), where a Block ACK and single LL ACK carrying compressed TCP ACK information cannot be decoded, respectively. In both these scenarios, to deliver compressed TCP ACK(s) reliably, the client must retain them until it determines that its LL ACK (whether a Block ACK or a single LL ACK) has reached the AP. There is no such explicit indication from the AP, however. The client must enclose the same compressed TCP ACKs in all LL ACKs it sends to the AP until an *implicit* indication from the AP that the AP received the client's LL ACK. When the client has sent a Block ACK in response to an A-MPDU, as in Figure 5(a), receipt by the client of any subsequent A-MPDU (whether containing retransmitted MPDUs or not) indicates that the AP has received the client's Block ACK—if the AP has not done so, it must instead send a Block ACK Request. Alternatively, when the client has sent a single LL ACK in response to a single MPDU, as in Figure 5(b), the client can be certain that the AP has received its LL ACK upon receiving an MPDU with a greater MAC-layer sequence number—if the AP has not done so, it must instead retransmit the MPDU with the same MAC-layer sequence number. In both these cases, once the client has implicitly determined that its LL ACK has been received by the AP, it can safely discard any compressed TCP ACK information it has previously sent to the AP within that LL ACK.

Loss of retransmission. Since the ACKs themselves are not acknowledged, the ambiguity shown in Figure 6 can arise. The client cannot tell from the Block ACK request for 4 that the Block ACK for 5 and 6 was actually received. Thus it appends the compressed ACKs for 1,2 and 3 to the Block ACK response. Note that we cannot use the starting sequence number in the Block ACK Request as a signal that we have moved on to new data here because in this case it points to a gap in the sequence space, even though the rest of the aggregate is new.

The AP has already received and decompressed these ACKs, so its state is incorrect for decoding their retransmission. ROHC already has a mechanism to cope with duplicates—it has a master sequence number that increases monotonically. The lower 4 bits are normally included in each compressed packet. This is not sufficient for the first compressed TCP ACK packet carried in a Block ACK as an A-MPDU can carry 64 packets. We extend this first master sequence number to 8 bits, allowing the AP to discard duplicate compressed TCP ACKs and get back in sync.

Lost Block ACK, No More Data. Another corner case arises in Figure 7 when a Block ACK with compressed ACKs is lost, and the client needs to send vanilla TCP ACK packets because the last batch was not marked MORE DATA. Here, the client clears any compressed TCP ACKs it has retained, and sends the next TCP ACK packet with a higher sequence number. TCP ACKs are cumulative and the upstream server will deal with the newer TCP ACK correctly even though there is a gap in received TCP ACK numbers.

Repeated loss of Block ACK. Finally, what happens when a Block ACK with compressed TCP ACKs is lost

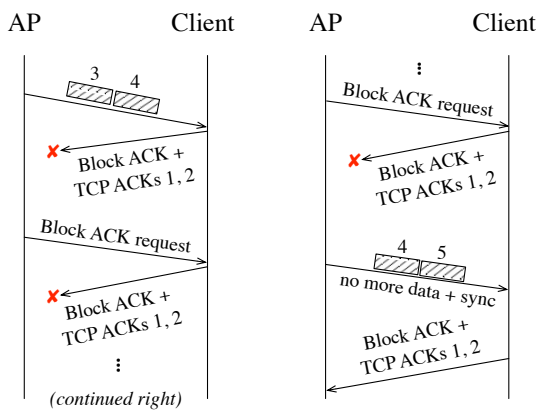


Figure 8: SYNC bit for retaining state.

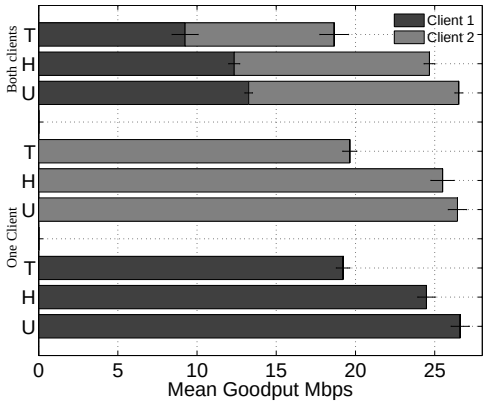


Figure 9: TCP throughput with stock 802.11a (T), TCP with HACK (H), and UDP (U) with stock 802.11a, with 1 and 2 clients.

repeatedly? Under normal 802.11n operation, the AP will continue to send Block ACK requests until it hits the retry limit, when it will give up and send the next batch of data. The client does not know that the AP has failed to receive the compressed TCP ACKs and, when it sees new data, it would normally discard the previously retained TCP ACK state. In this case, the AP explicitly notifies the client that it has moved on by setting a SYNC bit in the next batch's header. Upon seeing this bit set, the client doesn't discard the compressed TCP ACKs but rather appends them to the next Block ACK, as shown in Figure 8.

4 Evaluation

We evaluate TCP/HACK through a combination of simulation in ns-3 and experiments with a real-world implementation for the SoRa software-defined radio platform. We simulate TCP/HACK for 802.11n in ns-3, while our SoRa implementation is for 802.11a, as the public SoRa release does not support 802.11n.

4.1 SoRa Implementation

We implemented TCP/HACK including the More Data bit and ROHC compression for the SoRa user-level physical layer on Windows 7. Hardware limitations of our SoRa radio boards require us to run 802.11a in the 2.4 GHz band, but this does not affect protocol behavior.

One quirk of the SoRa platform bears mention. We have found that SoRa receivers sometimes return 802.11 link-layer ACKs later than the 802.11 specification's ACK timeout interval, causing spurious link-layer retransmits and backoffs. To avoid this performance hit, we increased the 802.11 ACK timeout to accommodate SoRa's late LL ACKs. The net effect of these delayed LL ACKs is that at 54 Mbps, our SoRa implementation only achieves 87% of the theoretical throughput across all protocols. We confirmed though simulation that this change does not significantly affect the relative benefit of TCP/HACK over regular 802.11a, but the absolute performance numbers are slightly lower.

Testbed Our three wireless nodes each have four-core Intel Core i7 CPUs, between 8–24 GB of RAM, and a PCI Express SoRa radio control board. One acts as the AP and the other two act as clients. We operate the SoRa interfaces in ad hoc mode to eliminate periodic beacon transmission. We run experiments on 802.11g channel 14 (2.484 GHz) in an open-plan office environment. We use *iperf* to generate TCP data streams with a 1500 byte MTU and send at 54 Mbps, the highest 802.11a rate.

4.2 SoRa Results

Besides demonstrating a successful implementation as evidence of TCP/HACK's practicality, we wish to answer several questions experimentally:

- Are TCP/HACK's capacity benefits in line with theoretical predictions?
- When an AP sends TCP flows to two clients, does TCP over 802.11a suffer collisions between clients' TCP ACKs, and if so, does TCP/HACK offer a performance benefit partly by eliminating such collisions?
- Do TCP/HACK's benefits come only from eliminating channel acquisitions and collisions, or are there other overheads that TCP/HACK eliminates?

Baseline Comparison Figure 9 compares the application-level throughput achieved by TCP/802.11a and TCP/HACK for bulk downloads, with UDP/802.11a for comparison. Each bar shows a different experiment: sending to one or both clients, using TCP over HACK, TCP over stock 802.11a or, as a control experiment, unidirectional UDP, which gives an upper bound on usable capacity. The data is the mean over five different 120-second runs; error bars show standard deviation.

Client 1's throughput is slightly less than Client 2's because it suffers a greater packet loss rate, even when only one flow is active. UDP's unidirectional data minimizes

medium acquisitions, and achieves the greatest throughput possible on SoRa with link-layer ACKs enabled. In an ideal 802.11 MAC, UDP would achieve 30.2 Mbps; on SoRa, UDP averages 26.5 Mbps across the three experiments. SoRa's link-layer ACK delays alone reduce the attainable throughput to 28.1 Mbps, and our UDP measurements approach that figure.

If TCP/HACK encapsulated all TCP ACKs in LL ACKs, it would achieve almost the same throughput as UDP (though UDP's packet headers are smaller). In practice, TCP/HACK's single-client throughput of 25.0 Mbps (mean of C1 and C2) is very close to the UDP benchmark. TCP/802.11a only achieves 19.4 Mbps in this scenario. TCP/HACK improves performance by 29% and 32.2% in the one- and two-client cases respectively. Both TCP/HACK and TCP/802.11a are fair.

		UDP/ 802.11a	TCP/ HACK	TCP/ 802.11a
Client 1	no retries	99%	97%	87%
	1 or more	1%	3%	13%
Client 2	no retries	99%	98%	88%
	1 or more	1%	2%	12%
Both	no retries	99%	98%	86%
	1 or more	1%	2%	14%

Table 1: Percentage of frames successfully sent on the first attempt (no retries) and after one or more retries, when the AP is sending to Client 1 and Client 2 alone, and both clients at the same time, using UDP/802.11a, TCP/HACK, and TCP/802.11a.

Where do TCP/HACK's savings come from?

We note with interest that TCP/HACK improves throughput more than predicted analytically in Section 2.1. That prediction focused solely on saving medium acquisitions for TCP ACKs. In Table 1 we show the percentage of frames received after the first transmission, and the percentage that required one or more retransmissions. We see that TCP/802.11a experiences far more link-layer retransmissions than TCP/HACK or UDP/802.11a. These retransmissions occur because of collisions between TCP ACKs sent by clients and TCP data packets sent by the AP. TCP/HACK obviates most (but not all) of these TCP ACKs, and so significantly reduces the number of retransmissions needed. TCP/HACK not only eliminates costly channel acquisition overheads, but by encapsulating TCP ACKs in LL ACKs, also incurs fewer collisions.

	ACK count	ACK bytes	ACK_C count	ACK_C bytes	Comp. ratio
TCP/802.11a	9060	471120	0	0	(1)
TCP/HACK	10	520	9050	39478	12

Table 2: Conventional and compressed ACK counts, and compression rates of ROHC-compressed ACKs.

To understand other contributing factors in more detail, we ran an experiment where the AP transmits 25 Mbytes of data to a client using TCP/802.11 and TCP/HACK. By fixing the amount of work we can compare both protocols in time. The first two columns of Table 2 show the number of TCP ACKs sent as well as how many bytes were in those ACKs. The next two columns show the same figures for compressed ACKs, and the last column shows the compression rates ROHC acheives.

Reducing the number of transferred bytes is beneficial, but TCP ACKs are treated as regular data when sending over 802.11 wireless links and are sent at 54 Mbit/s in our experiments. LL ACKs, however, use the more robust 24 Mbit/s rate. To factor this in, we investigate how saved bytes translate into saved transmission time, together with TCP/HACK's impact on channel acquisition time and retransmission time.

	TCP ACK	ROHC	Acquire Channel	LL ACK overhead
TCP/802.11a	70 ms	0	1093 ms	456 ms
TCP/HACK	0.08 ms	13.1 ms	1.17 ms	0.46 ms

Table 3: TCP ACK time overhead breakdown for TCP/802.11 and TCP/HACK.

Table 3 shows time taken to send TCP ACKs (TCP ACK), time to send compressed TCP ACKs (ROHC), time spent waiting for channel before transmitting TCP ACKs (Channel) and extra time waiting for LL ACKs (LL ACK overhead). From the table, we see that most savings come from channel acquisition and LL ACK overhead.

Ideally LL ACKs are returned immediately after a SIFS time, but this is not always the case in the real 802.11 implementations. On SoRa we observe 37 μs on average of additional LL ACK overhead, while on two different commercially-available wireless NICs (the Atheros AR9300 and the Intel 5300) we measure 10.4-13.4 μs of LL ACK overhead, on average. While TCP/HACK benefits more from saving ACK overhead on SoRa than on the commercial cards, the benefit on commercial wireless hardware is still large. TCP/HACK not only eliminates channel overheads, it also reduces collisions and any additional LL ACK overheads incurred by the device.

SoRa and ns-3 Cross-Validation

To cross-validate our SoRa implementation against the ns-3 simulator, we simulated 802.11a in ns-3 with the same packet loss rate as that observed on SoRa (12% and 2% for TCP/802.11a and TCP/HACK, respectively). Since ns-3 returns LL ACKs immediately after SIFS, whereas SoRa incurs additional delay, ns-3 running TCP/802.11a achieves 22.4 Mbit/s *vs.* SoRa's 19.6 Mbit/s. After post-processing to eliminate SoRa's added LL ACK delay, we observe SoRa throughput of 22 Mbit/s, which matches simulation. Similarly, when simulating TCP/HACK in

ns-3, we get 28 Mbit/s *vs.* SoRa's 25.5 Mbit/s. After accounting for SoRa's extra LL ACK delay, SoRa achieves 27.7 Mbit/s, which matches simulation.

4.3 Simulation Results

We now examine how TCP/HACK interacts with frame aggregation, with a larger number of clients than possible in our testbed. To this end, we implement A-MPDU support and TCP/HACK in ns-3. We evaluate both the opportunistic and MORE DATA variants of HACK described in Section 3.2 to verify that the the latter outperforms the former as hypothesized.

We simulate multiple WiFi clients scattered randomly within a circle of 10-meter radius centered on the AP. Our aim is to model the scenario where several clients connect via 802.11n WiFi to a server located nearby on a high-speed LAN. We present results modeling an 802.11n single-antenna setup using data packet and link-layer ACK bit-rates of 150 Mbps and 24 Mbps, respectively. The wired link between the server and the AP has a latency of one millisecond and a bit-rate of 500 Mbps.

To glean the benefits of the MORE DATA scheme, we would like AP's transmit queue to contain at least 126 packets per flow. We choose this number so that the AP may buffer of up to three batches of 42 packets per client, accounting for some variability in the A-MPDU size in the presence of TCP retransmissions. To avoid adverse "buffer bloat" effects [3], the transmit queue should not be too large in the case of one flow, but rather grow as the number of flows increases. A large buffer in our system would cause an excessive loss of packets when slow start overflows the buffer, with or without TCP/HACK. With ten clients, the AP's transmit queue would be 1260, which is reasonable since Linux drivers usually use buffer sizes of 1000 packets.

TCP/HACK *vs.* TCP/802.11n To determine the benefit of TCP/HACK and its constituent parts, we compute the aggregate goodput for TCP flows sending 1460 byte packets, averaged across five simulated runs per experiment. To mitigate phase effects with multiple clients, we stagger the starts of clients' downloads. As such, we compute the aggregate goodput over the steady-state portion of the runs, once all the clients have more or less exited slow start.

Figure 10 shows that UDP maintains a roughly constant goodput as the number of downloading clients varies, as expected. As a unidirectional protocol, UDP's performance is minimally affected by the number of clients competing for the link. In contrast, the goodput of TCP/802.11n decreases slightly as the number of downloading clients increases. Although the AP elicits TCP ACK packets from clients in turn, there is still a chance that two or more clients' TCP ACKs can collide, or that a TCP ACK can collide with a data packet from the AP.

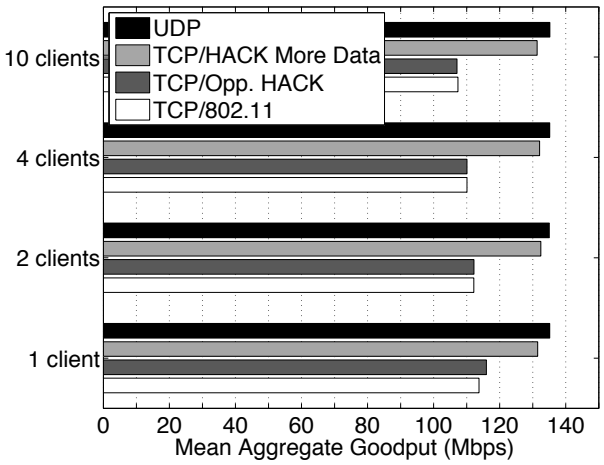


Figure 10: TCP goodput for different transmission schemes with 1–10 clients, and UDP for comparison.

These collisions account for the lower measured goodput than that predicted in Section 2.1.

We note with surprise that Opportunistic TCP/HACK does not significantly outperform TCP/802.11n: this most naïve implementation of HACK sends few compressed TCP ACKs in LL ACKs, and mostly regular TCP ACKs. It therefore does not achieve a TCP goodput closer to the physical rate.

Role of MORE DATA Bits We now turn our attention to the bars labeled "TCP/HACK More Data" in Figure 10. We observe that the MORE DATA variant of TCP/HACK achieves the most pronounced throughput gain over unmodified 802.11n. While simple, the MORE DATA mechanism is crucial to TCP/HACK's success in reducing medium acquisitions, and gives rise to goodput improvements between 15% for one client and 22% for ten clients at the physical rate of 150 Mbps.

Lossy Environment We next evaluate TCP/HACK under different SNR regimes. In addition to providing a wider spectrum of comparison between TCP/HACK and TCP/802.11n, these experiments will verify whether the HACK protocol with the properties described in Section 3.4 can indeed avoid any decompression CRC failures, or stalls due to recurring TCP timeouts.

We begin with a setup similar to that described above, and then place a single client at varying distances from the AP in order to simulate a decreasing set of SNRs. In lieu of simulating bit rate adaptation explicitly, at each particular distance we simulate a download of a 100 MB file at a rate selected from a range of 802.11n high throughput rates. This range corresponds to rates which are achievable using a 40 MHz channel, 400 ns guard interval and one antenna. The corresponding LL ACK rates are chosen from the set of basic rates (6, 12 and 24 Mbps) according to the rules outlined in the 802.11n specification. To

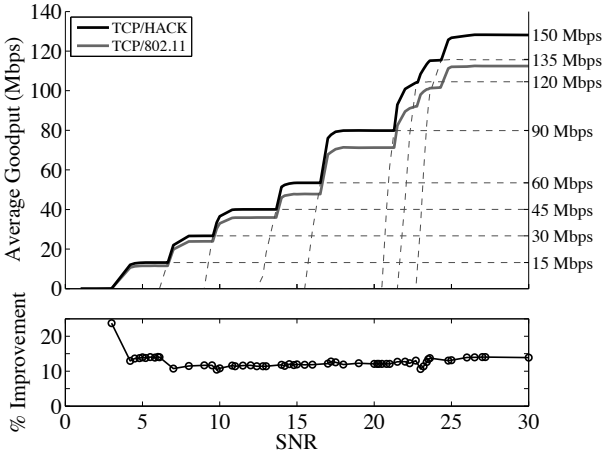


Figure 11: Envelope of average TCP goodput for TCP/HACK and TCP/802.11n under different SNR regimes and physical rates. The lower graph shows TCP/HACK's percent improvements over TCP/802.11n.

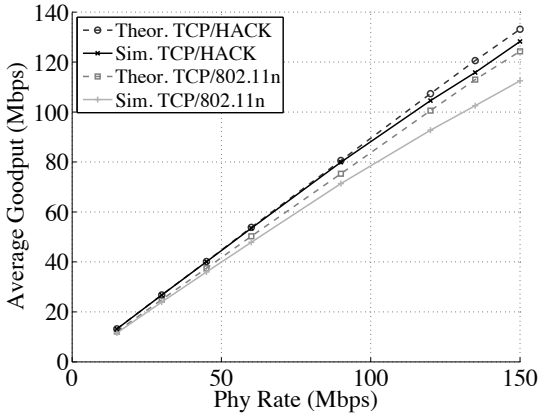


Figure 12: Theoretical and simulated TCP goodputs vs 802.11n physical rates.

emulate a real system, we applied the 4 ms transmit opportunity limit to all transmissions, therefore limiting the size of A-MPDU packets for experiments using lower physical rates. At each distance/physical rate combination, we computed the average TCP goodput (including slow start) over five runs.

Figure 11 shows the average TCP goodput for TCP/HACK and TCP/802.11n. It plots a separate dashed curve per 802.11n physical rate for TCP/HACK. We use these curves to compute the envelope (in black), which indicates the best goodput achievable by an ideal bit rate adaptation algorithm. Similarly we plot the corresponding envelope for regular TCP/802.11n (the separate rate curves for TCP/802.11n are not shown).

Our simulations indicate that TCP/HACK functions correctly in a lossy environment and does not elicit any decompression CRC failures. Moreover, TCP/HACK improves TCP goodput by an average of 12.6% across the range of SNR values. Figure 11 shows that as the physical rate drops, the relative improvement increases slightly for the cases where the transmit opportunity limit reduces the number of packets a station can possibly transmit in an aggregate. Recall that 802.11n uses aggregation to amortize medium access costs, therefore we expect a better goodput gain for TCP/HACK over regular TCP at these rates. Similarly, as the physical rate increases past 90 Mbps, the overall improvement increases slightly to about 14%, because the 802.11n medium access delays now consume a larger portion of the transmission time relative to data.

Analytical Predictions *vs.* Simulations How well does the average TCP goodput measured in simulation match that computed analytically in Section 2.1? We extract the highest achievable goodput at each physical rate for both TCP/802.11 and TCP/HACK from the prior experiment, and plot these and the analytical predictions in Figure 12. As we expect, simulated goodputs are lower than the corresponding analytical predictions—the predictions do not model 802.11n collisions or retries, nor do they take into account TCP's retransmissions and congestion control.

Note, however, that the goodput improvement TCP/HACK offers over TCP/802.11n exceeds that predicted analytically. Since TCP/802.11n suffers more from collisions than TCP/HACK, its throughput suffers correspondingly more. TCP/HACK greatly reduces the collision rate by eliminating medium acquisitions for TCP ACK packets. At 150 Mbps, TCP/HACK offers a simulated goodput improvement of 14%, *vs.* the 7% improvement predicted analytically.

5 DISCUSSION

Both batching using A-MPDUs and TCP/HACK help to reduce the time wasted on unnecessary WiFi medium acquistions. TCP/HACK relies on the MORE DATA bit to know when it is safe to compress ACKs and wait for another packet on whose LL ACK to piggyback. A-MPDUs require sufficient packets in the AP's queue to gain efficiencies. With sufficient buffering at the AP and a large window, both work well. In such cases the wireless medium is busy, and efficiency is important. TCP/HACK can significantly reduce collisions when there are multiple senders by turning bidirectional TCP flows into unidirectional ones, reducing the number of contending hosts. However, if the traffic patterns are such that queues do not build in the AP or clients, there won't be enough packets to fill A-MPDUs or any remaining packets in the queue to allow the MORE DATA bit to be set. Neither mechanism will work well in this case. Similarly, if an AP has very many clients, it may not buffer enough packets for each client for either mechanism to work well.

Longer batches improve utilization, but monopolize the medium for longer. 802.11e allows the AP to reduce

medium acquisition latency by specifying a shorter maximum batch duration through the transmit opportunity limit. In such cases, we would expect TCP/HACK to help claw back some of the efficiency loss caused by limiting the maximum batch duration.

Sending a TCP timestamp option in the last TCP ACK of a batch would generalize the MORE DATA mechanism. The TCP sender would echo it, and the client could use receipt of the echo as an implicit ACK-of-ACK. When the client hasn't yet received a timestamp echo, it can reasonably expect further data to arrive, and thus delay sending TCP ACKs. We leave this for future work.

6 RELATED WORK

One approach to amortizing medium acquisition overhead across more data is narrow-band channelization. Since the effective data rate on each subband is much lower than that of wide-band 802.11, the time required for MAC-layer contention becomes smaller relative to the packet transmission time on a single subband, thus more effectively amortizing medium acquisition overhead across multiple packet transmissions. FICA [11] and WiFi-NC [2] take this approach. Both require redesigns of the physical and MAC layers. TCP/HACK is complementary: combining the two systems should yield greater medium efficiency than either system achieves alone.

WiFi-Nano [6] shortens the 802.11 contention slot time to 800 ns. TCP/HACK is again complementary: while WiFi-Nano reduces medium acquisition overhead, our proposal eliminates many medium acquisitions entirely.

Maranello [4] is a link-layer design for 802.11 wireless networks that incorporates sub-frame granularity checksums into link-layer acknowledgments, allowing the communicating pair to undertake partial packet recovery on corrupted frames. Unlike TCP/HACK, Han *et al.* implement Maranello partially on the firmware processor of a commodity 802.11 NIC, thus requiring access to the assembly source code of the firmware processor. While Maranello does not share the same networking goals as TCP/HACK, it does share systems context in terms of the hardware and software available to both designs. Like Maranello, TCP/HACK is realizable with very few changes to the NIC itself.

Of prior work in reducing channel acquisition overhead, Pang *et al.* [7] most closely resembles TCP/HACK, proposing that a client use a MAC-layer ACK to signal successful reception of TCP data. However, the designs they propose are only capable of communicating to the AP when a client observes a TCP ACK for the *same data packet just received.* The authors do not mention the possibility of the generation of a TCP ACK with a lower ACK number after a loss, and the link-layer feedback mechanism they propose is incapable of communicating any information to the AP other than "cumulative ACK for the data packet just sent to the client" or "no ACK for the data packet just sent to the client." As a result, these designs prevent the delivery of duplicate ACKs to the TCP sender, and prevent the use of fast retransmit, leaving only inefficient TCP timeouts. Furthermore, this work took place before the introduction of 802.11n, and as a result, does not consider the interaction with frame aggregation or block ACKs.

7 CONCLUSION

In this paper, we have described the design and implementation of TCP/HACK, a cross-layer acknowledgment design for TCP and the 802.11 MAC that eliminates most of the expensive medium acquisitions that TCP ACK packets require, significantly increasing TCP flows' wireless throughput. TCP/HACK improves throughput further when used with frame aggregation, yet offers significant throughput improvements without it. While frame aggregation and other previous approaches reduce the cost of individual medium acquisitions [11, 2, 6], TCP/HACK eschews many medium acquisitions entirely. It is thus complementary to these prior approaches. Our evaluations in simulation and in a real-world implementation confirm TCP/HACK's throughput improvements.

REFERENCES

[1] IEEE Standard 802.11-2012. Mar. 2012.

[2] K. Chintalapudi, B. Radunovic, V. Balan, M. Buettener, S. Yerramalli, V. Navda, and R. Ramjee. WiFi-NC: WiFi over narrow channels. In *NSDI*, Apr 2012.

[3] J. Gettys and K. Nichols. Bufferbloat: Dark buffers in the Internet. *CACM*, 55(1), 2012.

[4] B. Han et al. Maranello: Practical Partial Packet Recovery for 802.11. In *NSDI*, 2010.

[5] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion control without reliability. In *SIGCOMM*, 2006.

[6] E. Magistretti, K. Chintalapudi, B. Radunovic, and R. Ramjee. WiFi-Nano: Reclaiming WiFi efficiency through 800 ns slots. In *MobiCom*, 2011.

[7] Q. Pang, S. Liew, and V. Leung. Performance improvement of 802.11 wireless network with TCP ACK agent and auto-zoom backoff algorithm. In *Proc. IEEE VTC*, June 2005.

[8] G. Pelletier, K. Sandlund, L.-E. Jonsson, and M. West. *RObust Header Compression (ROHC): A Profile for TCP/IP*. RFC 6846, Jan 2013.

[9] R. Rivest. *The MD5 Message-Digest Algorithm*. RFC 1321, April 1992.

[10] R. Stewart. *Stream control transmission protocol*. RFC 4960, Sept. 2007.

[11] K. Tan, J. Fang, Y. Zhang, S. Chen, L. Shi, J. Zhang, and Y. Zhang. Fine-grained channel access in wireless LAN. In *SIGCOMM*, 2010.