Cross-Layer Techniques for Efficient Medium Access in Wi-Fi Networks

Astrit Zhushi

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Computer Science

University College London

November 23, 2022

Abstract

IEEE 802.11 (Wi-Fi) wireless networks share the wireless medium using a Carrier Sense Multiple Access (CSMA) Medium Access Control (MAC) protocol. The MAC protocol is a central determiner of Wi-Fi networks' efficiency–the fraction of the capacity available in the physical layer that Wi-Fi-equipped hosts can use in practice. The MAC protocol's design is intended to allow senders to share the wireless medium fairly while still allowing high utilisation. This thesis develops techniques that allow Wi-Fi senders to send more data using fewer medium acquisitions, reducing the overhead of idle periods, and thus improving end-to-end goodput. Our techniques address the problems we identify with Wi-Fi's status quo. Today's commodity Linux Wi-Fi/IP software stack and Wi-Fi cards waste medium acquisitions as they fail to queue enough packets that would allow for effective sending of multiple frames per wireless medium acquisition. In addition, for bi-directional protocols such as TCP, TCP data and TCP ACKs contend for the wireless channel, wasting medium acquisitions (and thus capacity). Finally, the probing mechanism used for bit-rate adaptation in Wi-Fi networks increases channel acquisition overhead.

We describe the design and implementation of Aggregate Aware Queueing (AAQ), a fair queueing discipline, that coordinates scheduling of frame transmission with the aggregation layer in the Wi-Fi stack, allowing more frames per channel acquisition. Furthermore, we describe Hierarchical Acknowledgments (HACK) and Transmission Control Protocol Acknowledgment Optimisation (TAO), techniques that reduce channel acquisitions for TCP flows, further improving goodput. Finally, we design and implement Aggregate Aware Rate

Control (AARC), a bit-rate adaptation algorithm that reduces channel acquisition overheads incurred by the probing mechanism common in today's commodity Wi-Fi systems. We implement our techniques on real Wi-Fi hardware to demonstrate their practicality, and measure their performance on real testbeds, using off-the-shelf commodity Wi-Fi hardware where possible, and software-defined radio hardware for those techniques that require modification of the Wi-Fi implementation unachievable on commodity hardware. The techniques described in this thesis offer up to 2x aggregate goodput improvement compared to the stock Linux Wi-Fi stack.

Impact Statement

With the proliferation of mobile handheld devices, Wi-Fi has become the de facto technology for providing network connectivity in a large variety of environments ranging from small homes and offices to large offices, often densely placed in cities. In a small home/office environment, the algorithms described in this thesis make faster browsing and downloading of content via the Wi-Fi network possible, while in larger-scale environments they allow more users to share a Wi-Fi network more efficiently. We have implemented our algorithms using off-the-shelf, commercially-available Wi-Fi devices to demonstrate their practicality. Our evaluation results from an indoor testbed help to extend our understanding of Wi-Fi systems' performance.

Acknowledgements

I wish to express my deepest gratitude to my supervisors, Brad Karp, Mark Handley, and Kyle Jamieson for their guidance, support and encouragement. I am lucky to have had the opportunity to work with such brilliant minds and to be part of the wonderful Systems and Networks Research group at UCL. Also, my thanks extend to Stefano Vissicchio for his advice, support and always being available for a discussion.

I offer my sincere gratitude to my parents for giving me the opportunity to be what I wanted and always believing in me.

My wife, Besiana, was with me every step of the way during this endeavour. Thank you for always being there, through good times and bad times. Without your love, support and backing I couldn't be here today. Also, thank you for being a wonderful mother. To my wonderful children, Mat and Era, thank you for enriching my life every day.

My siblings, Artan and Blerta, thank you for your support and I could not have been more fortunate to have you.

I would also like to thank Lynne Salameh, Nikola Gvozdiev, Georgios Nikolaidis and Petr Marchenko, my fellow PhD students at UCL, whose friendship I highly value.

Finally, last but not least, to all my friends including Visar and Semra Broja, thank you for the continuous support and friendship and always being there for me. To Alban Bakija, Labinot Halilaj, Burim Meholli, and Visar Ramajli, thank you for being great friends and always being with me when I needed you.

This thesis is dedicated to my beloved parents

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Wireless communication technologies have become essential to delivering our daily communication needs. Given the surge in use of handheld devices such as mobile phones, wireless networking has become a prevalent networking technology. Today, the main wireless local area network technology is based on the IEEE 802.11 standard family, commonly known as *Wi-Fi*. Lack of wiring, ease of installation, and support for mobility are some of the reasons why Wi-Fi is ubiquitously deployed in diverse environments such as homes, offices, airports, shopping malls, and university campuses. To meet this ever-growing demand, IEEE 802.11 has evolved significantly in successive standards, offering improvements in performance and capabilities. For example, the early generation of Wi-Fi devices based on IEEE 802.11b [6] supported physical layer (PHY) rates up to 11 Mbps. The latest IEEE 802.11ax [23] promises PHY rates up to 10Gbps. These advancements have been achieved by enhancing the PHY layer and the medium access control (MAC) layer. In the PHY layer, better spectral efficiency [5], faster modulation and coding schemes [50], wider bandwidth [50], and the use of multiple antennas to transmit independent streams of data [50] have been introduced. The MAC layer, too, has seen several enhancements, such as quality of service [49], frame batching [49], and frame aggregation [50]. However, the Wi-Fi MAC layer's fixed idle periods when accessing the wireless medium have remained relatively unchanged. These fixed periods of idle time, while necessary, incur extra overhead. Reducing these overheads is important to achieve high wireless channel utilisation, and thus high goodput. For example, 802.11n [50] introduced frame aggregation, which allows senders to send multiple frames using a single channel acquisition, thus amortising the Wi-Fi MAC layer's idle time overheads. However, unless the sending host manages to queue packets efficiently to allow for a large number of frames to be aggregated, these frame aggregation benefits diminish. Another example concerns the Transmission Control Protocol (TCP), the protocol carrying the majority of today's Internet traffic. For every two TCP data packets received, the receiver transmits a TCP ACK back to the transmitter [4]. To transmit TCP ACKs, the receiver must acquire the wireless channel, directly contending for the wireless medium with the sender, thus incurring extra overhead.

In this thesis we take a cross-layer approach, where we look at all layers involved in transmitting or receiving data and analyse their interactions, with the aim of improving the MAC protocol's efficiency. We analyse the end-to-end goodput of commodity Wi-Fi systems and identify shortcomings in today's TCP/IP/Wi-Fi protocol stack that give rise to inefficient use of the wireless medium. Armed with these insights, we design algorithms and techniques that improve MAC protocol efficiency, resulting in significant end-to-end goodput improvements. We evaluate our systems using real indoor experimental testbeds, and show their practicality by implementing them on real hardware. As the hardware platforms available to us for open-source development dictate which IEEE 802.11 standards we can use, in this thesis we limit our scope to IEEE 802.11n, though the improvements we describe are applicable in later Wi-Fi standards that use similar MAC mechanisms, such as the recent IEEE 802.11ac [10]. More generally, we expect our techniques should offer efficiency benefits to other CSMA-based wireless systems.

1.1 Problem Statement

In a multiuser wireless network, sharing the radio spectrum resources fairly and efficiently is a perennially challenging problem. The main methods used to divide the spectrum amongst wireless users include frequency division [45], time division [45], code division [45] and a combination of these [45]. Some of the main factors that determine which of the methods are most appropriate include the number of wireless stations, traffic patterns, and power requirements. For example, in frequency division multiple access (FDMA) the base station assigns a frequency band used for transmission to each wireless station. While simple, FDMA suffers from several disadvantages, such as spectral inefficiency and frequency-selective fading [45]. To avoid interference from adjacent frequencies, FDMA utilises a guard band between adjacent frequencies, which incurs overhead, as such guard bands do not carry data. In addition, when wireless stations are not transmitting, their assigned frequency is unused, effectively wasting capacity that could be used by other stations.

Orthogonal frequency-division multiplexing [93] (OFDM) sees widespread use. In OFDM, spectrum is divided into orthogonal frequencies called *sub-carriers*. Unlike FDMA, OFDM does not require guard bands to guard against interference, making it more spectrally efficient and robust to frequency-selective fading. Prominent technologies that use OFDM include IEEE 802.11 a/g/n/ac [10], WiMAX [9], and DAB [69].

Also widely deployed is orthogonal frequency-division multiple access (OFDMA), an extension of OFDM. OFDMA may further divide spectrum into narrower sub-carriers and groups sub-carriers into resource units (RU). Unlike OFDM, where all of a channel's sub-carriers are used at once by a single wireless station, in OFDMA the AP can transmit to multiple users during a single channel acquisition by assigning different RUs to different wireless stations. This makes OFDMA an ideal fit for scenarios such as the Internet of things (IoT), where many low-bitrate wireless stations send traffic periodically. OFDMA is used in wireless technologies such as LTE-M [25] and NB-IoT [26].

More recently, IEEE 802.11ax [23] combines CSMA/CA with OFDMA for channel access to support a large number of wireless stations.

To support high-throughput, bursty traffic, in 802.11 a/g/n/ac wireless stations may transmit at different rates without centralised coordination. To manage the wireless medium, the 802.11 MAC protocol uses the Distributed Coordination Function (DCF) algorithm. In DCF, wireless stations coordinate their transmissions in a distributed fashion. At the heart of DCF is Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) [57, 32]. Like their Ethernet [66] cousins, wireless stations must listen on the channel before they can transmit. The wireless medium must be sensed free for some period of time before transmissions can commence. If the wireless medium is sensed busy, or it becomes busy while waiting, the transmission must be deferred. Wireless stations continuously listen on the channel while waiting for the full waiting period. Once the wireless medium has remained free, transmission begins. After each transmission, the transmitter waits for a response from the receiver to acknowledge the reception of the transmission. The receiver must respond within a fixed period of time; absence of the confirmation from the receiver is treated by the transmitter as transmission failure. Detecting collisions in wireless channels is challenging, as the transmitter cannot listen while it is transmitting. While some proposed systems can receive transmissions while transmitting on the same channel [53], these techniques have not yet found broad commodity use. As a result, failure to receive an acknowledgement from the receiver is treated as a signal of congestion, and the transmitter then increases the time during which it must sense the medium free before retransmitting. These waiting times, while necessary, incur extra overhead. And these idle times waste even more capacity as IEEE 802.11's PHY rates continue to increase. Figure 1.1 shows the overhead of the MAC protocol relative to the time it takes to transmit a fixed-size data packet, using the greatest supported PHY rate in each of the IEEE 802.11b,a/g,n standards.

802.11 supports multiple PHY rates and the greatest PHY rate supported

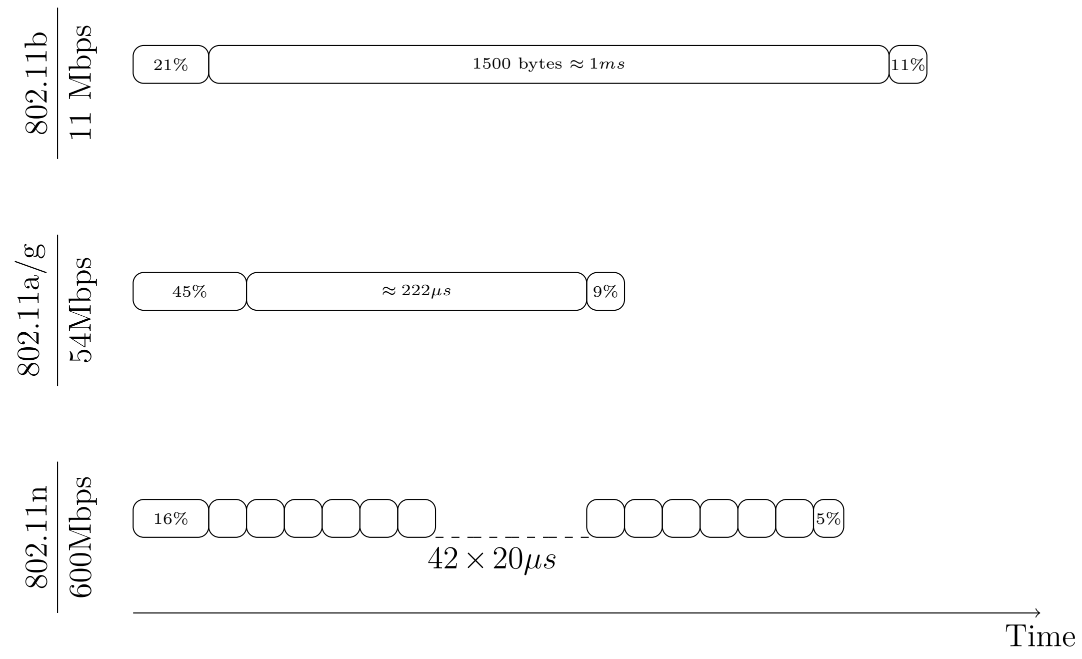


Figure 1.1: The relation between data serialisation time and the fixed overheads incurred by the wireless medium acquisition procedure for various IEEE 802.11 MAC protocols.

by IEEE 802.11b [6] is 11 Mbps. Transmitting a 1500-byte packet at this rate takes about 1 ms. The MAC protocol incurs 32% overhead, where the overhead includes the preamble prepended to each packet, the overhead to send back the ACK, a CRC, and fixed waiting times. In IEEE 802.11a/g [5][7] the maximum supported PHY rate increased to 54 Mbps, but the fixed waiting times remained the same. As a result, the overhead increased to about 54%. Finally, the greatest PHY rate supported by IEEE 802.11n [50] is 600 Mbps. At this high rate, it takes a mere 20 μs to transmit a 1500-byte packet. Sending single frames yields a prohibitively expensive 81% waiting time overhead. Frame aggregation was introduced in IEEE 802.11n in an attempt to amortise these overheads. Using frame aggregation, multiple packets are sent back-to-back once the wireless channel is acquired. If we could arrange for 42 packets to be sent in a single aggregated frame, frame aggregation would improve utilisation and reduce overhead to approximately 21%. Despite these enhancements, the IEEE 802.11 DCF's overheads penalise higher level protocols such as TCP [4]. Like the IEEE 802.11 DCF, TCP uses its own positive acknowledgments (TCP ACKs) to report successful receipt of TCP segments. TCP ACKs are small in size, typically 40 bytes. The IEEE 802.11 DCF treats TCP ACKs as normal data and so sends them using the channel acquisition procedures described above, incurring many medium acquisitions just for TCP's receiver feedback. Figure 1.2 shows the achievable goodput for various PHY transmit rates (Mbps) as a function of the number of frames aggregated after a single medium acquisition for a single TCP flow. Each curve stops at the maximum allowed aggregate size for a given PHY rate. The

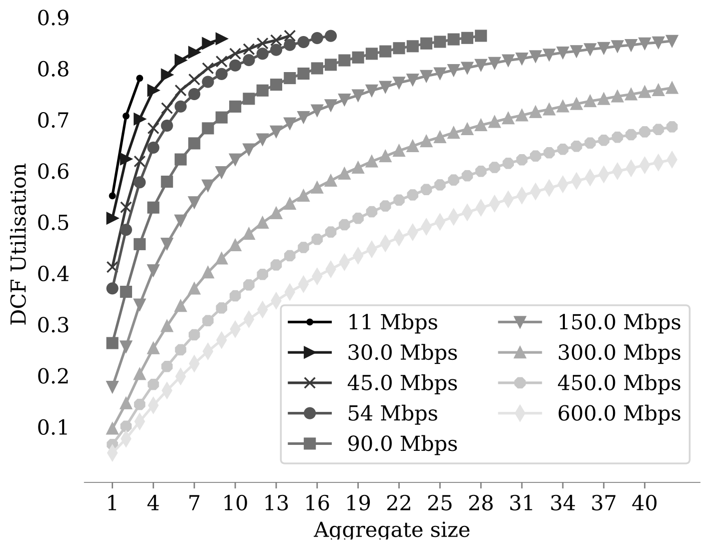


Figure 1.2: IEEE 802.11 DCF's channel utilisation for a single station sending a single TCP flow on an error free wireless channel. Default channel access parameters for IEEE 802.11b, curve labeled **11 Mbps**, IEEE 802.11a, curve labeled **54 Mbps**, and IEEE 802.11n rest of the curves.

two curves labeled 11 and 54 represent the highest rates supported by IEEE 802.11b and IEEE 802.11a/g, respectively. While IEEE 802.11b/a/g do not support aggregation, aggregation would in principle improve their goodput. The first observation from Figure 1.2 is that the waiting time overhead is even greater than that shown in Figure 1.1. This extra overhead comes as a result of medium acquisitions for TCP ACKs. For example, for IEEE 802.11b, overhead increases to 45%. The second observation is that sending larger aggregates increases channel utilisation. In this thesis we will investigate why IEEE 802.11 fails to achieve good channel utilisation in practice, despite the potential for frame aggregation to improve channel utilisation. We hypothesise that there are five main causes:

H1 Frame aggregation achieves the greatest channel utilisation when large aggregates are transmitted. To send large aggregates, enough packets destined for a single receiver have to be queued at the layer where aggregates are formed. Our hypothesis is that there are not enough packets at the aggregation layer to form large aggregates.

H2 Traditionally rate adaptation algorithms in Wi-Fi networks use probing to select among the available transmit rates. These probing frames interfere with frame aggregation.

H3 The Wi-Fi MAC treats TCP ACKs as regular data frames that are subject to the same channel contention procedures as other data frames. Contention of TCP ACKs with the AP's TCP data transmissions reduces channel utilisation.

H4 The receiving host's TCP stack generates TCP ACKs as it processes incoming TCP data. As a result, TCP ACKs arrive sparsely at the Wi-Fi stack and are immediately transmitted. This increases the number of channel acquisitions, reducing channel utilisation.

H5 When the AP is forwarding TCP data to multiple Wi-Fi clients, TCP ACKs from one TCP receiver contend with TCP ACKs from other TCP receivers, in part because of problem (**H4**).

1.2 Scope

The IEEE 802.11 standards define protocols that describe the operation of the physical (PHY) and link (MAC) layers. As such they incorporate a broad range of electrical engineering and computer science topics. Our scope in this thesis is limited to the MAC layer as defined in these standards. One of the aims in this thesis is to design systems that are practical and, where possible, run with minimal or no hardware changes. As such, we strive to design systems that can run out of the box using off-the-shelf, commercially available hardware. To achieve this aim we limit our scope to the CSMA/CA MAC protocol as defined in the IEEE 802.11 standards. During the work on this thesis, the IEEE 802.11 committee introduced a new IEEE 802.11ax standard that departs from CSMA/CA and introduces a more centralised MAC protocol to manage wireless transmissions. It aims to provide connectivity to large numbers of devices, such as in the Internet of Things scenario, sending at low bit-rates. By contrast, CSMA/CA, as used in IEEE 802.11b,a/g,n,ac networks, aims to provide high throughput to fewer devices. We will discuss the implications of the IEEE 802.11ax MAC design in Chapter 6. In this thesis the evaluation is limited to a typical scenario where users are downloading data from a remote server where the bottleneck is the AP. As such, this thesis focuses on measuring the throughput, which is the most relevant metric in such a scenario. Other metrics, such as jitter and latency, which are relevant for real-time voice and video workloads, are left as future work.

1.3 Outline and Contribution

In this thesis we analyse cross-layer protocol interactions in TCP/IP over Wi-Fi MAC networks and design and evaluate techniques that improve end-to-end goodput. We evaluate the performance of the Wi-Fi MAC protocol using real-world testbeds. In Chapter 2 we provide a generic background and prior research context for this thesis. A more specific background context is provided with each chapter. This thesis makes the following contributions, broken down by chapter, where bold text in parentheses indicates which of the hypotheses from Section 1.1 are addressed:

- In Chapter 3 we design and implement TCP/HACK, a scheme that increases the efficiency of the Wi-Fi MAC by encapsulating TCP ACK information in the Wi-Fi MAC's link-layer ACKs (**H3** and **H5**).
  - We offer an analysis of the capacity of the Wi-Fi MAC protocol for TCP traffic as a function of bit-rate, showing the theoretical goodput gains achievable by avoiding medium acquisitions for TCP ACKs.
  - We evaluate TCP/HACK using a software-defined radio platform and show that TCP/HACK offers up to 32% goodput improvement.
- In Chapter 4 we design and implement AAQ, a fair queueing algorithm that aids in sending large aggregates of frames by exchanging information between the IP and Wi-Fi device driver layers (**H1**).
  - We provide empirical measurements from an indoor IEEE 802.11 network testbed and show that today's IP stack and IEEE 802.11 NICs often fail to send large aggregates of frames. This results in poor goodput.
  - We evaluate AAQ using a real-world indoor testbed and show that AAQ offers up to 27% goodput improvement.

- In Chapter 4 we design and implement TAO, a TCP optimisation technique that allows a TCP receiver to send TCP ACKs in large aggregates on IEEE 802.11 networks (**H4** and **H5**).
  - We evaluate TAO using a real-world indoor testbed and show that TAO offers up to 30% goodput improvement.
- In Chapter 5 we design and implement AARC, a rate control algorithm that chooses the best transmit rate while preserving large aggregates (**H2**).
  - In an evaluation on a real-world testbed, we show that the default rate control algorithm in Linux disrupts frame aggregation significantly, thus reducing goodput.
  - We evaluate AARC in an indoor IEEE 802.11 network testbed and show that the goodput achieved is more then 2x the goodput of the default Linux configuration.

Finally, Chapter 6 concludes this thesis with a discussion and suggestions for future work.

Chapter 2

Background and Related Work

Wireless systems in general and those based on IEEE 802.11 in particular have long been the subject of active research. A large body of research has gone into analysing the behaviour of wireless networks and improving their performance [58, 87, 33, 95, 54]. In this chapter we give an overview of the background related to this thesis. We will provide a general overview here; more detailed background information is provided in each subsequent chapter according to that chapter's focus. We also will summarize some related work in this overview.

2.1 IEEE 802.11 Media Access Protocol

The IEEE 802.11 MAC protocol uses CSMA/CA, based on the Ethernet's CSMA/CD [66], which in turn is based on the ALOHA random access protocol [27]. Further, to combat the classic hidden terminal problem, IEEE 802.11 borrows from MACA [56]. Figure 2.1 shows the basic operation of IEEE 802.11's CSMA/CA protocol, the Distributed Coordination Function (DCF). Before transmitting, stations listen to the medium. If the medium is sensed

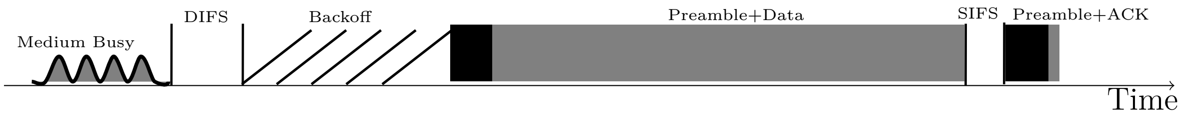


Figure 2.1: 802.11 Distributed Coordination Function

busy then stations defer their transmission. Once the medium becomes idle, for the fixed duration of *DIFS*, stations enter the so called contention period. Each station draws a uniformly distributed random number (*backoff* counter) between 0 and CW. CW starts at some CW_{min} and is doubled every time a transmission fails, up to some CW_{max}. For every timeslot of length T_{slot} that the medium is idle, the *backoff* counter is decremented. Once the *backoff* counter reaches zero, the station initiates transmission. If during the contention period the medium is sensed busy, the *backoff* counter is paused. It is resumed once the medium is sensed free for a *DIFS* period. Every data transmission is preceded with a *preamble* that marks its beginning. The preamble is transmitted using a predefined resilient bit-rate and contains information such as the bit-rate at which the following data is transmitted. After successful reception of the data, the recipient must acknowledge it after waiting for a *SIFS* period by sending an ACK control packet. To determine if the transmission is successful, the receiver computes the Frame Checksum Sequence (FCS), and compares it to the FCS calculated by the transmitter and appended at the end of the data. ACK packets do not contend for the medium as the time for their transmission is reserved by the transmitter. Data packet headers contain a *duration* field, which gives the period of time the transmission over the air lasts. Stations overhearing the transmission defer their own transmission for the *duration* period.[1]

As applications have demanded higher throughput, researchers have explored different ways to improve the existing 802.11 MAC protocol. Xiao and Rosdahl [95] give a theoretical throughput upper limit and a theoretical delay limit for the IEEE 802.11 MAC. They conclude that if one increases the PHY bit-rate without reducing MAC overhead, the improvement in throughput is bounded. Several efforts have enhanced the IEEE 802.11 DCF by adjusting various protocol parameters [36, 92, 37]. *Idle Sense* [48] improves the IEEE 802.11 DCF by ensuring that all stations have the same contention window. Stations converge to the same CW by tracking the number of consecutive *idle*

[1]This mechanism is known as Virtual Carrier Sensing

slots. More recently, in *Wi-Fi-Nano* [64], Magistretti et al. note that most of the channel acquisition overhead in the IEEE 802.11 DCF comes from its contention period backoff. This contention period is divided into slots of 9 μs. In *Wi-Fi-Nano*, time slots are reduced to 800 ns. Most of the work in Wi-Fi-Nano focuses on physical-layer techniques to allow proper operation of IEEE 802.11 over 800 ns time slots. These techniques require major changes to the current IEEE 802.11 hardware design.

Starting with IEEE 802.11a orthogonal frequency-division multiplexing (OFDM) is used for transmission. Briefly, OFDM is a frequency division multiplexing (FDM) scheme that divides a large bandwidth (*i.e.,* 20, 40, or 80 MHz) into small overlapping narrowband channels that do not interfere with each other. For example, a 20 MHz bandwidth is divided into 64 equal sub-channels each 312.5 KHz in width. OFDM is more robust to multi-path fading and is more efficient in using spectrum. A detailed explanation of OFDM is out of scope for this overview and the reader is referred to an introduction and history of OFDM [93].

In *FICA* [84], Tan et al. divide the wide IEEE 802.11 band into narrower sub-bands. A new physical layer architecture is developed in which multiple senders concurrently transmit on these different sub-bands. The narrow sub-bands are then divided into subcarriers, over which OFDM is used for transmission. A new MAC protocol is developed in which senders contend in the frequency domain rather than in the time domain for medium acquisition. By using narrow band transmission the physical data rate is lowered and the channel acquisition overhead lessened relative to the time required for data transmission. Furthermore, by allowing multiple senders to transmit at the same time, acquisition overheads are spread across multiple frame transmissions, further amortising the overhead. Chintalapudi et al. propose *Wi-Fi-NC* [38], a different design for narrow-band channelisation of Wi-Fi. Apart from reducing IEEE 802.11 MAC overhead at high physical layer data rates, *Wi-Fi-NC* also targets improving the efficiency of Wi-Fi networks in scenarios where heterogeneous physical layers must co-exist and available spectrum is fragmented, as with white-space networks [28]. Like *FICA*, *Wi-Fi-NC* requires redesigns of the physical and MAC layers.

To adapt to varying wireless channel conditions, IEEE 802.11 supports multiple PHY transmission rates, each with a different degree of robustness against bit errors. The original IEEE 802.11 DCF gives each station equal transmit opportunities. Tan and Guttag noticed that in a multi-rate environment, the IEEE 802.11 DCF is unfair to stations with a fast sending rate, because they get the same number of transmit opportunities as the stations with the slow sending rate. This leads to less time on the channel for the fast sending rates, and results in poor aggregate throughput. They propose time based fairness [83], where each station gets equal *time*.

IEEE 802.11 does not mandate any means for how stations choose the bit-rate at which to transmit at a particular time. Much research has investigated various algorithms to best track channel quality and use the most appropriate rate. There are a number of rate adaptation (RA) algorithms proposed. Many use various statistics, such as packet error rate or signal-to-noise ratio, to track channel quality and decide on the best bit-rate to use [55, 94, 34]. More recently, with the introduction of IEEE 802.11n, a new wave of RA algorithms has emerged. These algorithms incorporate new MIMO rates and features such as frame aggregation [72, 40, 100].

2.2 Enhancements incorporated in the IEEE 802.11 standard

To meet the demand for high throughput, the IEEE 802.11 standard incorporates some major enhancements to the MAC protocol. The IEEE 802.11e [49] amendment introduced transmit opportunity (TXOP), a form of time-based fairness, mentioned earlier. Using TXOP, stations are free to send back-to-back packets during a single channel acquisition up to the TXOP value. TXOP is a fixed channel time during which a station may transmit. The TXOP is a value broadcast by the Access Point (AP), the central entity managing the Wi-Fi network. In addition, IEEE 802.11e introduces a Block Ack protocol. The Block Ack protocol is a session-oriented protocol in the MAC layer, and is used to acknowledge multiple packets using a single Block Ack frame. To use the Block Ack protocol, stations have to establish a session. A Block Ack session is established by sending an Add Block Ack Request to the desired destination; the receiver responds with an Add Block Ack Response. The Block Ack protocol is a window-based protocol that can acknowledge up to 64 packets in IEEE 802.11n. A Block Ack session is identified by the 3-tuple $< source, destination, tid >$. Traffic Identification (TID) represents the traffic access category (AC). IEEE 802.11e [49] introduced four ACs in an effort to provide Quality of Service (QoS). The four categories, in decreasing order of priority, are *Voice*, *Video*, *Best Effort*, and *Background*. Priority is provided by adjusting the DCF access parameters and the TXOP value. These values are broadcast by the AP and stations must comply with them.

IEEE 802.11n [50] introduced frame aggregation, a major enhancement to the IEEE 802.11 MAC protocol. Frame aggregation works in conjunction with the Block Ack protocol and TXOPs. Two types of frame aggregation are supported: Aggregate MAC Service Data Unit (A-MSDU), and Aggregate MAC Protocol Data Unit (A-MPDU). A-MSDU aggregates multiple packets into a single "jumbo frame" with a single MAC header and a single Frame Checksum Sequence (FCS). A drawback of A-MSDU is the use of a single FCS to protect all the packets in the aggregate. As a result, a single bit error renders the whole A-MSDU unusable, wasting valuable channel time. In our experience we have not encountered Wi-Fi NICs that support A-MSDU transmission. A more robust form of aggregation is A-MPDU aggregation. In an A-MPDU each packet has its own MAC headers and a FCS, and therefore only frames that are in error are discarded provided that the preamble is decoded without error. Each A-MPDU consists of a PHY preamble followed by variable number of frames, separated by a well known pattern called a *delimiter*. The maximum number of frames that can be transmitted in a single A-MPDU is limited by three factors. First is the Physical Layer Convergence Protocol (PLCP) payload size, which is 64K in IEEE 802.11n. This limit includes MAC headers and delimiters. Sending 1500-byte packets limits the number of frames in an A-MPDU to 42. The second limit is set by the TXOP value for the given AC, broadcast by the AP. This value specifies the amount of time in milliseconds a station can transmit continuously with a single channel acquisition. Since the limit is in time, the number of frames that can be aggregated depends on the transmit rate used by the wireless station. The third and final limit is the limit imposed by the Block Ack protocol. A Block Ack can only acknowledge up to 64 frames. All the frames an in A-MPDU must belong to the same AC and be addressed to the same destination. In addition, all frames are sent at the same bit-rate.

In [70] Ozdemir et al. improve upon TXOP to further reduce channel overhead by allowing the recipient of the frame to respond with data frames during the same TXOP reserved by the transmitter, without performing channel acquisition. This enhancement was included in the IEEE 802.11n standard.

2.3 TCP/IP and Wi-Fi

The Transmission Control Protocol (TCP) [4] carries the majority of the traffic in the Internet. Originally it was designed for wired networks. The increased popularity of Wi-Fi sparked interest in the wireless research community to analyse and improve TCP's performance over Wi-Fi links. At the heart of TCP is the congestion control algorithm whose job is to find the optimal sending rate. TCP treats losses as a signal of a congested network and reacts to them by dropping its congestion window size, and thus its transmission rate. However, losses on wireless links can occur as a result of bit errors introduced by the wireless channel. Fu and Liew enhanced TCP's congestion control algorithm to work on wireless links [42]. Their protocol enacts changes to the TCP stack at the sender that improve throughput over wireless links.

Balakrishnan et al. [30] provide a comparison of mechanisms for improving TCP performance over wireless links. They divide these techniques into three broad categories: end-to-end, link-layer, and split-connection solutions. In the end-to-end category, they conclude that TCP with New Reno congestion control performs better then TCP Reno. For the link-layer category, they highlight that link-layer retransmissions can cause TCP to timeout, resulting in poor goodput. Split-connection protocols [98, 29] split TCP connections between sender and receiver into two connections: one connection between AP and sender and one between AP the receiver. These protocols incur extra overhead as each packet goes through the TCP stack twice. In addition, the state kept at the AP is increased significantly.

Pang et al. [71] propose a TCP ACK agent that sits at the AP and generates TCP ACKs on behalf of the wireless client. They use link-layer ACKs to signal to the AP that a TCP data packet has been received successfully. The AP then generates the TCP ACK and forwards it upstream on behalf of the TCP client. Their design is only capable of communicating to the AP when the client observes a TCP ACK for the data just received and their system does not mention if it is possible to generate TCP ACKs with a lower sequence number. This information is needed in the case of loss and as such it prevents delivery of duplicate ACKs, and prevents the use of fast retransmit, leaving only inefficient TCP timeouts. Furthermore, they don't mention how their system handles cases when TCP options are present, such as TCP timestamp options.

Chapter 3

Hierarchical ACKs for Efficient Wireless Medium Utilisation

3.1 Introduction

As described in Chapter 1, in Wi-Fi 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 desynchronise would-be concurrent senders. To use a concrete example, Distributed Channel Access (DCA) in 802.11g [7] enforces an average idle period of 110.5 μs before a frame's transmission, whereas a 1500-byte payload itself lasts only 222 μs at 54 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, Wi-Fi 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 amortise the significant overhead of medium acquisition over multiple data frames, 802.11n's MAC protocol batches multiple data frames into a single aggregated frame (A-MPDU), and incurs only a single medium acquisition for each such batch. IEEE 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.

Wi-Fi'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. This chapter describes TCP/HACK over IEEE 802.11a as implemented on SoRa Software Defined Radio Platform (SDR). For the implementation and evaluation of TCP HACK over IEEE 802.11n in simulation, see [75].

3.2 Problem and Design Goals

There are two distinct facets to improving the efficiency of the Wi-Fi MAC layer for TCP transfers at fast bit-rates. First, we must understand the overhead of medium acquisition in Wi-Fi 802.11a 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 TCP/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 Wi-Fi MAC protocol when optimising across

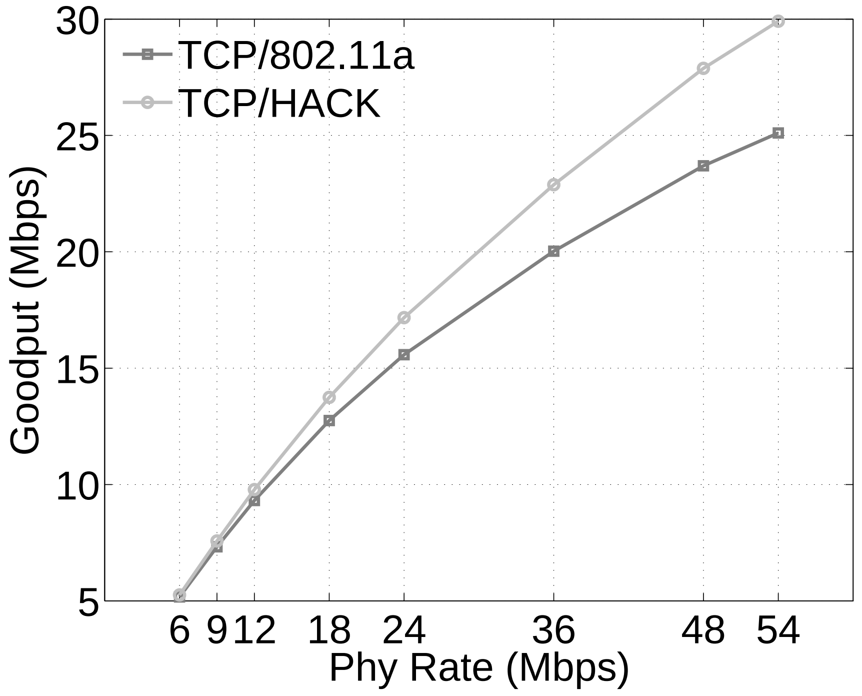


Figure 3.1: Theoretical goodput for 802.11a.

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.

3.2.1 Wi-Fi MAC Overhead

Consider a typical Wi-Fi use scenario, where a single 802.11a 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. Note that this assumption is the best case for the efficiency of the status quo Wi-Fi MAC—were delayed ACK not used, a TCP receiver would generate twice as many ACK packets, and the Wi-Fi MAC would incur significantly more medium acquisitions.

In Figure 3.1, the curve labeled "TCP 802.11a" shows analytical predictions of the goodput a single TCP downloader achieves as a function of physical-layer bit-rate on lossless 802.11a networks. These analytical predictions are based on the parameters of the 802.11a MACs. Figure 3.1 also shows the improved goodput achieved by HACK, our modified 802.11 MAC protocol that carries

TCP ACKs in link-layer ACKs, which we describe in Section 3.3.

$$TCP_{util} = \frac{2 \times T_{U_{data}}}{2 \times T_{data} + \underbrace{3 \times (T_{DIFS} + T_{SIFS} + T_{BO} + T_{DACK}) + T_{TACK} + (6 \times T_{PLCP})}_{T_{Overhead}}} \tag{3.1}$$

Equation 3.1 models IEEE 802.11 MAC utilisation when using TCP. $T_{U_{data}}$ is the time it takes to transmit user data at a given bit-rate, and T_{data} is the time it takes to transmit user data including TCP and IP headers. Let T_{DIFS}, T_{SIFS}, T_{BO}, and T_{DACK} be DIFS, SIFS, average back-off, and link-layer ACK times, respectively. TCP sends a TCP ACK for every two TCP segments. TCP ACKs are treated as a regular data transmission by the 802.11a MAC. Transmitting two TCP segments incurs three channel acquisitions as IEEE 802.11 MAC layer. Also, transmitting bytes of the TCP ACK over the wireless medium takes some time, represented by T_{ACK}. Finally, MAC protocol frames get a PHY layer preamble prepended incurring further costs (T_{PLCP}). TCP/HACK eliminates the need to acquire the channel for transmitting TCP ACKs, simplifying Equation 3.1 to Equation 3.2.

$$HACK_{util} = \frac{T_{U_{data}}}{T_{data} + \underbrace{(T_{DIFS} + T_{SIFS} + T_{BO} + T_{DACK_{HACK}}) + (2 \times T_{PLCP})}_{T_{Overhead}}} \tag{3.2}$$

To calculate the goodput shown in Figure 3.1, the PHY rate is multiplied with the utilisation from Equations 3.1 and 3.2. These models present the best-case scenario in which the channel does not corrupt packets (lossless channel) and there are no collisions. Table 3.1 shows the parameters used to generate Figure 3.1.

Parameter	**Value**	**Description**
T_{data}	$222\mu s - 2ms$	Time to transmit 1500-byte packets for 802.11a bitrates
T_{SIFS}	16	Short Interframe Spacing time in μs
T_{slot}	9	Timeslot in μs
T_{DIFS}	34	DCF Interframe Spacing time in μs. $T_{DIFS} = T_{SIFS} + (2 \times T_{timeslot})$
T_{BO}	67.5	Expected backoff time in μs. $T_{BO} = 7.5 \times T_{timeslot}$
T_{DACK}	22.6	Time in μs to transmit Datalink ACK at 12Mbps
$T_{DACK_{HACK}}$	22.6-25.3	Time in μs to transmit Datalink ACK + HACK at 12Mbps
T_{TACK}	$11\mu s - 101\mu s$	Time to transmit TCP ACK at 802.11a bitrates
T_{STF}	8	Short Training Field in μs
T_{LTF}	8	Long Training Field in μs
T_{SIGNAL}	4	SIGNAL in μs
T_{PLCP}	20	Physical Layer Convergence Procedure overhead. $T_{PLCP} = T_{STF} + T_{LTF} + T_{SIGNAL}$

Table 3.1: 802.11a goodput calculation parameters

3.2.2 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 Wi-Fi receiver must reply to a data packet with an LL ACK within SIFS, an interval defined in the 802.11a specification as 16 μs. That deadline is of course far too short to meet in host software, so Wi-Fi NICs validate received frames and generate LL ACKs in hardware. TCP/HACK must comply with these same LL ACK deadlines imposed by today's Wi-Fi 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 Wi-Fi 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 minimise 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 minimise the complexity and thus the cost of the NIC, but also because we would like HACK to generalise to other higher layers than TCP such as SCTP [82] or DCCP [59]: if the NIC treats the feedback to be appended to an LL ACK as opaque bits that it needn't understand, then HACK should generalise in this way.

No changes to TCP

TCP changes are difficult to standardise and difficult to deploy, as many widely used OSes ship with a single closed-source TCP implementation. TCP/HACK should preserve the end-to-end TCP funcionality, ensuring the flow on receive and transmit paths remain unchanged. Both at clients and APs, HACK-related functionality should be confined to the Wi-Fi 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 goodput). TCP/HACK must not disrupt the timely return of correct TCP ACKs to the sender.

3.3 HACK Design

We first offer an overview of TCP/HACK's design and then we explore nuances of the cross-layer interactions between TCP and IEEE 802.11a, 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."

3.3.1 HACK in Overview

When a regular TCP client receives a TCP data packet, its network stack generates a TCP ACK and enqueues it for transmission by the Wi-Fi NIC. Under TCP/HACK, a client does not immediately enqueue a TCP ACK for transmission. Instead, the client compresses each TCP ACK and appends it 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 recognises an "augmented" LL ACK, which it decompresses, reconstitutes the encoded TCP ACKs, and forwards them upstream.

3.3.2 Cross-Layer Nuances

We now refine our design to handle the subtle cross-layer interactions that arise between TCP and IEEE 802.11. In principle, we would like to encapsulate TCP ACKs on the LL ACKs of the TCP packets they acknowledge. For example, if a TCP segment arrives, the client would like to piggyback the TCP ACK for that TCP segment in the LL ACK for that TCP segment. However, the 16μs SIFS interval between receiving data and sending the LL ACK is too short for the host's TCP stack to turn around the TCP ACK, compress it, and DMA it to the NIC. For HACK to be practical, the compressed TCP ACKs will have to wait until the next data arrives, and piggyback on its LL ACK. It turns out that this significantly complicates the dynamics of TCP/HACK and we will explore the consequences.

Figure 3.2 illustrates this process[1]. In response to a data frame containing TCP packet 1, TCP ACK 1 arrives at the client's transmit queue too late to be carried on that frame's LL ACK. Instead, the TCP ACK is compressed but not yet sent. When the next data frame carrying TCP packets 2 arrives, its LL ACK can now carry the compressed frame with TCP ACK 1. The AP then reconstitutes the full TCP ACK and passes them up the network stack.

So long as TCP data packets continue to arrive, there is a steady stream of LL ACKs on which to piggyback compressed TCP ACK frames: all TCP ACKs are carried as TCP/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 3.2, TCP ACK 2 meets this fate, and is in turn LL ACKed.

Figure 3.1 summarizes the theoretical upper bound on TCP/HACK goodput on IEEE 802.11na. The curves assume that TCP/HACK manages to encapsulate all TCP ACKs in LL ACKs, and that the compression is performed using the algorithm in Section 3.3.3.2. As the bit-rate increases, TCP/HACK significantly improves capacity, with a 20% improvement seen at 54 Mbps. 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

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

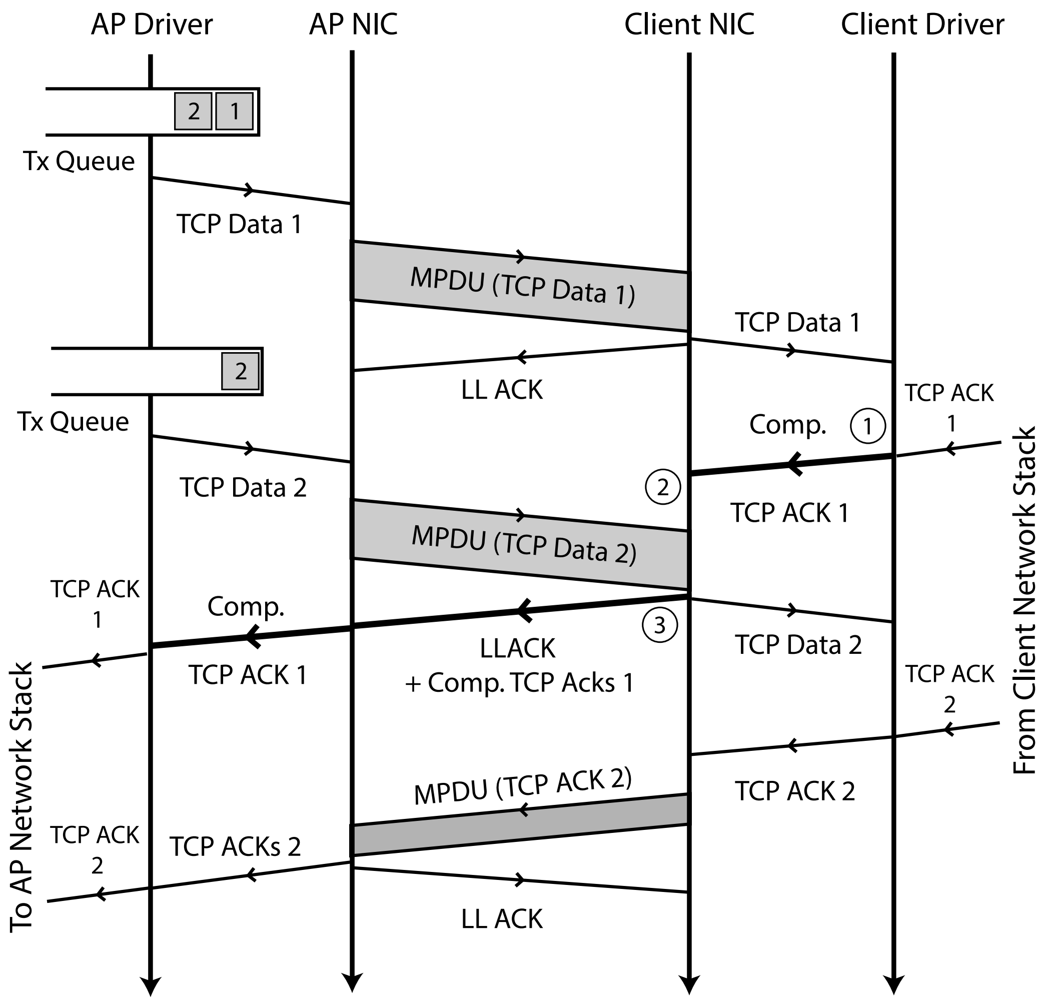


Figure 3.2: Interaction between Data, LL ACKs and encapsulated HACK packets

TCP/HACK sidesteps.

3.3.2.1 To HACK or not to HACK?

To maximise 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? 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 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, TCP/HACK can be used, but otherwise vanilla TCP ACKs will be sent. Such a design may often squander the opportunity to use TCP/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 3.2, initially there are two data packets queued at the AP. When the AP sends the first data frame, it already knows more data will be sent to that client, as it already has packet 2 in its queue. So long as the AP has more packets queued, 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 IEEE 802.11 header of the data frame. [2]. 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 TCP/HACK to send them.

3.3.3 HACK in Practice

In the preceding section, we have presented a conceptual description of TCP/HACK, but several questions concerning the practicality of this con-

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

ceptual design remain unanswered. First, how realisable is TCP/HACK given current systems and hardware? In particular, how should TCP/HACK's functionality be divided between a station's NIC hardware and NIC device driver? Finally, what manner of compression should TCP/HACK employ to reliably encode the TCP ACKs?

3.3.3.1 Driver and NIC Functionality

We envision to realise TCP/HACK (including the More Data mechanism) with very few changes to a station's IEEE 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 3.2. Our discussion is in the context of a modern Linux wireless driver, such as the Atheros ath9k driver [12]

The 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.

The Client

The client's driver needs to determine when it can use TCP/HACK and when it must send TCP ACKs normally. In Figure 3.2, on receiving packet 1, the client's NIC also passes the More Data state to the driver. The client's TCP stack acknowledges the data, generating TCP ACK 1, and puts them in the transmit queue at point ①.

Figure 3.3 shows what happens at points ① and ② from Figure 3.2 in more detail. If the driver is not in the More Data state, it simply enqueues these TCP 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 MAC destination address is needed to match compressed TCP ACKs with LL 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

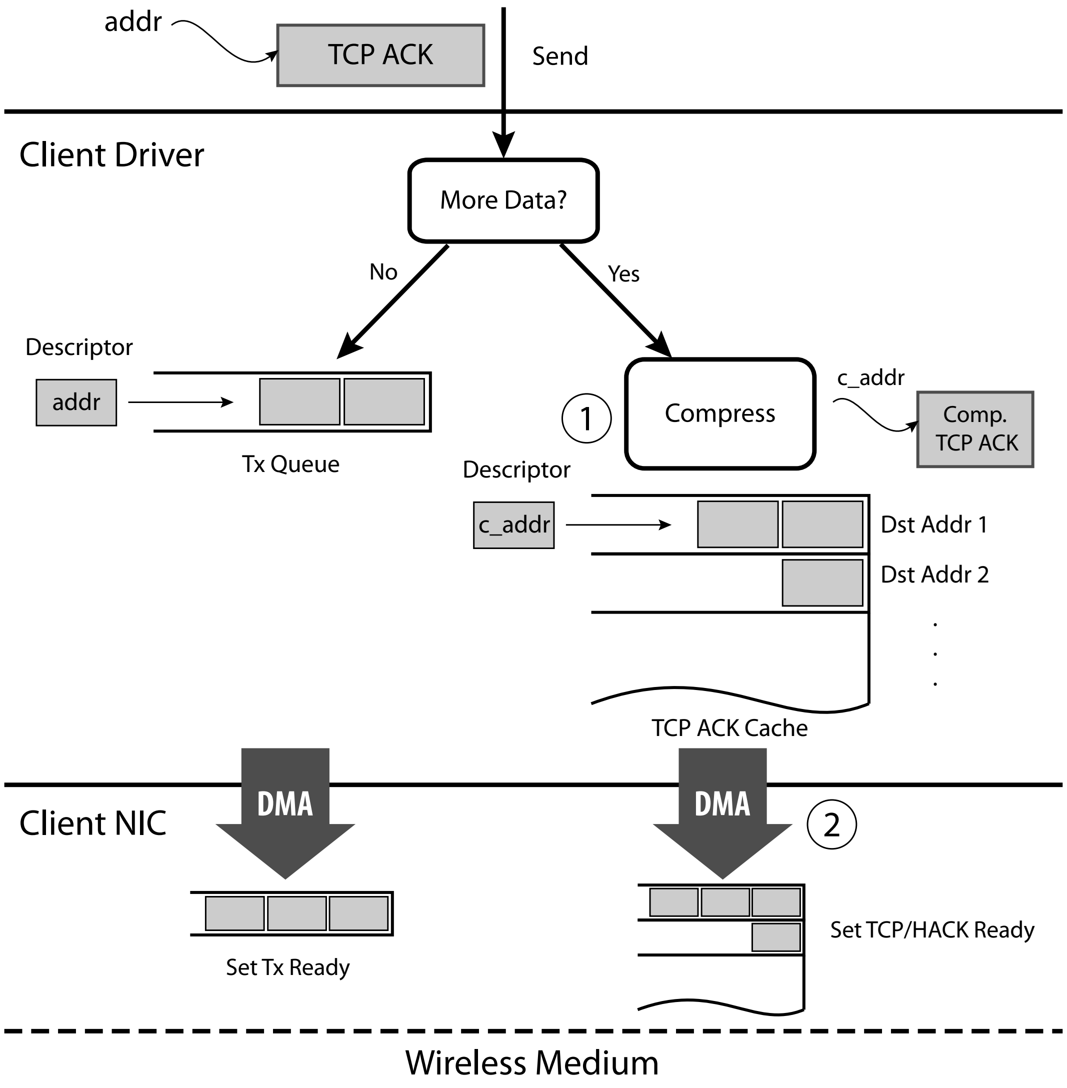


Figure 3.3: Client-side TCP/HACK compressing a TCP ACK, ready to be sent on the link-layer acknowledgment of the next frame.

indicate that TCP/HACK is ready.

Figure 3.4 shows what happens when the next frame 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 grabs one of the frames, and appends it to the returning LL 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 frame carrying

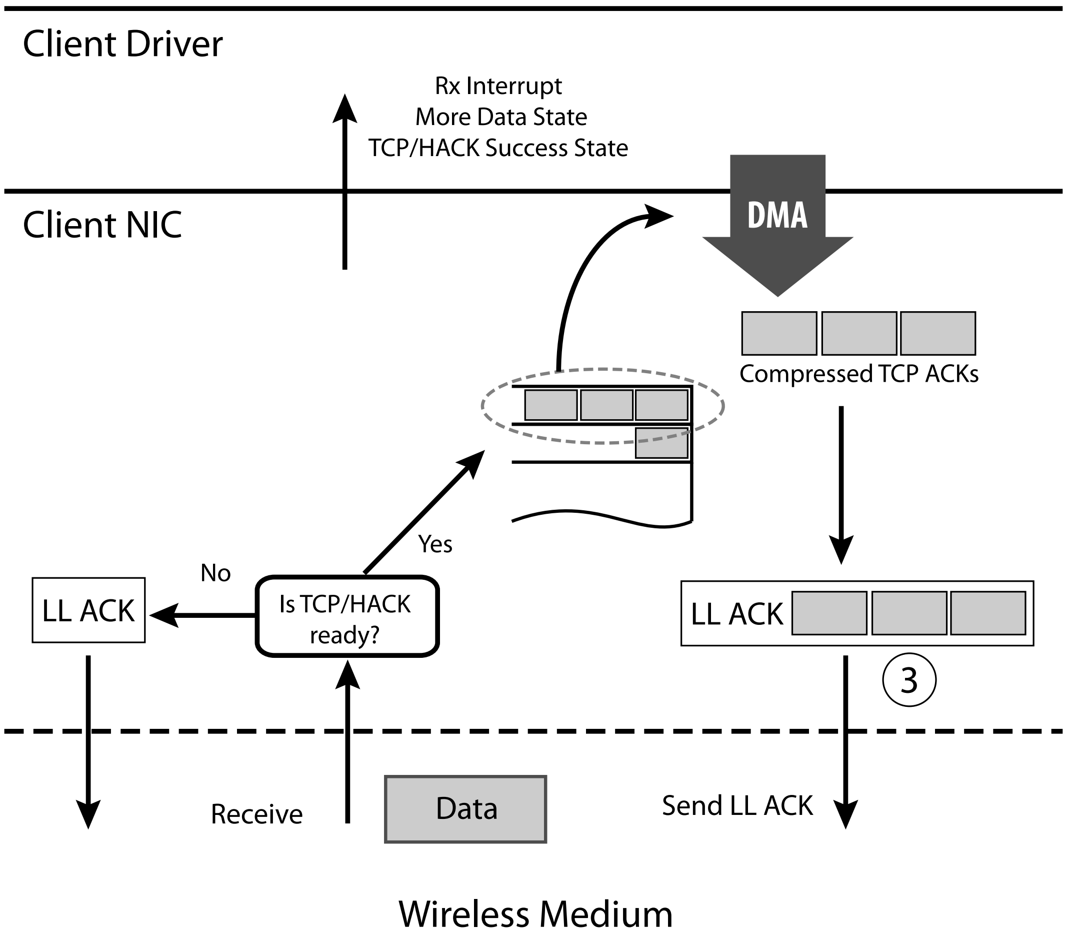


Figure 3.4: Client-side TCP/HACK receiving a data frame from the air and including compressed TCP ACK frames in the corresponding LL ACK.

packet 2 arrives with the More Data flag not set before the driver has succeeded in conveying compressed TCP ACK 1 to the NIC. In this case, the TCP/HACK "ready" check will fail. The NIC sends a normal LL 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.

The AP (ACK reception)

Finally, the AP needs to recognise and decompress the "augmented" LL ACKs. The task of recognition falls to the AP's NIC, which extracts the compressed TCP ACK frame from the received LL 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.

3.3.3.2 TCP ACK Compression

A critical component of the design is choosing a compression method for TCP ACKs. As IEEE 802.11a transmits LL ACKs at one of the lower basic rates, *e.g.*, 6 Mbps, it is desirable to minimise the size of the TCP ACK information appended to LL ACKs. Moreover, the IEEE 802.11a MAC protocols' DIFS interval protects "stock" LL ACKs from collisions. Ideally, the compressed TCP ACK information that TCP/HACK appends to LL ACKs should be short enough to fit within DIFS, to avoid risking a collision. 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)* [73] 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 optimisations

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

1. We do not explicitly send Initialise-Refresh (IR) packets from the TCP

client to the AP. To initialise 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 [74] hash over the ACK's 5-tuple and selects the lowest byte as the CID.
3. Compressed TCP ACK packets encapsulated within LL 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 3.3.4.

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) [73].

3.3.4 Avoiding Cross-Layer Pathologies

The protocol we have described so far works well in a lossless environment. When applying TCP/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 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 synchronised. Packet losses may cause loss of synchronisation of this state, and in turn cause CRC failures on decompressed TCP ACK packets. Such loss of synchronisation must not be persistent.

Loss of LL ACK.

Consider the scenario in Figure 3.5, where a LL ACK carrying compressed TCP ACK information cannot be decoded. In this scenario, to deliver compressed TCP ACKs reliably, the client must retain them until it determines

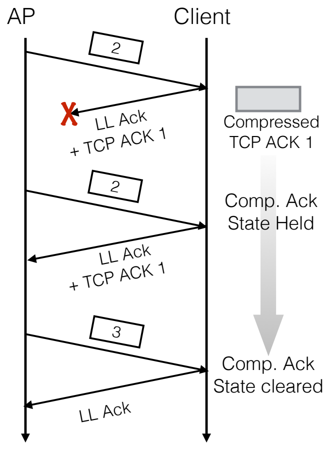


Figure 3.5: Coping with loss of LL ACKs by retaining TCP ACK state.

that its 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 LL ACK in response to a frame, as in Figure 3.5, the client can be certain that the AP has received its LL ACK upon receiving an frame with a greater MAC-layer sequence number—if the AP has not done so, it must instead retransmit the frame 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.

3.4 Evaluation

TCP/HACK works for IEEE 802.11a and IEEE 802.11n. We show here the results from the evaluation of TCP/HACK using the SoRa software-defined radio platform which only supports IEEE 802.11a. For completeness we briefly present simulation results of TCP/HACK running over a IEEE 802.11n network. For more details the reader is referred to HACK [75].

3.4.1 SoRa Implementation

We implemented TCP/HACK including the More Data bit and ROHC compression for the SoRa user-level physical layer on the Windows 7 operating system. Hardware limitations of our SoRa radio boards require us to run IEEE 802.11a in the 2.4 GHz band, but this does not affect protocol behaviour. We begin with an introduction to the SoRa SDR platform and some of the challenges we faced implementing TCP/HACK. Then we explain our TCP/HACK implementation.

3.4.2 A Brief Introduction to SoRa

SoRa [85] is a software-defined radio platform on a commodity PC. It enables the development of signal processing systems using high-level programming languages such as C and C++. SoRa relies on the PCI Express bus for high-speed data transfers between the radio front end and the host CPU. It makes heavy use of multi-core CPUs in today's commodity PCs for fast signal processing. The SoRa platform consists of three main components:

1. **Radio Control Board** (RCB): driven by a Xilinx [97] Field-Programmable Gate Array (FPGA) attached to the host PC using the PCI Express interface. It is responsible for low-level signal processing, such as transmitting and receiving signals over the wireless channel.

2. **Device Driver**: bridges the Windows operating system and the RCB.

3. **Software Development Kit** (SDK): a high-level API used by applications to control the RCB.

Out of the box SoRa includes user-level libraries/applications that implement the IEEE 802.11b/a/g PHY and a partial MAC protocol. We implement TCP/HACK in C++ on top of the provided user-level application. Implementing TCP/HACK on the SoRa platform proved more challenging then we'd anticipated. We next describe the difficulties that arose and how we addressed them. Additionally we explain some of the API functions used when building on SoRa.

Contention and Latency

Before transmission, signals must be explicitly transferred from the host PC's memory to the RCB's internal memory. To initiate such a transfer, the function *SoraURadioTransferEx()* [3] must be called. Once signals have been transferred and reside in the RCB's memory, they are transmitted by calling the *SoraURadioTx()* function. Therefore, each wireless transmission incurs two function invocations to the SoRa API. While validating the correctness of the IEEE 802.11a MAC implementation we noticed contention between these two functions. We observed that once a call to *SoraURadioTransferEx()* has occurred, subsequent calls to *SoraURadioTx()* must wait for the *SoraURadio-TransferEx()* function to complete. The order of calls does not matter; the latter function call must wait for the prior function call to finish before it is executed. To validate our hypothesis we wrote a two-threaded application. We instructed the first thread to continuously transfer a pre-defined signal (using *SoraURadioTransferEx*) to the RCB's memory while the second thread continuously transmits (*SoraURadioTx*) the signal over the wireless channel. Using timestamps we measured the time it took for each function to complete. In addition, when the second thread transmits the signal, we measure the elapsed time since the transfer (*SoraURadioTransferEx*) function was called. Figure 3.6 shows the time in microseconds for the transmit function to complete (y-axis) as a function of elapsed time since the transfer function was called (x-axis). These results suggest that SoRa serialises calls to RCB functions, causing each function to wait to execute until prior functions complete their execution. This causes the the transmit function to take longer to complete when there is an ongoing transfer function call. In Figure 3.6, a value of zero on the x-axis represents cases when the transmit function was invoked before the transfer function. Even in this case, transmit time for a signal takes on a range of values, suggesting that other function calls are in contention too. The closed-source nature of the SoRa's RCB firmware makes it hard to confirm if this behaviour

[3]This function assigns a unique *ID* to each transfer. The *ID* can be used later to manipulate the signal, i.e. to transmit and/or delete it.

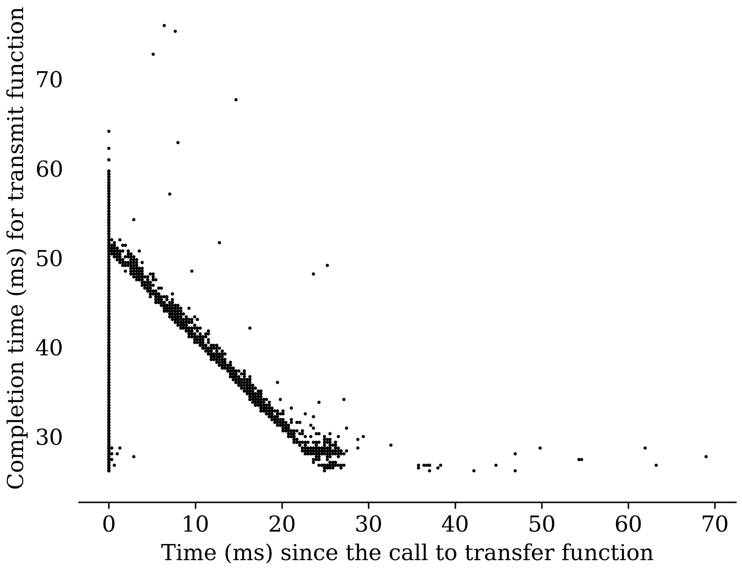


Figure 3.6: SoRa contention between function calls

is by design or simply a bug. One consequence of the results in Figure 3.6 is the potential delay in returning LL ACKs. We know that IEEE 802.11 LL ACK must be returned shortly after a SIFS time. To find out the time it takes for SoRa to return LL ACKs, we measure the time between the end of the data frame transmission and the receipt of the corresponding LL ACK. The distribution of the time it takes to receive a LL ACK is shown in Figure 3.7. For comparison we also show measurements for three different commercial Wi-Fi NICs. To measure LL ACK delay we transmit UDP data to Wi-Fi receivers equipped with various Wi-Fi NICs and capture the wireless traffic on a third monitoring station using *tcpdump*. We post-process the captured data offline and calculate the time difference between LL ACK reception and the end of the data transmission. This time difference is calculated using MAC timestamps provided by the device driver. Each received frame is timestamped by the NIC with a hardware clock running at microsecond granularity. The median time to return LL ACKs for SoRa is 55 μs (mean 72 μs). This is greater then the median $27-28$ μs (mean $28-35$ μs) time measured for commercial Wi-Fi NICs. To accommodate this latency we must increase the IEEE 802.11 LL

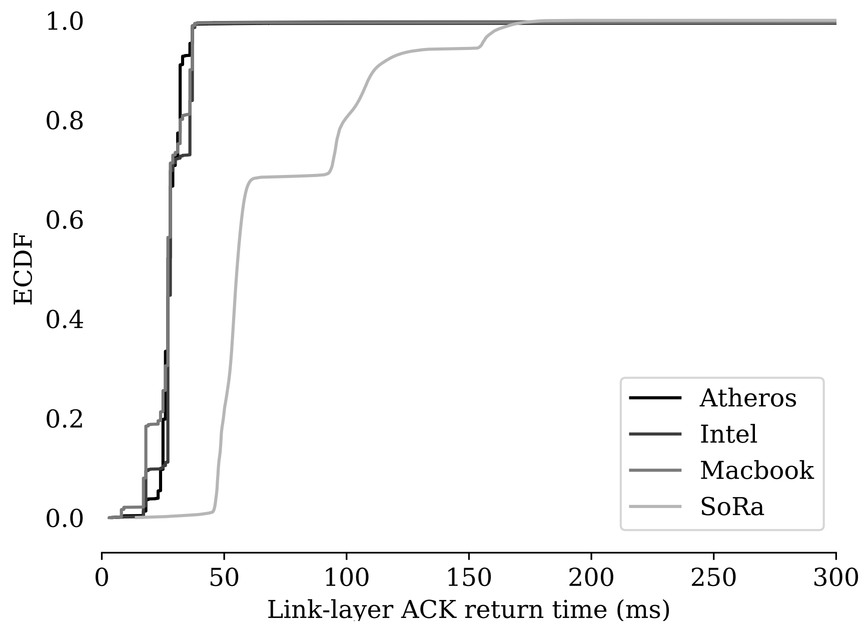


Figure 3.7: Empirical CDF of the time it takes to return LL ACKs.

ACK timeout.[4].

SoRa LL ACK delay and TCP

When running TCP on SoRA the LL ACK delay problem explained above is amplified. Occasionally, LL ACKs are delayed longer then the timeout value, causing the MAC layer to retransmit the frame. These retransmissions (more then two retransmissions) cause the receiver to send more LL ACKs, which are delayed too, leading to a burst of LL ACK transmissions. When the first LL ACK is received, the next frame (a new frame) is transmitted. This new frame collides with delayed LL ACKs, causing the receiver to drop the frame. In the meantime a delayed LL ACK, originally sent in response to a retransmission, arrives at the transmitter. This delayed LL ACK is mistaken for a LL ACK for the newly transmitted frame, causing the transmitter to move to the next frame transmission. TCP only learns about the loss after it receives three duplicate TCP ACKs. Upon receiving three duplicate ACKs, TCP enters the fast-retransmit procedure: it reduces the congestion window, effectively halving the goodput. This problem occurred frequently enough to cause a

[4]This timeout value is the amount of time the transmitter waits for LL ACKs from the receiver before declaring the transmission has failed and retransmiting. We set the timeout value to 192 μs to account for the tail of the distribution.

non-negligible reduction in the goodput achieved when running TCP on the SoRa platform. This is caused, presumably, by the increased congestion at the RCB board created by the transmission of TCP ACKs by the receiver. To fix this problem we add a one-bit[5] sequence number to LL ACKs. On the transmit side, the LL ACK sequence number is checked against the lowest bit of the frame's sequence number. If there is a match then the transmitter knows that the LL ACK is indeed for that frame and can move on to the next data frame. Otherwise, the transmitter assumes the frame was lost and retransmits the frame.

Automatic Gain Control SoRa does not implement hardware automatic gain control, which precludes the use of SoRa under rapidly varying channel conditions.

3.4.3 Testbed

Our three wireless nodes each have four-core Intel Core i7 CPUs, with 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. Hardware availability limits us to a testbed with only two clients. The SoRa interface operates in ad-hoc mode to eliminate periodic beacon transmissions. We run experiments on IEEE 802.11 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 IEEE 802.11a rate. While machines are located physically close to each other (see Figure 3.8), the transmit power is set to low, so there is still a possibility for other IEEE 802.11 sources to interfere. To ensure that there is no outside interference in our masurements, we continuously monitor the channel during the course of the experiments.

3.4.4 Results

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

[5]One bit proved to be sufficient in our experiments.

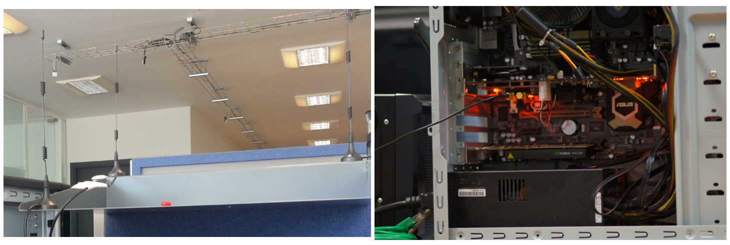


Figure 3.8: *Left:* the TCP/HACK testbed consists of three antennas spaced at close distances, each connected to a SoRa-equipped desktop computer (*right*).

- Are TCP/HACK's capacity benefits in line with theoretical predictions?
- When an AP sends TCP flows to two clients, does TCP over IEEE 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 3.9 compares the application-level goodput 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 IEEE 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 goodput 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 minimises medium acquisitions, and achieves the greatest goodput possible on SoRa with LL ACKs enabled. In an ideal IEEE 802.11 MAC, UDP would achieve 30.2 Mbps; on SoRa, UDP averages 26.5 Mbps across the three experiments. SoRa's LL ACK delays alone reduce the attainable goodput to 28.1 Mbps, and our UDP measurements approach that figure.

If TCP/HACK encapsulated all TCP ACKs in LL ACKs, it would achieve

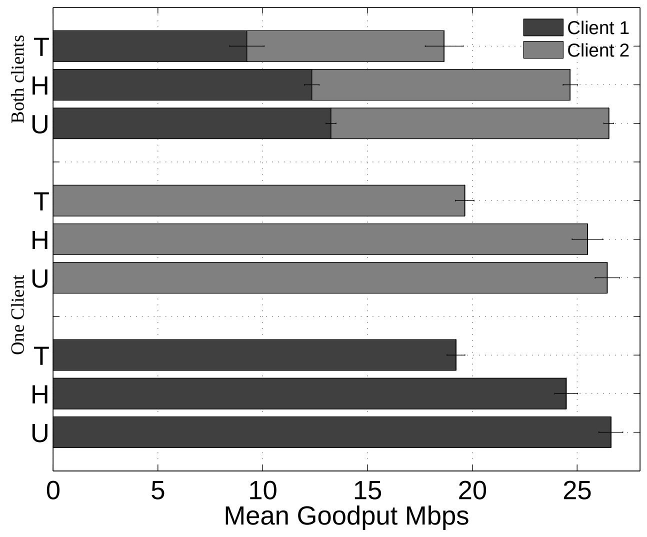


Figure 3.9: TCP goodput with stock 802.11a (T), TCP with HACK (H), and UDP (U) with stock 802.11a, with 1 and 2 clients.

almost the same goodput as UDP (though UDP's packet headers are smaller). In practice, TCP/HACK's single-client goodput 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 3.2: 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 goodput more than predicted analytically in Section 3.2.1. That prediction focused solely on saving medium acquisitions for TCP ACKs. In Table 3.2 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 LL 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**	**$\mathbf{ACK_C}$ count**	**$\mathbf{ACK_C}$ bytes**	**Comp. ratio**
TCP/802.11a	9060	471120	0	0	(1)
TCP/HACK	10	520	9050	39478	12

Table 3.3: 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 3.3 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 achieves. Reducing the number of transferred bytes is beneficial, but TCP ACKs are treated as regular data when sending over IEEE 802.11a 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.

Table 3.4 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

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

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 IEEE 802.11a 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 we simulated IEEE 802.11a in ns-3 with the same packet loss rate as 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 goodput of 22 Mbit/s, which matches the 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 the simulation.

TCP/HACK and IEEE 802.11n Frame Aggregation

We presented TCP/HACK as implemented on the SoRa platform for the IEEE 802.11a MAC protocol. In the interest of completeness we present ns-3 simulation results for TCP/HACK running using frame aggregation and the IEEE 802.11n MAC protocol.[6]

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 1500-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 3.10 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. Two variants of TCP/HACK are shown: Opportunistic TCP/HACK and TCP/HACK More Data. 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 The TCP/HACK More Data variant achieves the most pronounced goodput gain over unmodified IEEE 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 a physical rate of 150 Mbps.

3.5 Personal Contribution

TCP/HACK was joint work with Lynne Salameh, a co-author on our joint

[6]These results, which are from experiments concluded by a cooauthor on our conference publication [75], are presented with my coaugthors' permission.

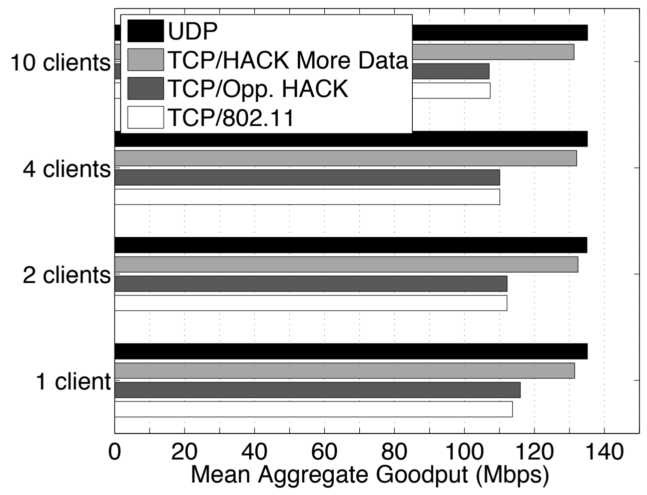


Figure 3.10: TCP goodput for different transmission schemes with 1–10 clients, and UDP for comparison.

paper [75]. Core components of the design, like the More Data bit and delaying the TCP ACK until the next data packet arrives, were developed in collaboration. My personal contributions to the project are as follows: Working on the implementation of HACK in the SoRa platform put me in a unique position to observe TCP/HACK's interactions with real-world wireless channel. As such, I noted that packet losses caused by the wireless channel led to ROHC desynchronisation. This desynchronisation led to TCP connection failures as a result of reconstructing incorrect TCP ACKs. I proposed the base-case solution for IEEE 802.11a which retains the compressed TCP ACK at the receiver until receiver knows that the LL ACK has been correctly received. Lynne further extended this approach to cover all the cases that arise when running TCP/HACK in IEEE 802.11n. Implementing and analysing TCP/HACK on the SoRa platform were entirely my contributions. This includes discovering and fixing the various SoRa limitations described in Section 3.4.1.

3.6 Discussion

TCP/HACK helps reduce the time, otherwise wasted, on unnecessary Wi-Fi medium acquisitions. TCP/HACK relies on the More Data bit to know when it is safe to compress TCP ACKs and wait for another packet on whose LL

ACK to piggyback. With sufficient buffering at the AP and a large window, TCP/HACK works well. In such cases the wireless medium is busy, and MAC 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 in the queue to allow the MORE DATA bit to be set. In this case TCP/HACK will not work well. Similarly, if an AP has very many clients, it may not buffer enough packets for each client for TCP/HACK to work well. In this chapter, we described the design and implementation of TCP/HACK for TCP flows and the IEEE 802.11 MAC TCP/HACK eliminates most of the expensive medium acquisitions that TCP ACK packets require, increasing TCP flow's wireless goodput. The design of TCP/HACK allows it to be transport protocol-agnostic; it could work with other transport protocols such as QUIC [60]. TCP/HACK as presented in this chapter has been implemented for the IEEE 802.11a MAC which does not support frame aggregation. TCP/HACK also works with IEEE 802.11n frame aggregation. TCP/HACK improves goodput even further when used with IEEE 802.11n frame aggregation. Frame aggregation and other previous approaches reduce the cost of individual medium acquisitions [38, 64, 84], TCP/HACK eschews many medium acquisitions entirely. It is thus complementary to these prior approaches. Our evaluation in a real-world implementation demonstrates TCP/HACK's goodput improvements. Our joint work offers a full description of a TCP/HACK implementation for IEEE 802.11n with aggregation refer to [75]. We have released our TCP/HACK implementation as open source and can be downloaded from `http://www0.cs.ucl.ac.uk/staff/A.Zhushi/hack/`.

Chapter 4

Keeping Wi-Fi Fast with Aggregation Aware Queueing

4.1 Introduction

In the previous chapter we presented TCP/HACK, a system that reduces IEEE 802.11 MAC protocol channel access overheads for TCP flows. We showed that TCP/HACK eliminates most wireless channel acquisitions when a TCP receiver transmits TCP ACKs, resulting in goodput improvements. In addition, using simulations we showed the goodput performance of TCP/HACK when using frame aggregation. However, these evaluations assume that the AP and Wi-Fi clients always send maximum sized-aggregates. The focus of this chapter is the behaviour of frame aggregation, introduced in IEEE 802.11n. As Wi-Fi PHY bit-rates increased, applications' transfer rates become increasingly hostage to Wi-Fi's inefficient Medium Access Control (MAC) layer, which incurs a fixed-duration overhead for each medium acquisition. IEEE 802.11n, -ac, and the latest IEEE 802.11ax use *frame aggregation*, which amortises a single medium acquisition's overhead across the transmission of multiple frames, to achieve transfer rates that can begin to approach the hardware's greatest PHY bit-rates. Unless senders on a modern Wi-Fi network manage to transmit aggregates of maximum size (ranging from 42 to 64 packets in IEEE 802.11n), their throughputs in practice will not approach the top bit-rates achievable by the PHY. Unfortunately, the layered design of modern network protocol stacks (and network interface drivers) is oblivious to the groupings of packets presented to a network interface for transmission. In this chapter, we demonstrate that under heavy TCP load on the Wi-Fi downlink, today's mainstream Linux, a commercial Wi-Fi AP, and client implementations typically send Wi-Fi aggregates short enough to keep transfer rates significantly below those supported by the modern Wi-Fi PHY. Using an evaluation testbed we identify and characterise the problems that lead to small aggregate sizes and from the insights gathered, we design and implement Aggregation-Aware Queuing (AAQ) and TCP ACK Optimisation (TAO), lightweight approaches that coordinate packet processing decisions in the TCP, IP queuing, end-host MAC, and Wi-Fi driver layers to ensure that a sender transmits full-sized aggregates. AAQ allows open-source and commercial Wi-Fi APs under heavy downlink TCP load to reclaim the vast majority of the PHY capacity left on the table in the status quo. Before describing details, we first provide an overview of how the frame aggregation mechanism operates in IEEE 802.11n. We describe frame aggregation in the context of IEEE 802.11n [50], as our testbed consists of hardware that supports only IEEE 802.11n. Frame aggregation is also supported in IEEE 802.11ac [10] and IEEE 802.11ax [23], with the only difference the support of even larger aggregates in the later standards.

4.2 Frame Aggregation

Frame aggregation is one of the key enhancements introduced in the IEEE 802.11n standard. It improves channel utilisation by sending multiple frames, back-to-back, in one channel acquisition. IEEE 802.11n supports two frame aggregation schemes: *Aggregated MAC Service Data Unit (A-MSDU)* and *Aggregated MAC Protocol Data Unit (A-MPDU)*. MSDU represents a data unit that the IEEE 802.11 MAC layer receives from the layer sitting above it (i.e, IP). When using A-MSDU, multiple data units from the upper layer are "glued" into one "big" IEEE 802.11 MAC frame. Each MSDU in the aggregate retains headers from the layer above, however they all share a single IEEE 802.11 MAC layer header and are protected by a single FCS (Frame Checksum Sequence). By "wrapping" all upper layer frames into a single IEEE 802.11 MAC frame significant space is saved (saving up to 36 bytes per frame in a A-MSDU[1]). However, single-FCS protection can seriously impact performance in the face of error-prone wireless channels. A single bit-error (after error correction) will cause all frames in the aggregate to be dropped and retransmitted, incurring significant overhead. The IEEE 802.11n standard mandates that all IEEE 802.11n receivers must be able to receive A-MSDU frames. Our experience with various IEEE 802.11n hardware, A-MSDU is not used for transmission. However, research has shown that A-MSDU performance suffers under lossy wireless channel and that the A-MPDU scheme is superior [44, 81].

When aggregating frames in an A-MPDU, by contrast, each frame (MPDU) is encapsulated with IEEE 802.11 MAC headers, and is protected by an individual FCS. Protecting each MPDU with a FCS makes the A-MPDU scheme more robust to wireless channel errors, as only those frames that fail FCS need to be retransmitted. This makes the A-MPDU scheme preferable when transmitting over lossy wireless channels. We focus on the A-MPDU scheme in this thesis, as it is supported by the hardware we use in our testbed.

For its operation, the A-MPDU scheme enlists the aid of the Block Ack protocol described in Chapter 4, Section 4.3 . Originally introduced in IEEE 802.11e [8], the Block Ack protocol is a session-based protocol that allows multiple frames to be acknowledged by a single acknowledgment (ACK) control frame. In the legacy IEEE 802.11 ACK scheme, each successfully received data frame is ACKed individually. Sending an ACK for every frame in an A-MPDU is inefficient. Instead A-MPDU aggregation is used in conjunction with the Block Ack protocol which, allows ACKing all data frames in an A-MPDU using a single ACK control frame.

[1]The IEEE 802.11n maximum header size is 36 bytes.

4.3 The Block Ack Protocol

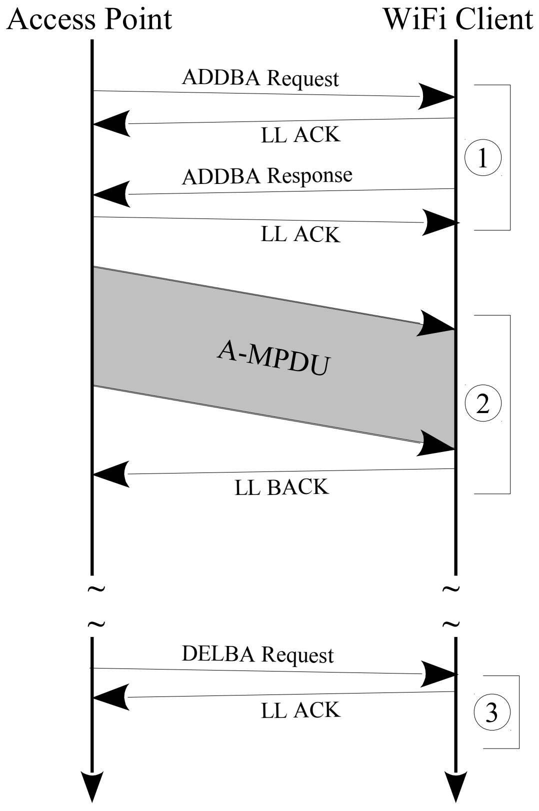


Figure 4.1: Block Ack Session

The time sequence diagram in Figure 4.1 shows the lifetime of a Block Ack protocol session. Before exchanging A-MPDUs, a Block Ack session is established ① between the communicating parties. Once the session is active A-MPDU frames can be exchanged ② between communicating parties. The Block Ack session remains active until the communication becomes idle, or it is explicitly terminated by either of the participants. The example in Figure 4.1 shows a case when the AP is the initiator of the Block Ack session, however Wi-Fi clients can initiate sessions too. The AP starts a Block Ack session by sending an ADDBA (Add Block Ack) Request action frame and the Wi-Fi client acknowledges its receipt by sending an ADDBA Response action frame.

During the ADDBA exchange, the AP and the Wi-Fi client share information such as buffer size, traffic identifier (TID), BACK frame type, BACK policy, frame density, etc. Buffer size specifies memory constraints and TID specifies traffic priority, using one of the IEEE 802.11(e) quality of service categories. The Block Ack protocol provides two types of BACK control frames, uncompressed and compressed. An uncompressed BACK frame contains 64 entries of 16 bits each to specify successfully received frames. Having a 16-bit entry for each of 64 frames significantly increases the size of the BACK (128 bytes just for the acknowledgment field), incurring overhead. Note that control frames are sent at lower and more robust rates (i.e. 12 Mbps). To overcome the space inefficiency of the normal BACK, IEEE 802.11n added the compressed BACK. The compressed BACK contains a single 16-bit field denoting the sequence number of the first frame in an A-MPDU and a 64-bit bitmap field marking the success state of up to 64 frames. The compressed BACK control frame is significantly smaller in size compared to the normal BACK control frame, making it more efficient. In our experience, we only ever saw Wi-Fi devices using compressed BACK control frames.

The Block Ack protocol also defines two BACK policies: *normal* and *delayed*. Under *normal* BACK the receiver responds with a BACK after receiving an A-MPDU while under *delayed* BACK the transmitter request a BACK by sending a BACK request. Another purpose of the BACK request is to force the receiver to advance the BACK window. This occurs in a situation where the transmitter has given up retrying frames allowing the protocol to advance. Finally, frame density defines frame spacing in an A-MPDU. This allows receivers with limited resources to successfully process A-MPDUs.

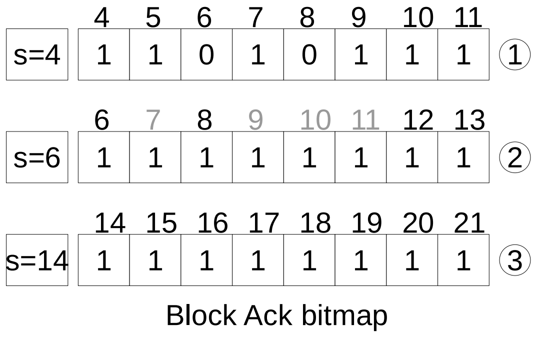


Figure 4.2: Compressed Block Ack reduces A-MPDU size in face of frame losses

The Block Ack protocol operates with a window of up to 64 unacknowledged frames. The window advances up until the first "hole" in the sequence space. On wireless channels that do not corrupt frames, the Block Ack protocol operates without problems. However, problems arise when the wireless channel corrupts frames. Frame losses cause "holes" to appear in the Block Ack window and the window only advances up to the first "hole" in the window. Subsequent A-MPDUs' sizes will be determined by the position of this first "hole" in the window. To illustrate this problem, consider the example in Figure 4.2. The starting sequence number of the first transmitted A-MPDU ① is 4 ($s = 4$) and from the received BACK bitmap the receiver indicates that it has failed to decode frames with sequence numbers 6 and 8, which must be re-transmitted. As a result, the new starting sequence number is 6 ($s = 6$), and only two new frames (sequence numbers 12 and 13) can be included in the subsequent A-MPDU ②. While the A-MPDU ① size was 8 frames, the second A-MPDU ② is reduced to four frames only: two retransmitted frames (sequence numbers 6 and 8) and two new frames (sequence number 12 and 13). Finally, the starting sequence number advances to sequence number 14 when transmitting the last A-MPDU ③, resulting in a full-sized A-MPDU.

4.3.1 Aggregated Frames

After establishing a Block Ack session the transmitter and the receiver can exchange A-MPDU frames. When transmitting A-MPDUs, multiple MPDU frames are "glued" together. The structure of an A-MPDU is shown in Figure 4.3. A-MPDU frames start with a PHY header (Preamble) which is used to detect a transmission. Each MPDU in an A-MPDU is separated by a delimiter field. The delimiter serves two main purposes: it spaces frames to conform with the negotiated frame density, and serves as a synchronising sequence when one or more delimiters are received in error. It is 32 bits in length and has four fields: reserved (4 bits); *length* (12 bits) which represents the length of the MPDU in bytes; *CRC-8* an 8-bit CRC (over the *reserved* and *length* fields); and *signature* (8 bits), a unique pattern used to detect the delimiter field when scanning for MPDU. A delimiter with *length* set to zero is valid, and is used

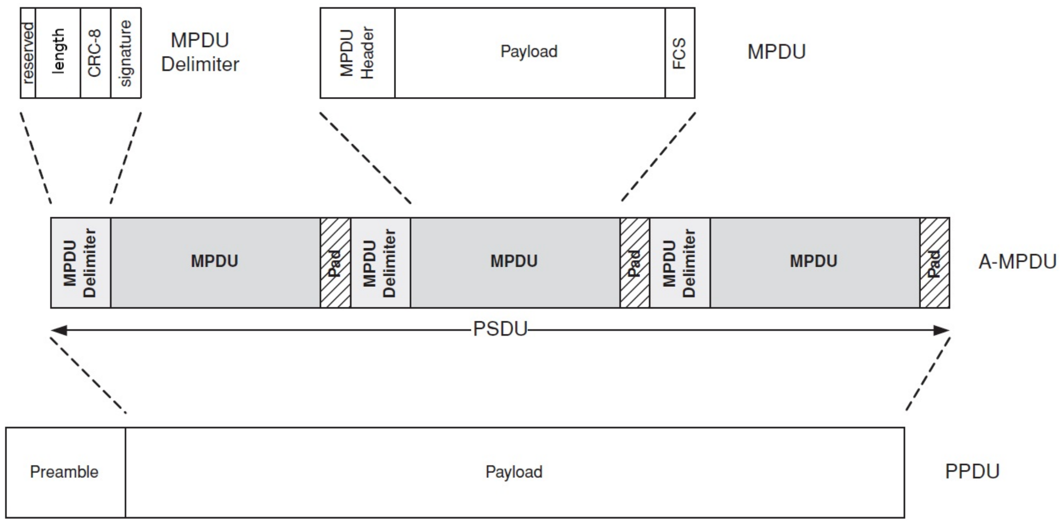


Figure 4.3: A-MPDU structure

to meet frame density constraints negotiated during Block Ack session setup. Following the *delimiter* is the padded MPDU consisting of the MAC header and the FCS. Each A-MPDU can contain many MPDUs separated by delimiter. The maximum number of MPDUs is constrained by three factors. First, the Block Ack protocol uses a window of 64 frames, so the maximum A-MPDU size to 64 frames. Second, the total size in bytes of an A-MPDU is limited to 64 kilobytes (for IEEE 802.11n). Finally, the over-the-air time of an A-MPDU is limited. According to the IEEE 802.11n standard, a station cannot transmit for more than 10 milliseconds after a single channel acquisition.

4.4 Problem Definition

How efficient are A-MPDUs in amortising MAC protocol overheads and what are the benefits of doing so? To understand the benefits of frame aggregation, we extend the model presented in Chapter 3, Equation 3.1, to include A-MPDUs.[2] In Figure 4.4 we plot theoretical goodput as a function of aggregate size (the number of frames in an A-MPDU). The model assumes that the AP forwards MTU-sized (1500 byte) TCP segments to a single Wi-Fi client. We further model TCP with delayed ACKs and a lossless wireless channel, i.e., one that

[2]Appendix A shows the derivation of the model's expansion.

delivers all the packets on the first attempt. In the case of TCP delayed ACKs, we further assume that the TCP receiver transmits TCP ACKs in response to the AP's A-MPDUs in aggregates half the length of the received A-MPDUs. In other words, if the AP sends an A-MPDU containing n TCP packets, then the client returns TCP ACKs in an aggregate of length $\frac{n}{2}$. Finally, we assume default wireless channel access parameters for the IEEE 802.11n DCF, with short guard interval (SGI) and 40 Mhz channel width. Every curve in Figure 4.4 stops at the maximum A-MPDU size, subject to the IEEE 802.11n limitations described in the previous section.

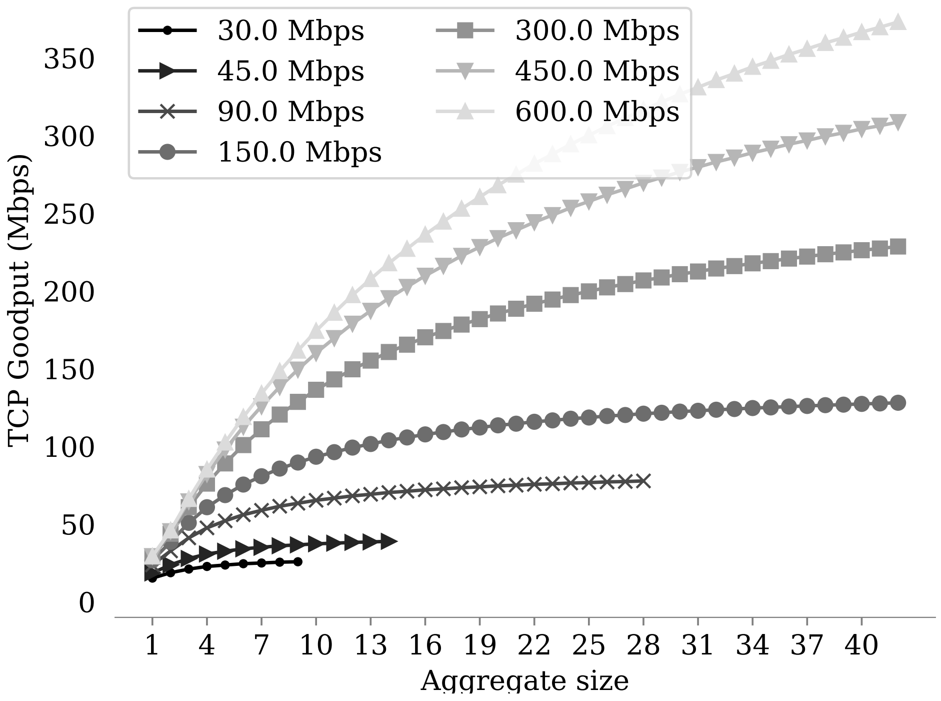


Figure 4.4: TCP theoretical goodput over IEEE 802.11n DCF when using some of the IEEE 802.11n PHY bit-rates.

The curves in Figure 4.4 are labeled with the PHY bit-rate in Mbps. They represent the various modulation and coding schemes (MCS) supported by IEEE 802.11n. For example, the curve labeled *15* is for MCS-0, which uses a BPSK modulation, a coding rate of 1/2 and a single spatial stream. At the other extreme, the curve labeled *600* is for MCS-31, the highest PHY bit-rate supported by IEEE 802.11n. It uses four spatial streams, QAM-64 modulation and a coding rate of 5/6. It is evident that using frame aggregation significantly improves the performance of IEEE 802.11n networks, especially when transmitting using high PHY bit-rates. With increasing bit-rate, the over-the-air time of a frame decreases. Because of fixed channel access overhead, the ratio between the frame time and the overhead decreases (effectively reducing utilisation). For example, transmitting single frames at 600 Mbps (MCS-31) achieves a goodput of 29 Mbps, compared to 372 Mbps goodput achieved when transmitting the largest possible A-MPDUs. Using frame aggregation gives a 12.5× improvement. 600 Mbps is the greatest possible PHY rate supported by IEEE 802.11n. In our experience it is uncommon for IEEE 802.11n Wi-Fi hardware to support that PHY rate, which requires four transmit and receive antennas. A more common configuration is Wi-Fi hardware with three or fewer antennas. For example, 150 Mbps (MCS-7) is the highest PHY bit-rate that can be achieved when using a single antenna. Transmitting single frames at the 150 Mbps PHY rate yields a goodput of 26 Mbps (17% wireless channel utilisation). Using frame aggregation achieves 128 Mbps goodput (85% utilisation), a 5× improvement. The take-away from Figure 4.4 is that frame aggregation is necessary, and should always be used when transmitting using IEEE 802.11n rates.[3] Furthermore, sending the largest possible A-MPDUs is desirable to achieve high wireless channel utilisation. But do today's commercial Wi-Fi NICs manage to send large A-MPDUs?

4.5 A-MPDU Size in Practice

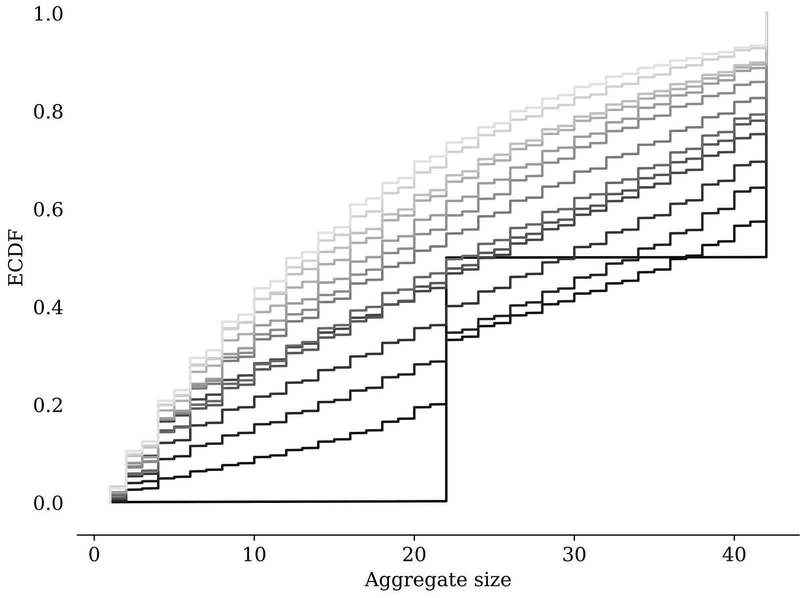
To explore A-MPDU sizes systems send in practice we set up a testbed consisting of an IEEE 802.11n Access Point (AP) and a varying number of Wi-Fi clients. Our base-case AP runs the *hostapd* [13] application under the Linux operating system. We set up a separate machine to act as a traffic generator (the sender) and connect it to the AP's distribution network using Gigabit Ethernet. The

[3]IEEE 802.11ac made it mandatory to use frame aggregation for all transmissions.

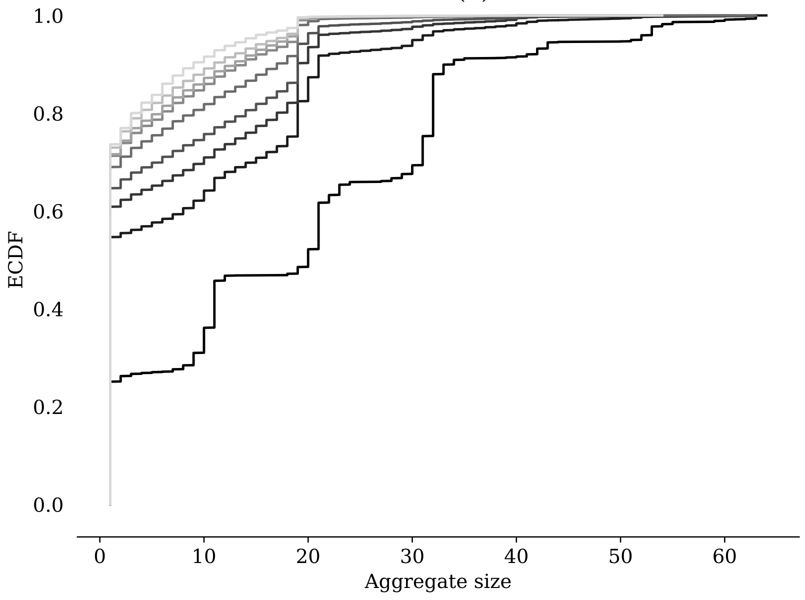
traffic generated by the sender is forwarded to Wi-Fi clients by the AP over the Wi-Fi network. First, Wi-Fi clients associate with the AP and each Wi-Fi client runs the *iperf* [14] application in server mode. After Wi-Fi clients successfully associate with the AP, the sender establishes a TCP connection with each Wi-Fi client using *iperf* application and sends to each Wi-Fi client as fast as the sending socket allows for 120 seconds. The Wi-Fi stations in these experiments transmit using a fixed PHY bit-rate (MCS-6). We instrumented the device driver to record the size of each A-MPDU transmitted. A full description of the testbed and the experimental setup is provided later in the chapter. The distribution of the A-MPDU sizes when the AP transmits to a varying number of Wi-Fi clients is shown in Figure 4.5a. The first observation is that as the number of Wi-Fi clients increases the AP's A-MPDU size decreases. For example, when sending to one client the A-MPDU size alternates between 42 (the maximum A-MPDU size) and 22. This alternation occurs as a result of the interaction between the device driver algorithm used to transmit A-MPDUs and the Block Ack protocol. We explain this behaviour in more detail later in this chapter.

When sending to two Wi-Fi clients, as compared to one Wi-Fi client, the median A-MPDU size has increased slightly, from 32 to 37, however the fraction of time the AP sends maximum-sized A-MPDUs has fallen by 2%. In addition, the fraction of time A-MPDU size is below 22 has increased by 20%. This trend of the A-MPDU size dropping as the AP transmits to a grager number of Wi-Fi clients continues. When the AP transmits to fourteen Wi-Fi clients only 7% of A-MPDUs are maximum size, with a median A-MPDU size of 13 frames. Figure 4.5b shows the distribution of A-MPDU sizes Wi-Fi clients send when transmitting to the AP. With the exception of the single Wi-Fi client case, more then 50% of the time Wi-Fi clients transmit single frames. Given these results in Figures 4.5a and 4.5b what is the impact on achieved goodput?

Figure 4.10a shows the TCP goodput achieved when the AP transmits to between one and fourteen clients. As expected, the goodput decreases as the



(a) Downlink



(b) Uplink

Figure 4.5: A-MPDU size empirical CDF when the AP transmits to increasing number of Wi-Fi clients (Figure 4.5a) and when Wi-Fi clients transmit TCP ACKs back to the AP (Figure 4.5b). The solid black line is one client and lightest grey line is 14 clients.

number of clients the AP transmits to increases. With a single Wi-Fi client, the achieved goodput is 103 Mbps, however it drops to 81 Mbps when the AP transmits to fourteen clients. Going from one client to fourteen clients results in a 27% goodput loss. Our findings suggest that the A-MPDU sizes sent by

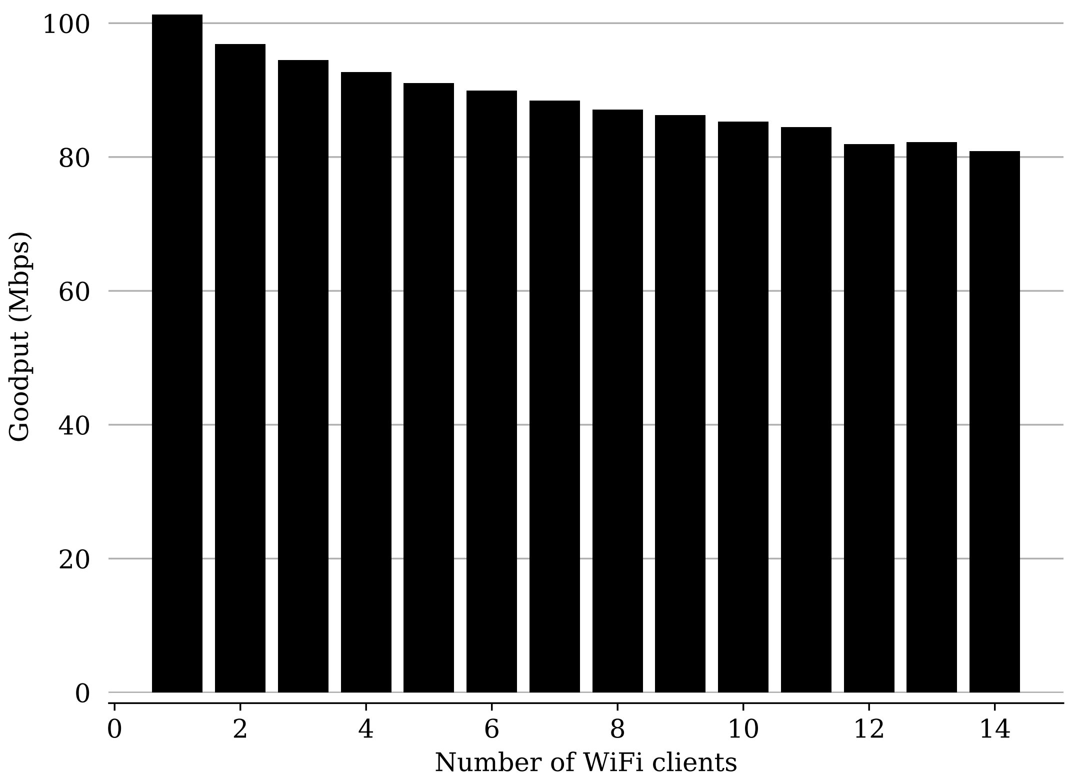


Figure 4.6: TCP Downlink Goodput

the AP and Wi-Fi clients are small enough to negatively effect the achieved goodput.

We use these results as a motivation to answer the following questions:

1. Why is the majority of A-MPDUs sent small?
2. Why are more then half of TCP ACK transmissions single-frame transmissions?
3. TCP goodput decreases as the number of Wi-Fi clients increases. Is the A-MPDU size responsible for this drop?

To answer these questions one must take account of the full stack of protocols, including the Wi-Fi device driver. In particular, how and where packets are buffered and queued as they traverse the stack is important. Queues are an important component of any networked system as they glue together rate-mismatched input and output processes. For this work we choose Linux, as its open-source nature enables us to dive into the stack's details.

4.6 Packet Buffering in Linux

Figure 4.7 shows a general overview of the Linux network stack's packet transmission architecture. Above the stack are application programs that use some transport protocol, such as TCP, to transmit data. TCP accepts the application's bytes and segments them into packets, and after performing TCP-related processing, it passes each packet down the stack as a *struct sk_buff*. After IP decides how to forward the packet, it checks if the outgoing network interface can accept the packet (i.e., it can transmit the frame immediately) it is passed to the device driver. Otherwise the packet is queued in the IP queue and stays there until the outgoing interface can accept new packets.

How packets are queued, dropped, and dequeued is determined by the queueing discipline (qdisc) subsystem. The modular architecture of the qdisc subsystem allows for various queueing policies to manage the IP queues. By default, the FIFO qdisc is installed on a Wi-Fi device[4]. FIFO is the simplest queueing discipline: it enqueues incoming packets at the tail of the queue and dequeues them from the head of the queue. When the queue becomes full[5], no more packets can be admitted into the queue, and new packets are dropped. Once the device driver can accept new packets for transmission, the IP queue starts dequeuing packets at its head and passes them to the device driver.

On Linux, several Wi-Fi card device drivers (including that for the Atheros 9000 series) use the generic *mac80211* layer. *mac80211* handles standard IEEE 802.11 functionality, such as adding link-layer headers, maintaining stations' association information, and handling various management frames.

The *mac80211* layer does not include active queue management except for temporary buffers used to perform tasks such as fragmentation. A detailed description of the *mac80211* layer is out of scope for this thesis. In summary, after adding IEEE 802.11 headers and performing validation tasks such as making sure that the station is associated, *mac80211* hands the packet to the device driver (ath9k).

[4]Four FIFO queues are installed, one for each IEEE 802.11e access category.
[5]By default the queue size is set to 1000 packets.

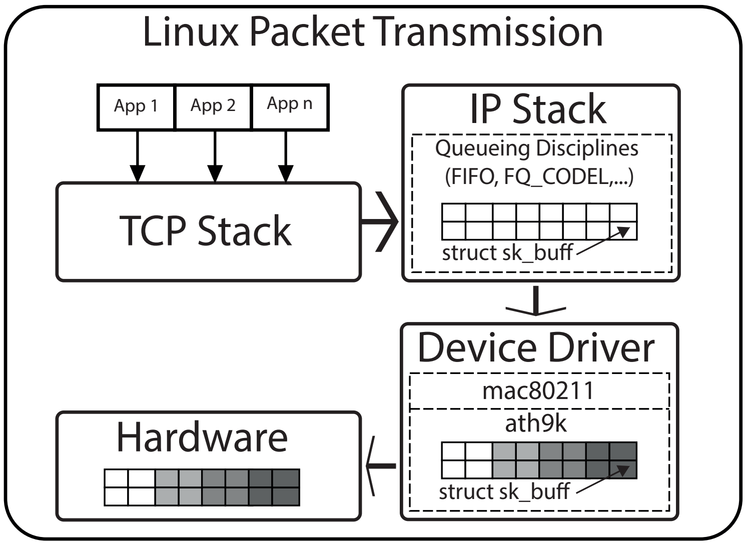


Figure 4.7: Linux Network Stack

The device driver is the lowest software layer before the packet is handed to the hardware device for transmission. Our experimental testbed runs on Atheros hardware driven by the *ath9k* [12] open-source driver. Frame aggregation is performed in the device driver, so we provide a description of how A-MPDUs are created. The driver keeps a queue for each of the established Block Ack sessions. Each queue is identified by a unique traffic identification (TID) and access category (AC). Upon receiving a packet from the *mac80211* layer, the packet's corresponding TID is found and the packet is queued on that TID's queue. TID queues are categorised based on access category defined by IEEE 802.11e quality of service (QoS). The device driver schedules TID queues for packet transmission in round-robin fashion. First TID queues from the highest AC are scheduled, followed by those on lower ACs. When the device driver picks a TID for packet transmission it dequeues packets one at a time and adds them to the A-MPDU. The driver ensures that the number of packets in an A-MPDU does not exceed the IEEE 802.11n constraints. To help create large A-MPDUs the *ath9k* driver manages a double buffer of DMA descriptors. DMA descriptors are regions of host memory that can be directly accessed by the hardware. Packets in the DMA descriptors are ready for transmission by the hardware. Double buffering makes sure the hardware does not run dry and gives an opportunity for packets to be buffered in TID queues while there are DMA descriptors filled with packets waiting to be transmitted.

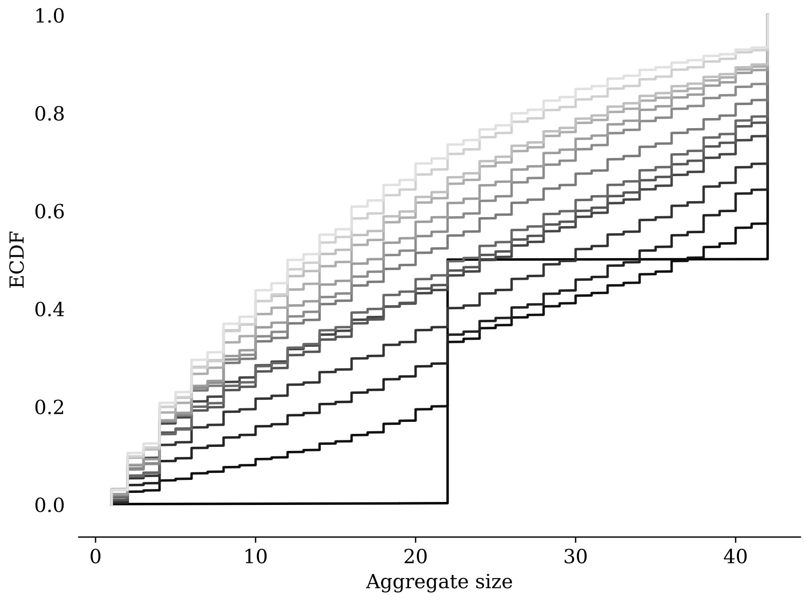
While the combination of FIFO queues and TID scheduling in the *ath9k* driver works well when the AP serves only one Wi-Fi client, it gradually starts sending smaller aggregates as the number of Wi-Fi clients increases. The reason for this is twofold. First, the FIFO discipline manages only one queue, and packets are dequeued on a first-in-first-out basis regardless of which Wi-Fi client they are destined for, resulting in random dequeueing of Wi-Fi clients' packets. The second problem comes from the decoupling of the driver's TID scheduling and dequeuing of packets from the IP queue. This approach results in TIDs being scheduled prematurely or the IP queue's dequeuing more packets than it should. When a TID is scheduled prematurely, a chance is missed to create larger A-MPDUs, as there could be more packets for the given TID waiting in the IP queue. On the other hand, as was mentioned previously, A-MPDU size is limited by the length in bytes, maximum number of packets, and maximum transmission time. Handing more packets to the driver than can fit into an A-MPDU could potentially lead to sub-optimal A-MPDU size.

Armed with an understanding of how Linux buffers packets, we can turn to the questions we previously posed, but before we do so we offer a brief overview of another queueing discipline, namely Controlled Delay (CoDeL) [68]. CoDeL is an active queue management algorithm developed in response to oversized queues observed on the Internet, a problem known as "bufferbloat" [43]. CoDeL's aim is to achieve a parameterless active queue management algorithm that keeps delay low while permitting bursts of traffic, and adapting dynamically to link changes without having an adverse impact on utilisation. It achieves these goals by using novel approaches such as using local minimum queue delay as measure of standing/persistent queues and measuring packet-sojourn time instead of queue size in bytes or packets. As long as the local queue delay is below a **target** value (5 milliseconds, by default), packets are not dropped. If the minimum queue delay has exceeded **target** for a time greater than **interval**

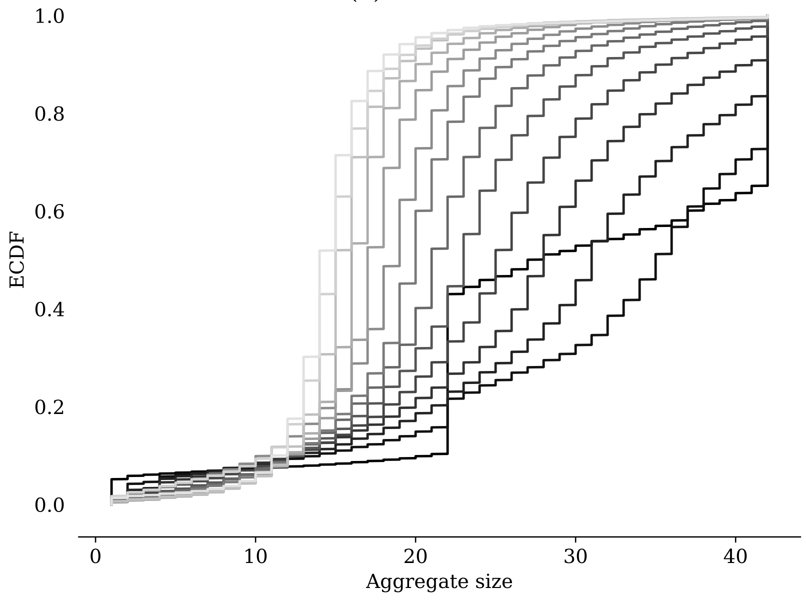
(100 milliseconds, by default) CoDeL enters dropping mode. While in dropping mode, packet drops are governed by a control mechanism ensuring that packet drops cause a linear drop in throughput. Packets are dropped from the head of queue. Once the minimum queue delay falls below **target** CoDeL exits dropping mode.

An interesting variant of CoDeL is the Fair Queuing CoDel (FQ CODEL) queueing algorithm. FQ CODEL is a combination of deficit fair queueing [80] and the CoDeL algorithm. Each packet is classified by a flow and each flow has a queue managed by an instance of the CoDeL algorithm. Fair sharing of resources among queues is achieved using round-robin scheduling. By default each flow is given a deficit of MTU (Maximum Transfer Unit) bytes and packets are dequeued from the same queue until that flow's deficit is used. Using a queue per flow could potentially lead to large A-MPDUs, and we wanted to see if that was the case. We repeat the same experiments we ran using FIFO, however this time we run the AP with FQ CODEL instead.

Figures 4.8b (downlink) and 4.9b (uplink) show the distribution of A-MPDU sizes when the AP runs FQ CODEL. Compared to A-MPDU size when the AP is running FIFO (Figure 4.8a and Figure 4.9a), FQ CODEL sends smaller A-MPDUs. This is contrary to what we'd expect from having per-flow queues. However, FQ CODEL uses a deficit value of one MTU size, translating to one packet per flow. This leads to fewer packets being queued in TID queues, resulting in small A-MPDUs. In addition, by aiming to keep delay low, FQ CODEL incurs more packet drops, which reduces the number of packets in queues, leading in turn to smaller A-MPDU sizes. To see how these A-MPDU sizes affect the goodput in Figure 4.10 we show TCP goodput for FQ CODEL (Figure 4.10b) and FIFO (Figure 4.10a), for comparison. As the number of Wi-Fi clients increases, FIFO achieves *better* goodput compared to FQ CODEL. For example, when the AP uses FQ CODEL to transmit to nine Wi-Fi clients, it achieves goodput similar to that achieved when the AP uses FIFO to transmit to fourteen clients.



(a) FIFO

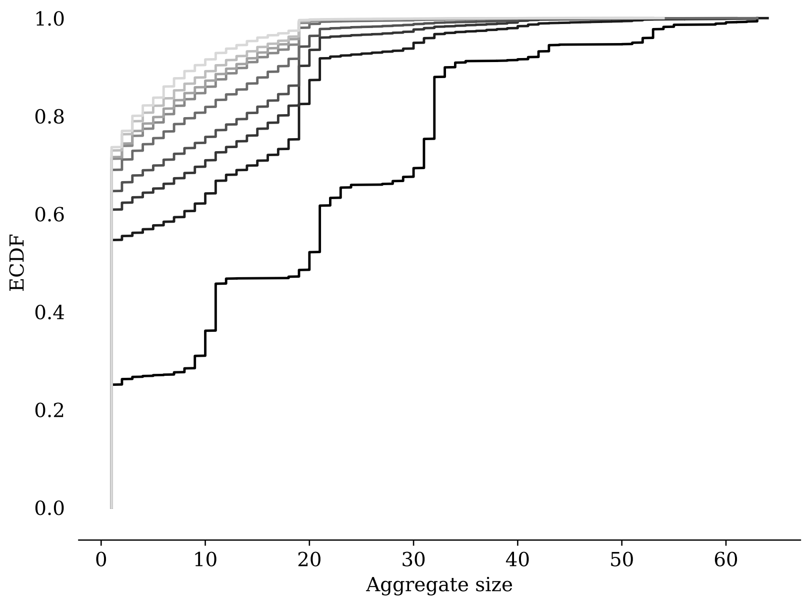


(b) FQ CODEL

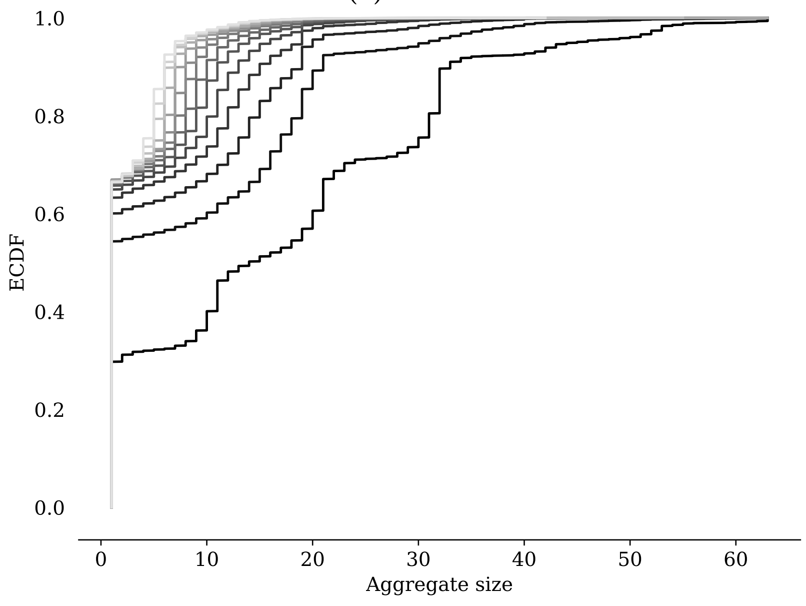
Figure 4.8: Downlink A-MPDU size empirical CDF when the AP is running FIFO and FQ CODEL. The solid black line is one client and the lightest grey line is 14 clients.

4.6.0.1 Why does *hostapd* send small A-MPDUs when running FIFO and FQ CODEL?

As we described earlier IEEE 802.11n constrains the number of packets that fit in an A-MPDU. An A-MPDU cannot be larger then 64 KB in size and sending MTU-sized packets thus results in maximum of 42 packets in an A-MPDU.



(a) FIFO

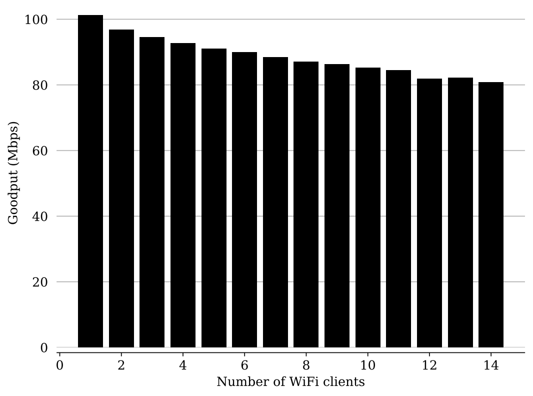


(b) FQ CODEL

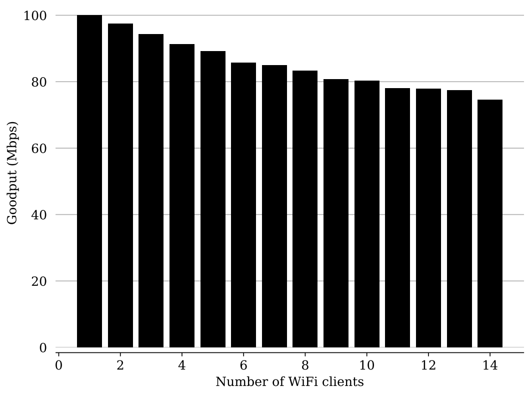
Figure 4.9: Uplink A-MPDU size empirical CDF when the AP is running FIFO and FQ CODEL. The solid black line is one client and lightest grey line is 14 clients.

The other limitation comes from the window size of the Block Ack protocol, which is 64 packets. This is why the maximum A-MPDU size on downlink is 42 packets and 64 on uplink[6]. Two factors cause the A-MPDU size to be less then the maximum allowed. First, there are not enough packets buffered

[6]TCP ACKs are small in size and are not limited by the maximum size in bytes, but they are limited by the Block Ack protocol window.



(a) FIFO



(b) FQ CODEL

Figure 4.10: TCP goodput when the AP is transmitting to varying number of Wi-Fi clients.

in the TID queue, and second, the Block ACK protocol window is limiting. Packet losses can limit the Block ACK window as explained in Section 4.3. To understand what causes A-MPDU size to be smaller than the maximum size, we instrumented the device driver to log why an A-MPDU resulted in a particular size. We log A-MPDU size, the number of packets in the TID queue left after the A-MPDU was formed, and whether the Block ACK window was closed. We classify A-MPDUs into four categories: *baw*, *good*, *tidqe*, and *single*

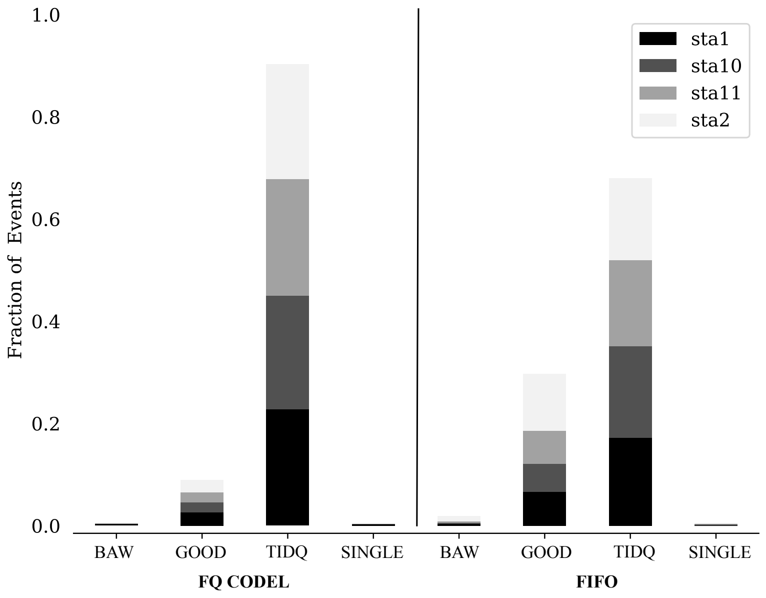


Figure 4.11: FIFO and FQ CODEL fail to send large A-MPDUs because there are not enough packets in the queue

The *baw* category indicates the Block Ack protocol window was closed.

The *good* category indicates an A-MPDU was of maximum size. The *tidqe* category indicates that the TID queue ran dry while forming an A-MPDU. And the *single* category indicates single packet transmissions. For this experiment we choose a configuration where the AP sends to four Wi-Fi clients, as the Block Ack window will not be limited by the device driver's double buffering in this scenario. The histogram in Figure 4.11 shows the fraction of time each category of A-MPDU occurs when the AP transmits to Wi-Fi clients using FIFO and FQ CODEL. An empty TID queue is the main reason why A-MPDU sizes are small. More than 60% of cases when the AP uses FIFO and more than 85% on the AP when it uses FQ CODEL are in this category. Categories *baw* and *single* occur infrequently, and so have negligible effect on A-MPDU sizes. In summary, FIFO randomises the order of clients for which packets are dequeued, resulting in TID queues being empty, while FQ CODEL keeps queues small, resulting in empty TID queues.

4.6.0.2 Why are the majority of uplink transmissions single-packet transmissions?

Surprisingly, the uplink results in Figures 4.9a and 4.9b show that the majority of TCP ACK transmissions are single-packet transmissions. Understanding this phenomenon requires an understanding of how A-MPDUs are processed at the receiving Wi-Fi client. A-MPDU frames received by the hardware are de-aggregated before being handed to the device driver upon firing of a receive interrupt. Upon handling the receive interrupt, the device driver processes packets one by one and passes them up the stack. As packets propagate up the stack they elicit TCP ACKs once the TCP stack has processed them[7]. These TCP ACKs are passed down the stack for transmission (following the same transmission path described in Section 4.6) while further incoming packets are still being processed. The processing of the TCP data and the transmission of TCP ACKs occur in parallel. TCP ACK transmission may be spread out

[7]With delayed TCP ACK enabled, every second TCP data packet received will elicit one TCP ACK.

in time, depending on the processing time for the incoming packets, and they may be scheduled prematurely before all are queued by the device driver in the hardware. This premature transmit scheduling of TCP ACKs results in the driver's not being able to form large A-MPDUs, as simply not enough TCP ACKs are in the TID queue, since they are still being generated.

Our hypothesis is that we can send large A-MPDUs by exchanging information across layers to aid the device driver in scheduling and forming large A-MPDUs. The next section describes AAQ, a queueing discipline designed achieve these aims. In addition, we present the design of TAO, a simple optimisation that arranges aggregation of TCP ACKs into large A-MPDUs by delaying TCP ACK transmission until all TCP ACKs for a received A-MPDU have been generated.

4.7 Aggregation Aware Queueing (AAQ) and TCP ACK Optimisation (TAO)

This section presents AAQ and TAO, techniques designed to send large A-MPDUs and improve goodput for TCP flows that traverse Wi-Fi access points. AAQ is designed to send maximum-sized A-MPDUs when Wi-Fi AP forwards packets to Wi-Fi clients. And TAO is an optimisation that lets Wi-Fi clients send optimal-sized A-MPDUs when sending TCP ACKs back to the AP for forwarding to the remote sending host. TAO does not require AAQ and can operate independently of the queueing discipline. Based on our observations of the stock Linux stack's behaviour, as described in the previous section, we set the following goals for AAQ and TAO:

1. Maximise A-MPDU size on the downlink.

2. Maximise A-MPDU size on the uplink.

3. Share resources fairly between Wi-Fi clients.

4.7.1 Aggregation Aware Queueing

AAQ is an IP queueing discipline designed to increase A-MPDU size. To achieve this aim, AAQ exchanges information with the low-level device driver that forms A-MPDUs. AAQ aims to provide fair sharing amongst Wi-Fi clients and aid the device driver with scheduling decisions. To achieve fair sharing, AAQ uses round-robin fair queueing, as does FQ CoDel. Unlike FQ CoDel, AAQ uses per-destination queues instead of per-flow queues. Given that Block Ack sessions are per-Wi-Fi destination, per-flow granularity does not provide further benefit. AAQ creates 256 queues and determines on which queue to enqueue a packet based on the last octet of the destination's MAC address. When two or more Wi-Fi stations have the same last octet, a collision occurs. To handle collisions we use a linked list of queues resolved using the destination MAC address. Like FQ CoDel, AAQ enforces a total queue length limit across all queues, and packets are dropped from the flow with the greatest number of packets enqueued when this limit is exceeded. When dropping packets, AAQ uses a drop-head policy.

When dequeueing, AAQ picks queues in round-robin fashion. AAQ relies on information from the device driver to decide when it should move to the next queue. When dequeuing packets, AAQ keeps dequeuing from the same queue until the device driver informs it to stop doing so. The device driver does this by passing a message up the stack. Two different messages can be sent by the device driver to AAQ. The first message is a *PAUSE* message, which tells AAQ that it should stop dequeuing packets from the given queue and move on to the next queue. The second message is a *RESUME* message, which informs AAQ that it can dequeue packets for the given destination. The driver sends the *PAUSE* message when the BlockAck window does not allow further transmissions or the maximum allowed number of bytes/packets in an aggregate has been reached. The driver sends the *RESUME* message when space for more packets in the driver queue becomes available—that is, when packets have been transmitted and the BlockAck window has opened. Algorithm 1 shows the pseudocode for the *enqueue* operation. It enqueues a packet into the corresponding queue, and checks if the global queue limit has been reached. In cases when the queue overflows, it finds the longest queue and drops the packet at its head.

Algorithm 1 enqueue(*packet*)

```
1: $idx \leftarrow packet.dstmac[5]$
2: $q \leftarrow get\_queue\_for(idx)$
3: $enqueue\_packet(q, packet)$
4: $qlen{+}{+}$
5: if $qlen \geq global\_qlen$ then
6:    drop packet from the head of the largest queue
7: end if
8: if $q.is\_new$ then
9:    add_to_list($active\_queue\_list$, $q$)
10:   $q.is\_new \leftarrow FALSE$
11: end if
```

The dequeue operation, whose pseudocode is given in Algorithm 2, dequeues packets from one of the queues. To ensure that packets are dequeued successively from the same queue, the current queue is stored in the shared variable *current_queue*.

Algorithm 2 dequeue

```
1: if not $current\_queue$ then
2:    if not empty(active_queue_list) then
3:       $current\_queue \leftarrow get\_next\_queue(active\_queue\_list)$
4:    else
5:       return $null$
6:    end if
7: end if
8: $packet \leftarrow get\_packet(current\_queue)$
9: if not $packet$ then
10:   $remove\_queue(active\_queue\_list, current\_queue)$
11:   $curren\_tqueue \leftarrow NULL$
12:   go to 1
13: end if
14: return $packet$
```

Finally, Algorithm 3 shows pseudocode for a callback function registered with the device driver. This function is called by the device driver to pass messages to AAQ about the queue's state. Depending on the message received, it either moves the queue to the list of active queues (for the *RESUME* message) or it moves the queue to the list of paused queues (for the *PAUSE* message). When the device driver invokes the callback method it passes a unique *id* as a parameter. This *id* is used to map the device driver's queue (TID queue) to the corresponding AAQ queue. In the beginning *id* is set to zero for all AAQ queues; *id* is updated when the driver passes it to AAQ.

Algorithm 3 driver_callback(*queue_id*, *message*)

```
1:  if $message == PAUSE$ then
2:      if not $current\_queue.id$ then
3:          $current\_queue.id \leftarrow queue\_id$
4:          $move\_queue(paused\_queue\_list, current\_queue)$
5:          $current\_queue \leftarrow null$
6:      end if
7:  else if $message == RESUME$ then
8:      $q \leftarrow get\_queue(paused\_queue\_list,\ queue\_id)$
9:      if $q$ then
10:         move_to_tail($active\_queue\_list$, $q$)
11:     end if
12: end if
```

4.7.2 TAO

Today's TCP design is blissfully ignorant of Wi-Fi's link-layer aggregation of frames. When an A-MPDU arrives at a TCP receiver, the host CPU interleaves processing of TCP packets from the received A-MPDU with generation of TCP ACKs for those received TCP packets. The result is that even when an A-MPDU consisting of multiple tens of TCP packets has been received, the receiver's Wi-Fi card is presented with small bursts of just a few TCP ACKs, which the Wi-Fi card sends as short A-MPDUs, unaware that more TCP ACKs for TCP packets already received will arrive imminently. We rectify this lack of coordination across layers by introducing the TCP ACK Optimisation (TAO), which counts the TCP packets received and forces the Wi-Fi device driver to delay their formation into A-MPDU and transmission until the driver possesses TCP ACKs for *all* received TCP packets. TAO, too, requires passing of state across layers of the stack, and new control mechanisms that inspect and update this state.

To summarise, AAQ is a fair queueing discipline designed to share state with the low-level device driver to aid in forming large A-MPDUs. It uses a per-destination queue with round-robin dequeuing and a drop-head policy. It is designed to be used by Wi-Fi APs sending data to Wi-Fi clients. TAO is a receiver-side optimisation that allows Wi-Fi clients to form large A-MPDUs when sending TCP ACKs.

4.8 Implementation and Evaluation

We now describe the implementation of AAQ and TAO. In addition we present experimental results from our testbed that evaluate AAQ and TAO 's efficiency.

4.8.1 AAQ

AAQ is implemented as a new queueing discipline (qdisc) module for Linux kernel version 3.18.7. The core AAQ functionality is implemented in 750 lines of code. In addition, support for AAQ is added to the *tc*[8] tool used to control and manipulate Linux's queueing subsystem. AAQ is registered under the name **fq_aaq**. We install **fq_aaq** using tc as follows:

```
# /sbin/tc qdisc add dev wlan0 root fq_aaq limit 1000
```

The above command installs AAQ and sets the global limit to 1000 packets on Wi-Fi device *wlan0*. To support callback functionality from the device driver to AAQ we modified *struct Qdisc_ops*[9] to add a new functifn pointer, *pause*. When AAQ is initialised it registers the *pause* callback function. Calling the function from the low-level device driver works in a similar fashion to the existing method to stop the driver queue. When the driver queue becomes full it calls *ieee80211_stop_queue* which in turn propagates the call up the stack. We added calls to *ieee80211_pause_queue* to propagate the *PAUSE* and *RESUME* messages up the stack. AAQ sets the pre-existing *more_data* field of *struct*

[8]Traffic Control is part of the iproute2 tools.

[9]This structure is used when registering a new queuing discipline. It contains various function pointers, such as for *enqueue* and *dequeue* functions.

sk_buff to inform the device driver if there are more data queued for the given queue.

ath9k Changes To support AAQ we modify the *ath9k* device driver to make use of the extra information provided by AAQ. *struct_ath_atx_tid* represents a TID session. We added the following new members to this structure:

- *bool_dont_sched_yet*, a flag to hint to the TID scheduler that AAQ has more packets to dequeue so it can postpone the scheduling of the given TID.
- *unsigned long_id*, the unique identifier for the given TID, and the *id* that will be passed up to AAQ to map AAQ queues to TID queues. It is set to the *jiffies*[10] value at the time of a Block Ack session's creation.
- *int bytes_in_tid*, keeps track of the number of bytes currently in TID.
- *int bytes_allowed*, holds the maximum number of bytes that the current TID can hold. When *int bytes_in_tid* reaches this value, the device driver passes a *PAUSE* message to AAQ.

In addition to these *struct ath_atx_tid* modifications, two of the *ath9k* driver's functions have been modified:

- *ath_tx_start* is responsible for deciding a packet's TID queue and invokes the TID scheduling routine. We modify it to incorporate the decision on when AAQ should stop dequeuing packets for the given queue. If the Block Ack window is closed, or *bytes_in_tid* has reached *bytes_allowed* we invoke *ieee80211_pause_queue* with *id* and the *PAUSE* message. In addition, based on the state of the *more_data* flag, we set *dont_schedule_yet*.
- *ath_txq_schedule* was modified to postpone scheduling of TIDs that have *dont_schedule_yet* set.

[10]*jiffies* is a global variable in the Linux kernel holding the number of ticks since the system was booted.

4.8.2 TAO

Recall that TAO keeps a counter of the number of TCP packets received. Because the MAC layer can retransmit packets, TAO must not count duplicate TCP data packets. As duplicate suppression is handled in the *mac80211* layer, we added a callback to TAO from the *mac80211* layer for each non-duplicate packet. We register the callback function, *ath9k_rx_success*, when the *ath9k* driver is loaded (if TAO is enabled). First, TAO looks for the TID the received packet belongs to. Next, if the packet is a TCP data packet, the counter *data_to_ack* for the given TID is incremented. If the packet is the last packet in the A-MPDU we set the flag *more_data* for the TID to false. On the transmit side, TAO examines each outbound packet and determines if it contains a TCP ACK. If that is the case, TAO adds the packet to a *TCP ACK queue* unique to the given TID. We modify the TCP stack to pass the exact number of TCP segments the TCP ACK acknowledges to the lower layer in a new field *ack_cnt* in *struct skb_buff*. For each outbound TCP ACK packet, *data_to_ack* is decremented by *ack_cnt*. When *data_to_ack* reaches one and *more_data* is false, TCP ACKs queued will be released for transmission.

When TCP delayed ACK is used, there are cases where an inbound A-MPDU contains an odd number of TCP packets. In such cases, the last packet in the A-MPDU will only cause an ACK to be generated when the delayed ACK timer fires 200 ms later. We don't wait for the delayed ACK timer to fire before releasing the TCP ACKs, as we don't want to further delay them.

Before we discuss the evaluation of AAQ and TAO we outline two bugs we found in the *ath9k* driver while validating our testbed. First, TCP occasionally retransmitted TCP segments. These retransmissions were not the result of packet drops at the IP queue, nor were they caused by the RTO (TCP retransmit timeout) firing, nor by drops in the MAC protocol. Investigating further, we noticed that the Wi-Fi hardware occasionally prematurely fired the receive interrupt before the receive descriptors were properly filled. The driver treated the given descriptor as invalid and never passed the descriptor up the stack, causing TCP to retransmit. We fixed this problem by not dropping the descriptor, and instead delaying its processing. The hardware later properly filled in the descriptors.[11]

The second problem was in the TID scheduling function. The scheduler keeps a list of active TIDs that need to be scheduled. In cases when there are already two A-MPDUs in memory reserved for DMA to the Wi-Fi card, and the scheduler tries to schedule a third TID, that TID will not be scheduled. When this condition occurs the third TID is replaced on the list of active TIDs. In the original driver this TID was added to the tail of the active TID list. This behaviour has the potential to be unfair when the AP is serving multiple Wi-Fi clients. To fix this problem, we re-insert the given TID at the head of the queue instead of the tail. This way, the TID will be the first to be scheduled when the AP has finished transmitting the previously queued A-MPDUs.

4.8.3 Testbed

We evaluate AAQ and TAO using a testbed (Figure 4.12) consisting of fourteen Wi-Fi clients equipped with a combination of Intel i7 3.2 Ghz desktop PCs and Intel NUC mini-PCs with 1.8 Ghz i5 CPUs and 8 GB of RAM. All Wi-Fi stations in our testbed run Ubuntu 14.04.5 LTS [22] and our modified Linux Kernel 3.18.7 [16]. The first AP runs on a Intel NUC using *hostapd v2.3* [13] and the second AP is a commercial Cisco Linksys WRT610Nv1 [15]. A passive monitoring Wi-Fi client captures and stores traffic during experiments using *tcpdump* [17]. The traffic generator is a desktop PC with an Intel i7 3.2 Ghz CPU and 24 GB of RAM and is connected to the APs using 1 Gbps Ethernet. We ensure that the traffic generator is not the bottleneck by increasing socket buffer sizes. In addition, TCP small queues [21] is disabled. TCP is configured to run the *Reno* congestion control algorithm [19], with selective acknowledgments (SACK) [20], window scaling [18], and timestamp options [18] enabled. *iperf* [14] is used to generate traffic. The default IP queue length is set to 1000 packets. We use Wi-Fi hardware with Atheros chipsets (9300 and AR5418) running

[11]This issue was reported on the ath9k mailing list.

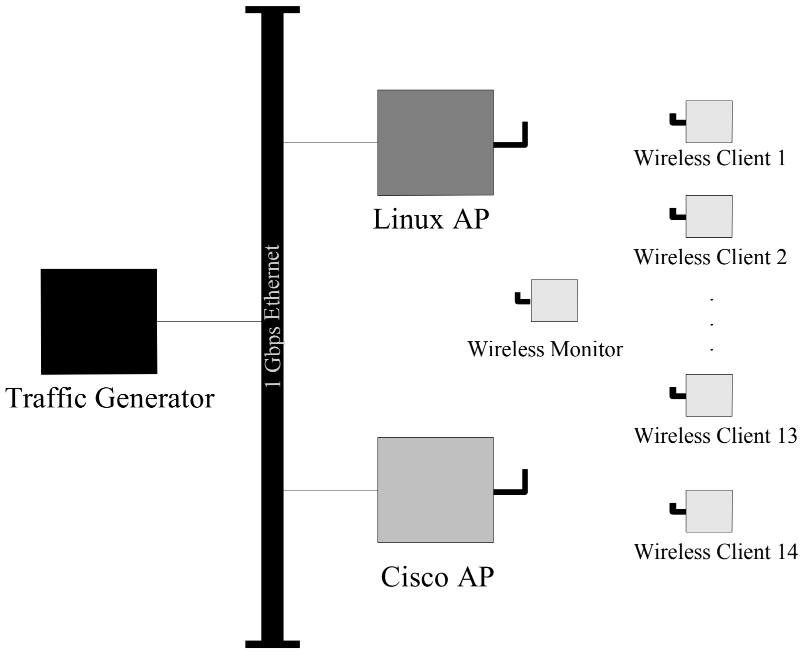


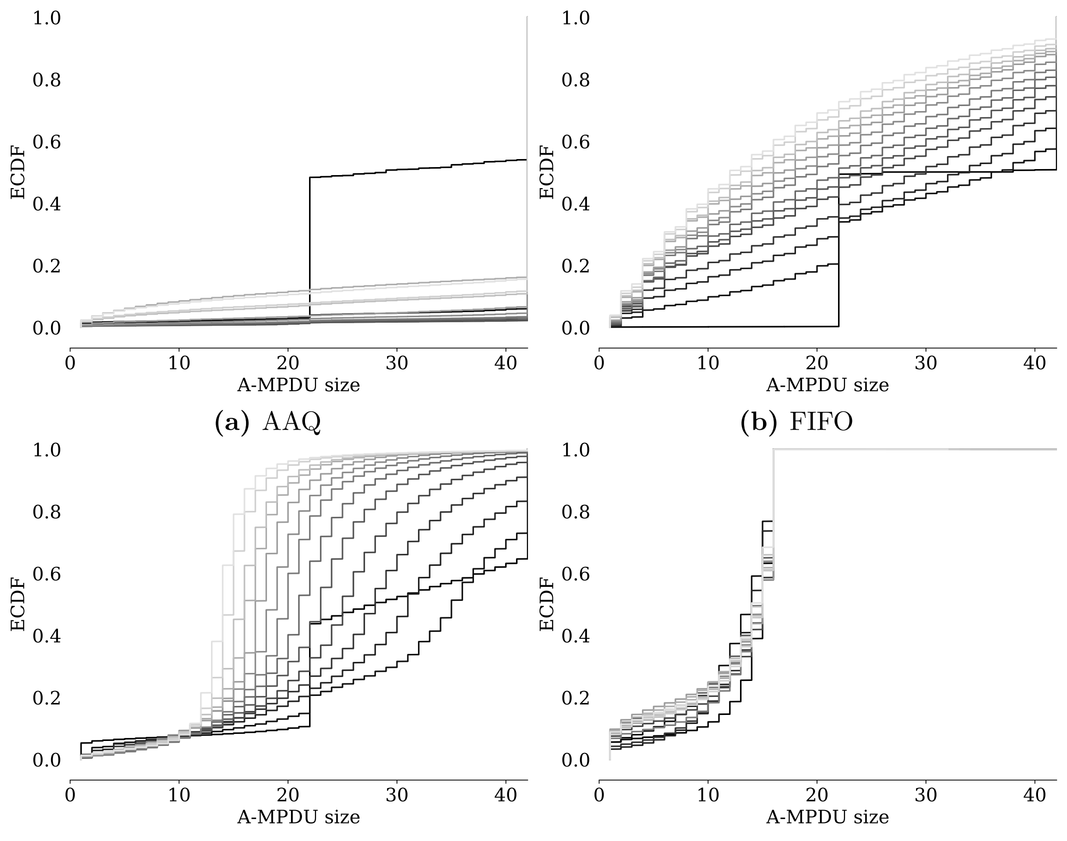
Figure 4.12: Experimental testbed.

the *ath9k* device driver. Wi-Fi stations are configured to use the 5 Ghz band (Channel 157) and 40 Mhz channel bandwidth with the short-guard-interval (SGI) and RTS/CTS enabled. Both APs and Wi-Fi clients are configured to use a fixed PHY rate of 135 Mbps (MCS-6).

4.8.3.1 AAQ Evaluation

We evaluate AAQ using the testbed described in the previous section. The aim of the evaluation is to see if AAQ manages to send large A-MPDUs. In addition to comparing AAQ to FIFO and FQ CODEL, we also compare it to a commercially available Linksys AP. We are interested to see how the A-MPDU size changes as the number of Wi-Fi clients the AP serves increases. We start with the AP serving only one Wi-Fi client and increment the number of Wi-Fi clients up to fourteen. The traffic generator's *iperf* client is instructed to establish a TCP connection with each Wi-Fi client's *iperf* server. Each experiment lasts for 120 seconds and is repeated five times.

Figure 4.13 shows the empirical distribution of A-MPDU sizes when APs transmit to Wi-Fi clients. Results combine measurements from five experiments for AAQ, FIFO, FQ CODEL, and Linksys. The solid black curve shows the



(a) AAQ **(b)** FIFO **(c)** FQ CODEL **(d)** LINKSYS

Figure 4.13: Downlink A-MPDU size empirical CDF when the AP is sending from one to fourteen Wi-Fi clients. The solid black line is one client and lightest grey line is 14 clients.

case when the AP transmits to one Wi-Fi client and the lighter coloured curves show increasing numbers of Wi-Fi clients. The case with fourteen clients is the lightest coloured curve. When the AP sends to one Wi-Fi client, we observe the effect (Figure 4.13a and Figure 4.13b) of double buffering in the *ath9k* driver. Because there is only one Wi-Fi client available to which to send packets, the A-MPDU size alternates between 42 packets and 22 packets. At first the AP aggregates 42 packets and initiates a DMA transfer. When forming the next A-MPDU, while the previous A-MPDU is still waiting for transmission, the next A-MPDU can only be as large as 22 packets, because the BlockAck window closes. Once the first A-MPDU of 42 packets has been transmitted and acknowledged by the receiver, space becomes available in the window and another 42 packets can be aggregated. This process repeats in alternation leading to the results in Figure 4.13a and Figure 4.13b. This effect is less visible when the AP runs FQ CODEL (Figure 4.13c) because FQ CODEL keeps IP queues small, resulting in more empty TID queues when forming A-MPDUs. In cases when the AP sends to more then one Wi-Fi client, Figure 4.13a shows that AAQ sends maximum-sized A-MPDUs more than 90% of the time, except for cases when the AP sends to more then 9 Wi-Fi clients. To further investigate why, we examine A-MPDU size *per client* when the AP sends to 14 Wi-Fi clients, shown in Figure 4.14. We choose the case where the AP is transmitting to 14 Wi-Fi clients as this is the case when all of the Wi-Fi clients in our testbed are active. Two Wi-Fi clients in Figure 4.14 send

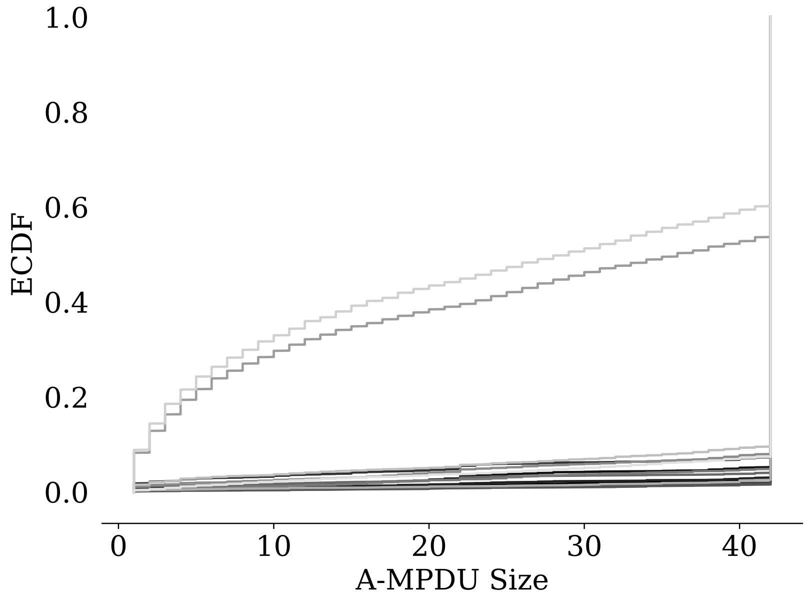


Figure 4.14: Per Wi-Fi client empirical CDF of A-MPDU size using AAQ. The solid black line is one client and lightest grey line is 14 clients.

small A-MPDUs, with only 41% and 43% of A-MPDUs maximum size, and about 6% of transmissions single-frame transmissions. Investigating further, we find that two Wi-Fi clients in question experience a higher loss rate, leading to MAC protocol retransmissions. Losses cause the Block Ack window to close, a phenomenon observed previously [72]. In Figure 4.15 we show the *per Wi-Fi client* distribution of the number of bad frames in an A-MPDU. A value of zero in the x-axis means all packets in an A-MPDU were successively acknowledged

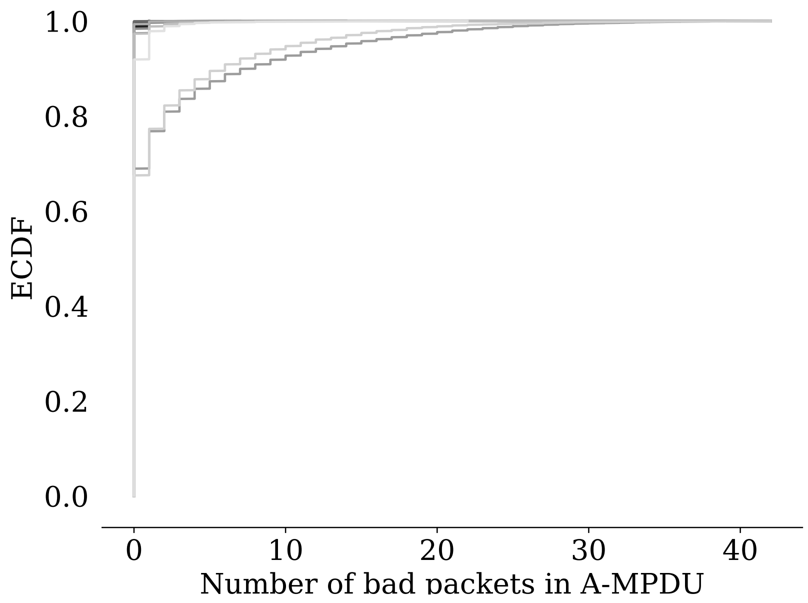


Figure 4.15: Distribution of number of bad packets in an A-MPDU *per Wi-Fi client* when the AP is transmitting to fourteen Wi-Fi clients. The solid black line is one client and lightest grey line is 14 clients.

by the receiver. More then 30% of A-MPDUs transmitted by the AP to the Wi-Fi clients in question, have one or more packets in an A-MPDU fail. These packet failures result in smaller A-MPDU sizes because the Block Ack window closes. Note that this phenomenon is not observed when the AP is running other schemes, as it is masked by empty TID queues.

The results in Figure 4.13 show that AAQ successfully sends large A-MPDUs when the AP forwards packets to Wi-Fi clients. But do the large A-MPDU sizes observed result in better goodput? Figure 4.16 shows mean aggregate goodput, averaged over five experiments, with standard deviation as error bars. When calculating aggregate goodput, for each TCP flow (one TCP flow per Wi-Fi client), we filter samples at the beginning of the flow and only consider samples when all TCP flows are active. To avoid in-phase back off of TCP flows, a phenomenon where TCP flows drops are synchronised, we randomise starting times for TCP flows. The results in Figure 4.16 show the aggregated one-second samples taken from the middle part of each experiment. To calculate aggregate goodput we first calculate the mean goodput each Wi-Fi client gets across all experiment runs, and then we sum the mean of each Wi-Fi client to get the aggregate goodput. The solid lines on top of each bar shows the best theoretical goodput achievable.[12] Compared to other schemes, AAQ achieves higher goodput as the number of Wi-Fi clients increases. When the AP sends to one or two Wi-Fi clients, all schemes but Linksys, achieve similar goodput. With so few clients, the limiting factor is the double buffering in the *ath9k* driver. As the number of Wi-Fi clients increases beyond two, AAQ achieves consistently better goodput then the other schemes. For example, when the AP is transmitting to 14 Wi-Fi clients, AAQ (92 Mpbs) improves goodput by 27%, 21%, and 19% compared to FQ CODEL (72 Mbps), FIFO (76 Mbps), and Linksys (77 Mbps), respectively. Linksys presents an interesting

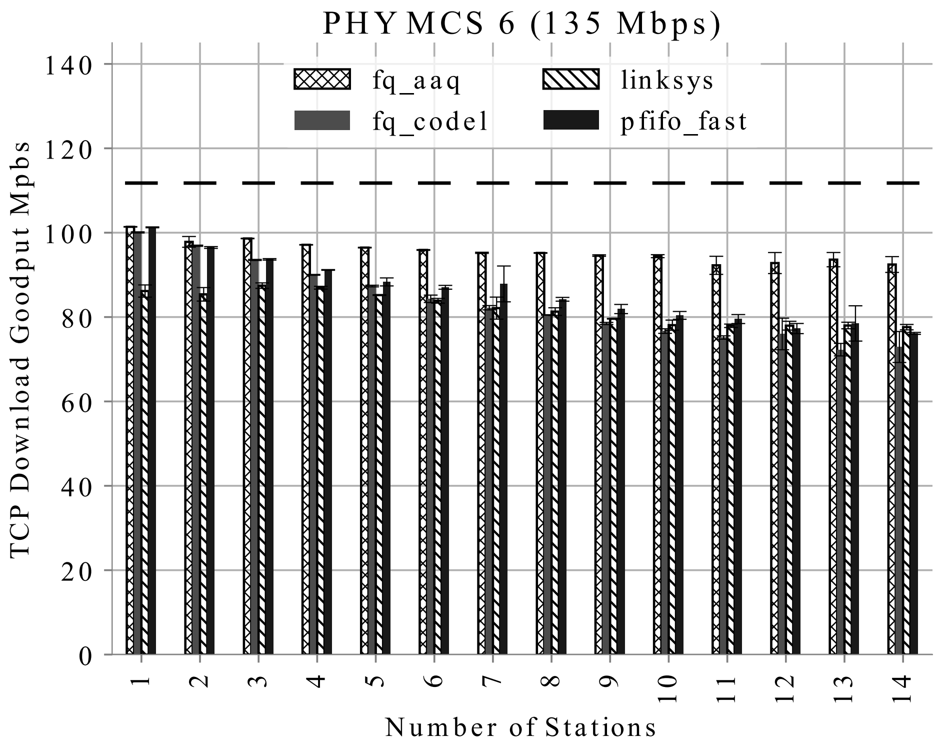


Figure 4.16: TCP downlink goodput when the AP transmits to varying number of Wi-Fi clients.

case where despite sending smaller A-MPDUs (see Figure 4.13d), it manages to keep the goodput at the same level as FIFO and FQ CODEL. We postpone

[12]The model used to calculate theoretical goodput is shown in in Appendix A.

the discussion of why is the case until later in this section.

Despite AAQ's goodput improvements, there is still a gap between the achieved goodput and the theoretical goodput. Multiple factors contribute to this gap. The theoretical model assumes a lossless channel and that A-MPDUs are transmitted with maximum size. In addition, the model does not take into account collisions between the AP and Wi-Fi clients transmitting TCP ACKs. As shown in Figure 4.15 the AP experiences some number of retransmissions when transmitting to a few Wi-Fi clients. Packet retransmission takes air time, in addition to reducing A-MPDU size as a result of the Block Ack window. Furthermore, A-MPDU sizes when Wi-Fi clients are transmitting TCP ACKs back to the AP are smaller in size than assumed by the model.

Next we look at the uplink A-MPDU sizes. In Figure 4.17 we show the distribution of A-MPDU sizes when Wi-Fi clients are transmitting TCP ACKs back to to the AP for AAQ, FIFO, FQ CODEL, and Linksys. As we have discussed in Section 4.6, Wi-Fi clients transmit TCP ACKs mostly as single packet transmissions. AAQ helps Wi-Fi clients form larger A-MPDUs when transmitting TCP ACKs as compared with FIFO and FQ CODEL. When using AAQ the AP sends large A-MPDUs resulting in the wireless medium's being busy for more of the time, once the AP has acquired the wireless channel. This results in Wi-Fi clients having to wait for longer periods of time before they can acquire the channel. This delay causes more TCP ACKs to be enqueued in TID queues, which helps in forming larger A-MPDUs for TCP ACKs.

We briefly mentioned that Linksys presents interesting behaviour. We now further investigate this Linksys behaviour. We note that when the AP is transmitting to Wi-Fi clients it sends small A-MPDUs (see Figure 4.13d), with a maximum A-MPDU size of 16 packets. Despite this small A-MPDU size, the Linksys AP manages to keep the aggregate goodput in line with that of FIFO and FQ CODEL (see Figure 4.16). In addition, in Figure 4.17d, we see that Wi-Fi clients transmit TCP ACKs back to the AP with larger A-MPDUs as compared to other schemes. We have seen that AAQ enables Wi-Fi clients to

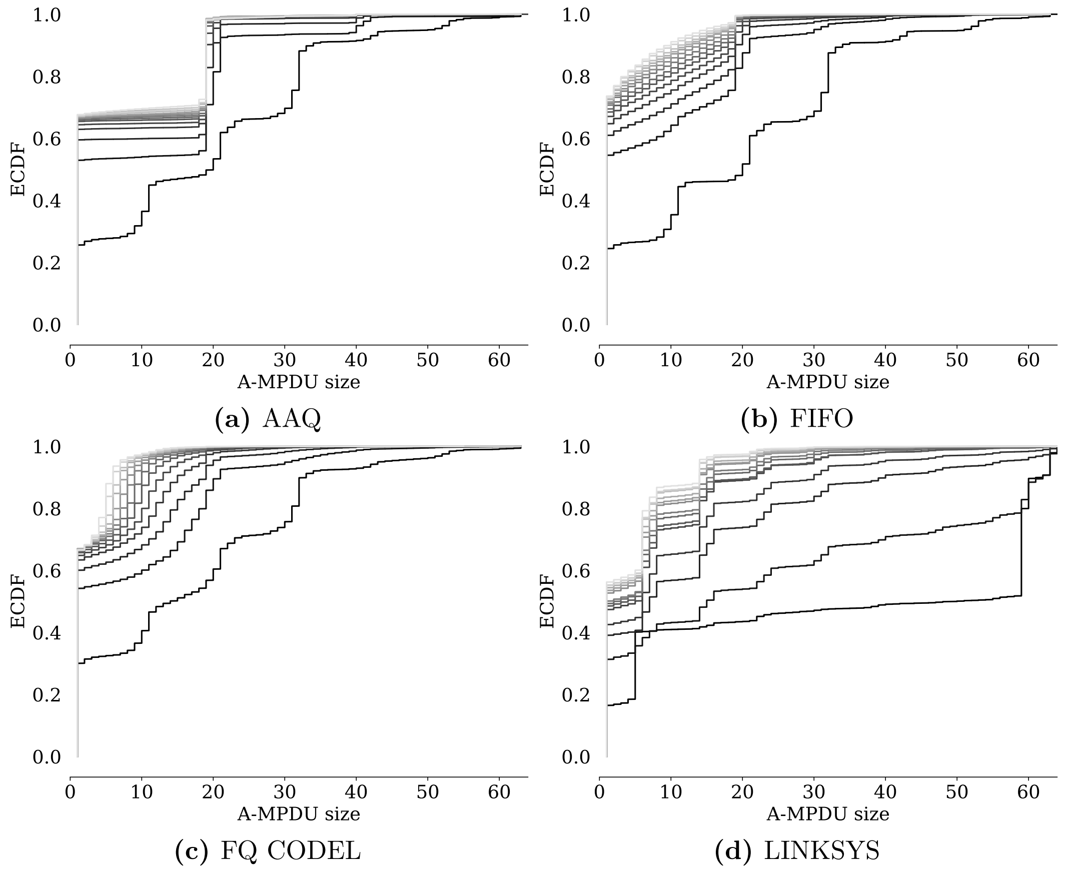


Figure 4.17: Distribution of A-MPDU sizes when Wi-Fi clients transmit TCP ACKs back to the AP. The solid black line is one client and lightest grey line is 14 clients.

send larger A-MPDUs because it keeps the medium busy for longer. However, Linksys sends a maximum A-MPDU size of 16 packets, so it can't be keeping the medium busy. Our hypothesis is that Linksys sends multiple 16-packet A-MPDUs back-to-back, as that is the only other strategy that could keep the wireless medium busy for longer.

We do not have access to the Linksys source code to instrument it for logging. We therefore rely on wireless traces captured by the monitoring station. We process these traces offline and count the number of back-to-back A-MPDUs Linksys transmits to Wi-Fi clients. Figure 4.18 shows the distribution of back-to-back A-MPDUs sent by Linksys and AAQ, for comparison. A value of of zero in the x-axis means that only one A-MPDU was transmitted by the AP. Most of the time Linksys sends back-to-back A-MPDUs with only 5% single

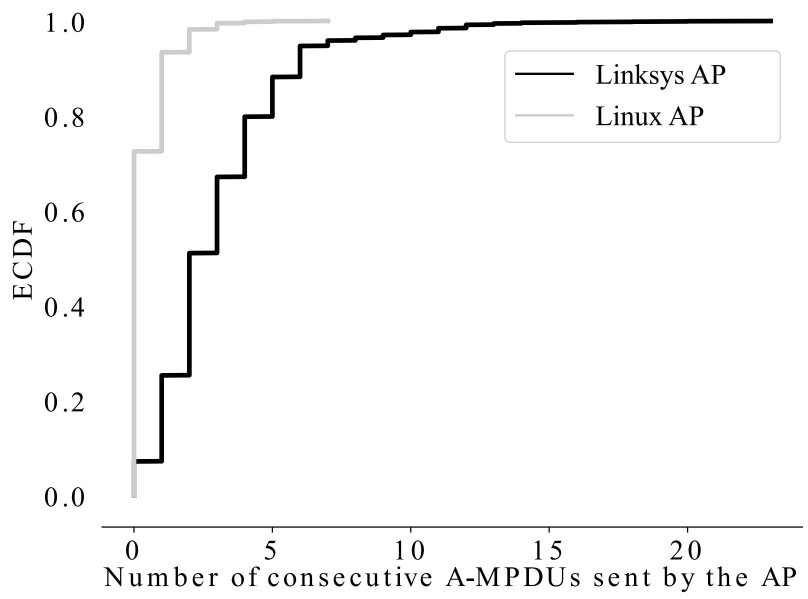


Figure 4.18: Empirical CDF for consecutive A-MPDUs transmitted by the AP.

A-MPDU transmissions. In contrast, AAQ sends single A-MPDU more than 70%. Linksys has a median consecutive A-MPDU transmission of two, resulting in three back-to-back A-MPDU transmissions. The median A-MPDU size when the Linksys AP transmits to Wi-Fi clients is 16 packets (see Figure 4.13d). Effectively, Linksys's median A-MPDU size is 48. When using Linksys Wi-Fi clients wait longer, on average, before they can acquire the wireless medium for transmission, resulting in more TCP ACKs enqueued in TID queues.

4.8.4 TAO Evaluation

The previous section examined AAQ's performance. We now repeat the same experiments as in the previous section, but with TAO enabled on clients. Again, we vary the number of Wi-Fi clients and instruct the sender to send TCP data to each client as fast as the sending socket allows. The results in Figure 4.19 show that TAO further improves goodput. When the AP is sending to fourteen Wi-Fi clients mean aggregate goodput improves by 10% for AAQ, by 30% for FIFO, 20% for FQ CODEL and 15% for Linksys. FIFO sees a significant goodput improvement, from 72 Mbps to 100 Mbps. The result in Figure 4.20 show the distribution of A-MPDU sizes Wi-Fi clients transmit when sending TCP ACKs. TAO virtually eliminates all single-frame transmissions across

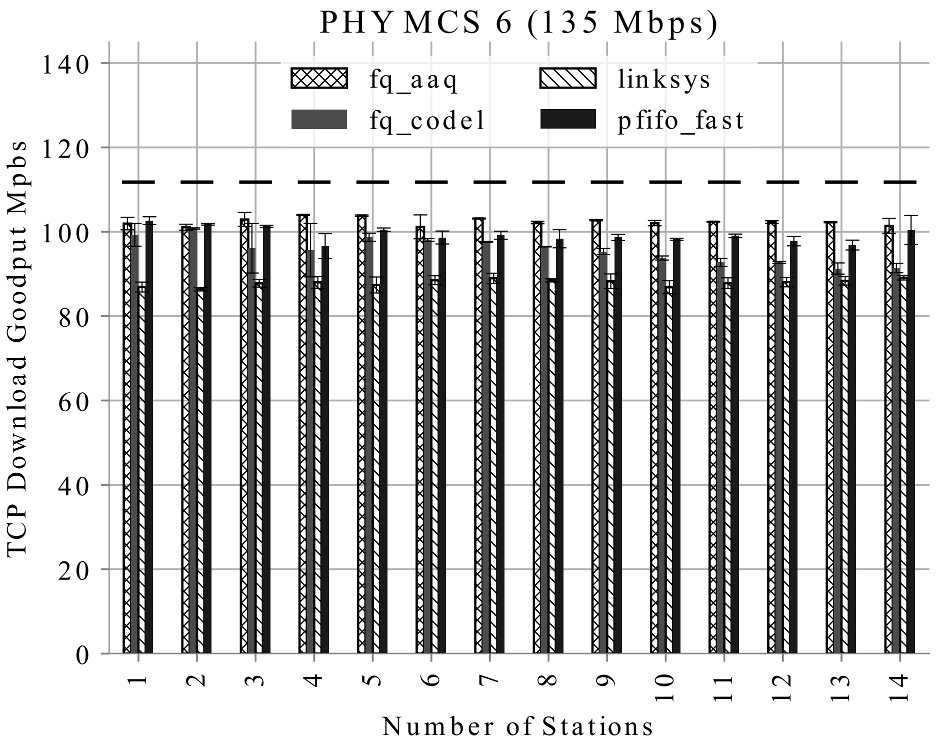
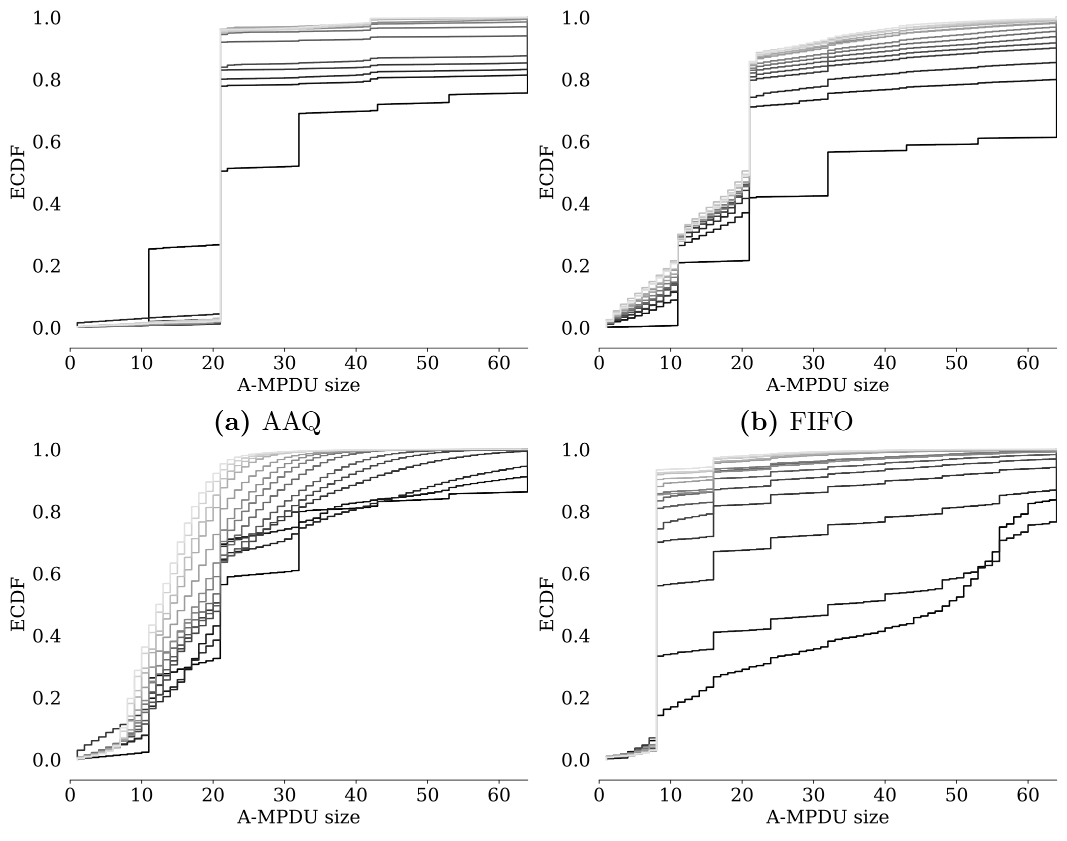


Figure 4.19: TCP Downlink Goodput with TAO Enabled

all the schemes. When using AAQ majority of A-MPDU sizes are 21 frames, which corresponds to half of the maximum-size A-MPDU sent by the AP (42 frames). Sometime, TAO sends more then 21 frames in an A-MPDU as a result of the interaction between the TCP ACK queue maintained by TAO, the TID queue and the double-buffered DMA queue. TAO may decide to release TCP ACKs, but released TCP ACKs may wait in the TID queue to be scheduled for transmission, resulting in more then 21 TCP ACKs being transmitted in an A-MPDU.

FIFO with TAO sees the greatest goodput improvement. This is interesting as it suggests that AAQ might not add any benefit as just running TAO with FIFO could be enough. Do we need AAQ, or is combining FIFO and TAO enough to achieve high goodput? To answer this question we need to look at the goodput fairness achieved.



(a) AAQ
(b) FIFO
(c) FQ CODEL
(d) LINKSYS

Figure 4.20: Uplink A-MPDU size empirical CDF when the AP is sending from one to fourteen Wi-Fi clients with TAO enabled. The solid black line is one client and lightest grey line is 14 clients.

4.8.5 Goodput Fairness

The results in Figure 4.19 suggest that combining FIFO and TAO could be enough to achieve high aggregate goodput and that AAQ may not be necessary. In this section we look at the goodput for each Wi-Fi client. When the AP sends to multiple Wi-Fi clients, it is important that each Wi-Fi client gets a fair share of the available goodput. In Figure 4.21 we show the distribution of one-second interval goodput each Wi-Fi client gets, across all experiments, when the AP transits to fourteen Wi-Fi clients. From Figure 4.21 we can conclude that AAQ, FQ CODEL, and Linksys are fair, and each Wi-Fi client gets a fair share of goodput. On the other hand, FIFO is not fair, as we would expect from a single FIFO queue system. FIFO's mean aggregate goodput is high because some Wi-Fi clients are penalised and obtain a smaller share of

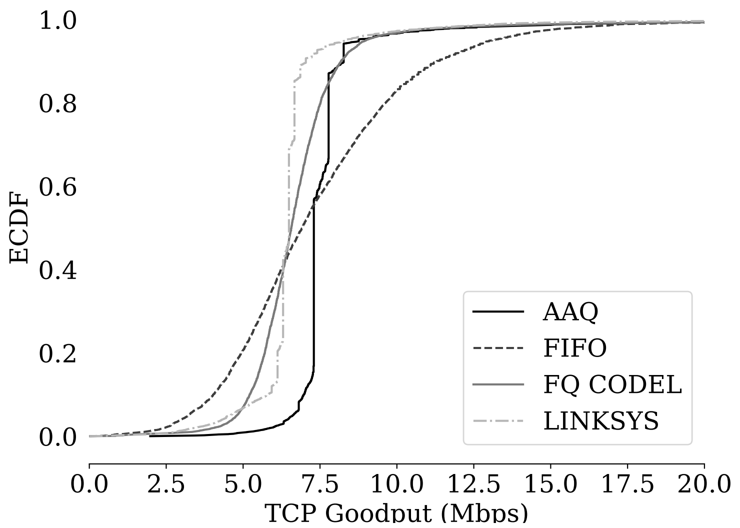


Figure 4.21: Empirical CDF of one-second interval TCP goodput of each Wi-Fi client when the AP is transmitting to 14 clients.

the available goodput.

4.9 Discussion

Providing information exchange mechanisms between layers increases A-MPDU size when Wi-Fi APs send, resulting in improved TCP goodput. We showed that AAQ improves goodput by as much as 27% compared to FQ CODEL and 21% compared to FIFO. AAQ achieves this while providing fair shares of available goodput to all Wi-Fi clients as a result of using fair queues. In bi-directional protocols such as TCP, faster transmission of TCP ACKs improves goodput. We showed that today's Linux implementation sends small A-MPDUs when transmitting TCP ACKs back to the sender. We designed TAO, a simple TCP optimisation technique that further improves aggregate goodput by transmitting large A-MPDUs when Wi-Fi clients send TCP ACKs back to the AP for forwarding. TAO improves goodput across all of the schemes we tested.

We evaluated AAQ and TAO using a fixed transmit rate (MCS-6). But Wi-Fi clients choose best transmit rate dynamically based on the wireless channel conditions. For AAQ and TAO to be successfully deployed on today's

Wi-Fi networks they must work even when the transmit rate varies. Three factors influenced the decision to evaluate AAQ and TAO using only one transmit rate. First, using a fixed transmit rate removes the dynamics of rate control algorithms, simplifying the analysis. Second, using a fixed rate let us choose a rate that would limit the Block Ack window as a result of losses. Finally, when we evaluated AAQ and TAO using Linux's default rate control algorithm, we noticed that A-MPDUs were broken into smaller sizes. In the next chapter we will look into this problem in more depth, and present a rate control algorithm design that can send large A-MPDUs and show that AAQ and TAO still produce large aggregates when using dynamic transmit rates.

Chapter 5

Aggregation Aware Rate Control

In the previous chapter we showed that by coordinating scheduling of packets between IP queues and device driver queues maximum-sized A-MPDUs can be transmitted, thus improving MAC protocol utilisation. In this chapter we continue our journey to improve MAC efficiency by reducing unnecessary channel acquisitions incurred by the rate control (RC) mechanism used for Wi-Fi.

The wireless channel is characterised by time-varying random effects that can cause bit-errors. One of the mechanisms used in Wi-Fi networks to communicate robustly despite the dynamics of the wireless channel is to adaptively adjust the transmit rate depending on channel conditions. IEEE 802.11 defines a range of transmit rates designed to cover various wireless channel conditions. Low bit-rates provide robustness for wireless channels with low SNR, while high bit-rates are designed for wireless channels with high SNR. While IEEE 802.11 specifies the list of transmit rates that can be used, it does not mandate how and when these rates should be used. Instead, selection of the best transmit rate is up to W-Fi implementations. Rate control (RC) algorithms play an important role in Wi-Fi performance and as such they, have been the subject of study for more then two decades, resulting in many RC algorithm proposals. In this chapter we will focus our attention on the RC scheme currently in Linux, and we will show that it falls short at sending large A-MPDU. To address this problem, we propose Aggregate Aware Rate Control (AARC) a new algorithm

MCS	Modulation	Coding Rate	Data Rate (Mbps)
0	BPSK	1/2	15
1	QPSK	1/2	30
2	QPSK	3/4	45
3	16QAM	1/2	60
4	16QAM	3/4	90
5	64QAM	2/3	120
6	64QAM	3/4	135
7	64QAM	5/6	150

Table 5.1: 802.11n MCS list for single spatial stream

that sends large A-MPDUs, thus improving MAC protocol utilisation and outperforming Linux's current RC scheme.

5.1 Background

The wireless channel is characterised by time-varying random effects that cause bit-errors. These effects, such as signal attenuation, multi-path fading and interference are time-varying, typically unpredictable processes that are accentuated by changes in the environment, such as the movement of people and objects [45]. IEEE 802.11 provides various PHY rates that can be used to combat these effects on the wireless channel. The supported rates come from the combining of various modulations and coding rates. IEEE 802.11n supports four different signal modulations, namely BPSK (Binary Phase Shift Keying), QPSK (Quadratic Phase Shift Keying), 16-QAM (16-Quadrature Amplitude Modulation), and 64-QAM. In addition to these modulation schemes, IEEE 802.11n supports $\frac{1}{2}$, $\frac{2}{3}$, $\frac{3}{4}$, and $\frac{5}{6}$ coding rates. The combination of modulation and coding rate gives the various PHY rates supported. Table 5.1 shows the modulation and coding schemes (MCSes) supported by IEEE 802.11n for one spatial stream. The data rate shown is the greatest PHY rate achieved using a 40Mhz channel and the short-guard-interval (SGI). The list of MCSes in Table 5.1 repeats for each additional spatial stream. For example, MCS-12 has the same modulation and coding as MCS-4, but there are two independent spatial streams, doubling the PHY rate.

Each MCS in Table 5.1 is best for some particular operating regime, which depends on the wireless channel conditions. If conditions on the channel are worse than this operating regime for the MCS, performance will suffer, as the receiver will not be able to successfully decode data transmitted using this MCS. In Figure 5.1 we show operating points for each MCS from Table 5.1. The operating points are shown as a function of SNR, and were generated using well known theoretical SNR-BER mapping equations [88, 31]. It is assumed that bit-errors are independently distributed, and the y-axis shows raw PHY rate (i.e., excluding overheads incurred by the MAC protocol and other higher lever protocols). The time-varying nature of the wireless channel makes it a challenging task to identify a wireless channel's current most appropriate PHY rate. Rate control algorithms aim to first identify the channel conditions and then decide the best MCS to use when transmitting. Ideally, RC algorithms would follow the envelope in Figure 5.1 (dashed line) by moving up or down a rate at the intersection points as wireless channel conditions change. At first finding the best MCS to use may seem like a trivial task: measure the SNR and use the SNR-to-BER mapping formulas to find the operating point. However, it has been shown [99] that in practice this approach does not work well in practice because of imperfections in hardware devices.

Broadly speaking, RC algorithms either use some PHY metric such as SNR to predict channel conditions, or they gather frame loss statistics. Further, they can be divided into algorithms that require feedback from the receiver, or make rate decisions independently. Earlier RC algorithms relied on SNR to make transmit rate decisions, but it has been shown that SNR is not a good metric for predicting wireless channel performance [46, 67]. As such, rate control algorithms that use frame delivery statistics have become the default type of RC used with today's off-the-shelf hardware. Their main advantage is simplicity: if the frame loss rate is high, use a lower (more robust) rate, or increase transmit power. While loss-based algorithms are able to find the best transmit rate for a slow varying wireless channel, they fail to adapt quickly to

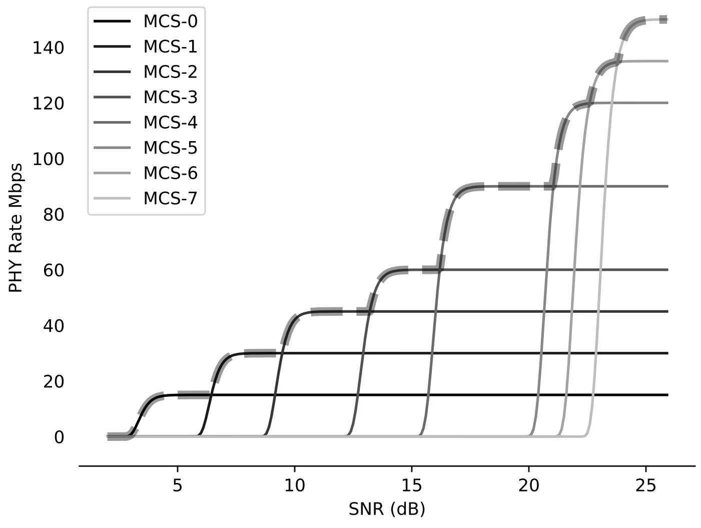


Figure 5.1: SNR vs PHY Rate Envelope for 802.11n rates.

dynamic environments (i.e., moving devices). Also, the latest 802.11 standards (IEEE 802.11n, IEEE 802.11ac), significantly increase the number of available transmit rates (and combination of different configurations), increasing the search space, thus rendering rate adaptation more complex.

A well known frame-based RC is SampleRate [67]. In addition to providing an implementation for off-the-shelf hardware (IEEE 802.11a only), the SampleRate authors also show that SNR is not a good predictor. SampleRate keeps frame delivery statistics for each Wi-Fi station and groups statistics for large and small frames. Minstrel and Minstrel HT [24] are similar to the SampleRate algorithm and are the default RC algorithms used in Linux. Minstrel is used for legacy IEEE 802.11 standards (IEEE 802.11a,IEEE 802.11g, and IEEE 802.11b) and Minstrel HT is used for the more recent IEEE 802.11n and IEEE 802.11ac. RC algorithms such as SoftRate [91] use fine-grained PHY information to decide on the best transmit rate. Since its publication, SoftRate has served as a benchmark for comparing RC algorithms. While a state-of-the-art RC scheme, it relies on low-level PHY layer information, which is not available from off-the-shelf hardware. More recently, eSNR [46] has been shown to be a simple solution using Channel State Information. Our AARC rate control is based on eSNR and, to the best of our knowledge, is the first RC design based on eSNR that works online using off-the-shelf hardware.

5.2 Minstrel HT

Minstrel is the default RC algorithm in Linux and is similar to SampleRate in that it uses frame loss rate statistics to estimate channel conditions and select the best transmit rate. For each station, it keeps a table of statistics for each of the supported rates. Rate statistics are filled in using data gathered from transmit reports for each of the rates used. To explore new (or update old) rates, Minstrel uses probing frames. Probing incurs overhead. As probing frames are more likely to be lost, Minstrel attempts to limit probing to 10% of transmitted frames. In addition, it has mechanisms to further reduce probing for cases when channel conditions are good and not changing. Minstrel randomises the rates to be explored, and for each Wi-Fi station it keeps a table of four rates (used as a retry policy table): the best throughput rate, the second-best throughput rate, the best delivery probability rate and lowest base rate. Combining single-frame probing and a retry policy gives Minstrel the advantage of quick recovery when the probing frame is lost.[1] Minstrel HT is an extended version ff Minstrel RC designed to work with IEEE 802.11n rates and configurations. The novelty of Minstrel HT is in the grouping used when exploring new rates. In the next section we highlight the problems that Minstrel HT probing incurs and describe our AARC scheme in Section 5.5.

5.3 Problem Statement

In the previous chapter we evaluated AAQ and TAO using a fixed bit rate only. Avoiding interaction with RC allowed us to simplify the dynamics of the system and better understand the performance of AAQ and TAO. However, for AAQ and TAO to be usable in real-world scenarios, they should also perform well when run under a RC algorithm. When enabling RC and running AAQ

[1]When sending probes the retry policy slightly changes: probing rate, best throughput rate, best probability rate, and lowest base rate.

and TAO, we observed that their benefits are diminished. The culprit is Minstrel's probing mechanism. Because probing frames might get lost, Minstrel HT probes using single-frame transmissions. When Minstrel HT probes a transmit rate, it marks the corresponding frame with a PROBING flag. When the device driver encounters probing frames, it ensures that these frames are transmitted as a single frame, by breaking an A-MPDU at the probing frame. Thus a single A-MPDU transmission often ends up being divided into three transmissions: the first part of the transmission is formed from the frames up to the probing frame, followed by a single probing frame transmission, and finally the remaining frames. To illustrate this problem we show the A-MPDU size when using Minstrel HT and the MCS-6 fixed transmit rate in Figure 5.2. We

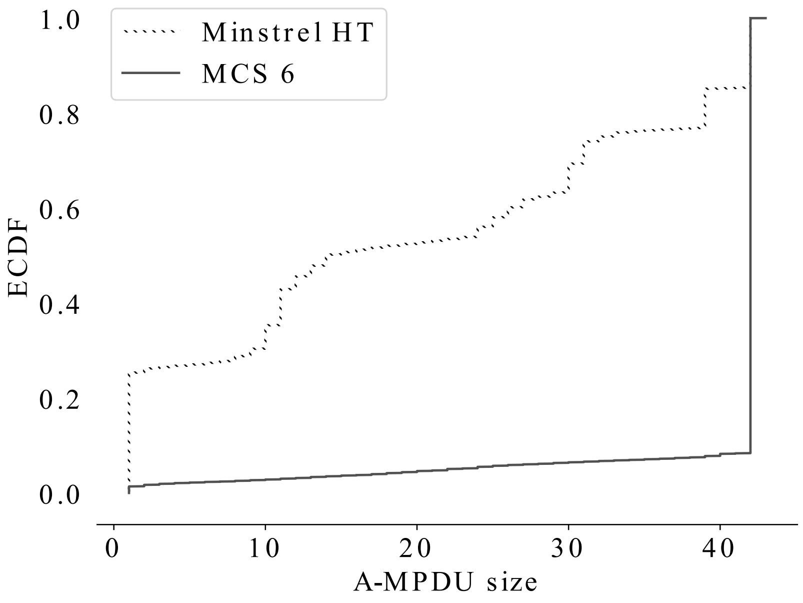


Figure 5.2: A-MPDU size Minstrel HT v Fixed MCS 6 Rate

can see that enabling Minstrel HT significantly reduces the number of frames that are transmitted in an A-MPDU. More than 20% of frame transmissions are single-frame transmissions, which incur a significant overhead. While not all of these frames are probing frames, the effect of the 10% probing frames used by Minstrel HT is magnified when it interacts with frame aggregation. In the next section we explain our aggregate friendly approach to RC.

5.4 Goal

The central question of this chapter is: is it possible to design an RC algorithm that can be implemented using off-the-shelf hardware and that manages to send large A-MPDUs? A simple straw-men approach to this problem is to probe using a whole A-MPDU, instead of probing using single frames. Two problems arise with this approach. First, probing with A-MPDUs is risky in cases when the probed rate is too fragile for the channel, as a large number of frames will be lost and hence retransmitted. This is the problem Minstrel HT tries to avoid by sending single-frame probes. The second problem is the limitation of the Block Ack window as a result of the experienced losses. As was explained in Chapter 4 (Section 4.3), frame losses leave “holes” in the Block Ack window, reducing the number of frames that can be transmitted in subsequent A-MPDUs. Next we will show a simple RC algorithm and explain why that approach is not sufficient. Motivated by those limitations we then present our full RC algorithm, AARC.

5.5 Aggregation Aware Source Rate Control (AASRC)

We first describe an initial approach to aggregate-friendly RC, Aggregation Aware Source Rate Control (AASRC) and empirically show cases where this simple design is not enough. This motivates the AARC design described later in the chapter. AASRC is an RC algorithm where the transmitter (the source) makes rate decisions independently. As mentioned earlier, sending probes using full-sized A-MPDUs would eliminate the problem Minstrel HT has with sending single-frame probes. However, losing whole A-MPDUs could be expensive, especially if an A-MPDU is lost multiple times (i.e., at probing higher rates). To minimise A-MPDU losses during probing, AASRC introduces an exponential backoff mechanism that will delay subsequent probings of a rate. When probing a new rate, AASRC sends a full A-MPDU. When AASRC decides (we explain later in the chapter how AARC decides if it should switch to the new rate) that the rate being probed is not better than the currently used rate, a backoff mechanism for the probed rate is invoked. Subsequent probings of the rate are postponed until the backoff timer has expired. The backoff time is calculated using Equation 5.1. b starts at zero, and is incremented for every failed probe for a rate, until b_{max}. When a probe succeeds, b is reset to zero.

$$MCS_{backoff} = 2^b \; \{ \text{ for } b = 0, .., b_{max} \; \} \tag{5.1}$$

Like Minstrel HT, AASRC keeps statistics for the number of transmitted frames and the number of failed frames for each MCS. In addition, AASRC tracks (using an EMWA) the A-MPDU size for every supported rate. To decide what rate to use AASRC uses throughput Equation 5.5.

$$Util = \frac{AMPDU_{tx_{time}}}{MAC_{overhead} + AMPDU_{tx_{time}}} \tag{5.2}$$

$$AMDPU_{tx_{time}} = \frac{AMPDU_{avg_{size}} \times PKT_{length}}{PHY_{rate}} \tag{5.3}$$

$$SuccessRate = 1 - \frac{failed_{frames}}{sent_{frames}} \tag{5.4}$$

$$Tput = SuccessRate * Util * PHY_{rate} \tag{5.5}$$

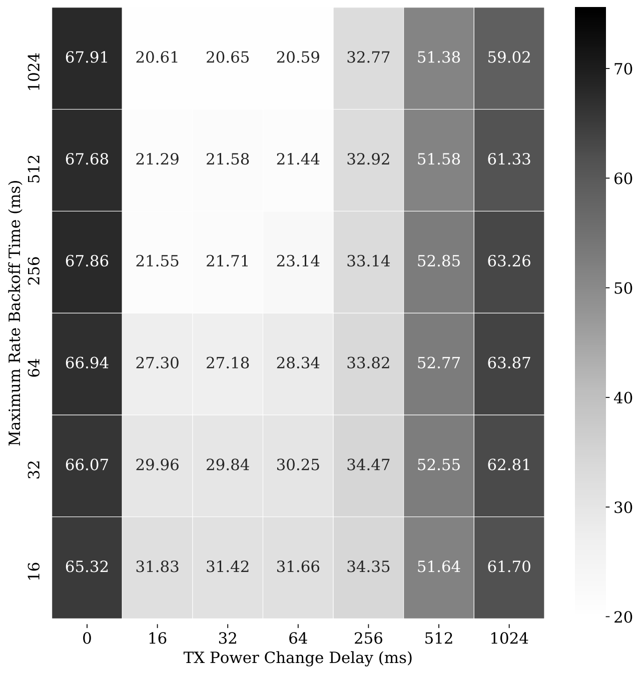
Initially, when there are no statistics collected for a rate, the PHY_{rate} is used to compare rate performance. PHY_{rate} is the raw physical transmit rate of the rate (see Table 5.1). $MAC_{overhead}$ captures the 802.11 MAC protocol[2] overheads, and PKT_{length} is set to 1200 bytes. AASRC starts at the lowest rate (MCS 0) and progresses upwards, probing using full A-MPDUs. When Equation 5.5 yields that the throughput of the probed rate is lower than that of the currently used rate, the backoff procedure described above is invoked and the next probe is delayed. Unlike Minstrel HT, AASRC only uses linear probing: it only probes the rate above the current rate, and only advances one

[2]These overheads come from average contention time, DIFS, and SIFS.

rate at a time. When channel conditions worsen, i.e. the current rate is not supported by the channel, AASRC switches to the rate below immediately and continues to go downwards until a working rate is found. When switching to a lower rate, AASRC also resets statistics for the higher and the current rate prior to switching. One crucial parameter that affects AASRC's performance is the maximum backoff parameter. Setting it to a low value will incur frequent probing, resulting in poor performance. Setting it to a high value may result in missing a probing opportunity, for cases when the channel has improved. We choose a maximum backoff value of 32 ms, which was derived empirically as we explain in the next section.

5.5.1 Empirical Evaluation of the Maximum Backoff Parameter

Choosing a correct maximum backoff parameter for AASRC is important as it will have a direct impact on performance. To choose the value for maximum backoff, we empirically evaluate a range of values under varying channel conditions. We set up a Wi-Fi client, an AP and a sender. The sender is connected to the AP via Gigabit Ethernet and establishes a TCP connection with the Wi-Fi client. To emulate varying channel conditions we vary the transmit power on the AP. We emulate a random walk by randomly choosing transmit power change values in the range of [-2,-1,0,1,2] dBm. Negative values indicate that the transmit power is reduced by the given amount, zero indicates that the transmit power remains the same, and positive values indicate the transit power increases by the given amount. The channel delay specifies the time between transmit power changes. For example, for a channel delay of 32 ms the AP transmits for 32 ms at the chosen transmit power before moving to the next transmit power. For each combination of Channel Delay and Maximum Backoff, the sender sends for 10 seconds. Each experiment is repeated 20 times and the experiments are run during the night to minimise the effects of changes in the surrounding environment. Figure 5.3 shows a heatmap of the goodput achieved for various configurations. Column 0 represents the default



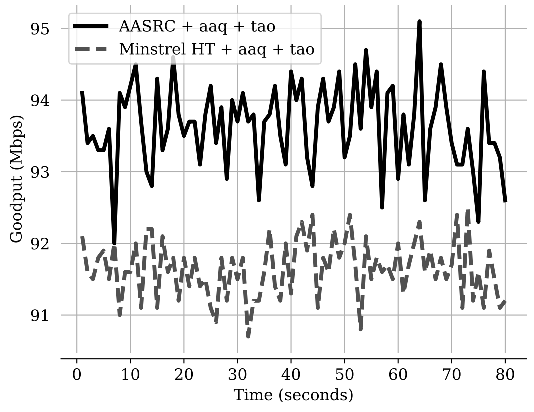
Maximum Rate Backoff Time (ms) \ TX Power Change Delay (ms)	0	16	32	64	256	512	1024
1024	67.91	20.61	20.65	20.59	32.77	51.38	59.02
512	67.68	21.29	21.58	21.44	32.92	51.58	61.33
256	67.86	21.55	21.71	23.14	33.14	52.85	63.26
64	66.94	27.30	27.18	28.34	33.82	52.77	63.87
32	66.07	29.96	29.84	30.25	34.47	52.55	62.81
16	65.32	31.83	31.42	31.66	34.35	51.64	61.70

Figure 5.3: Heatmap of Maximum Backoff Parameter vs Channel Variability

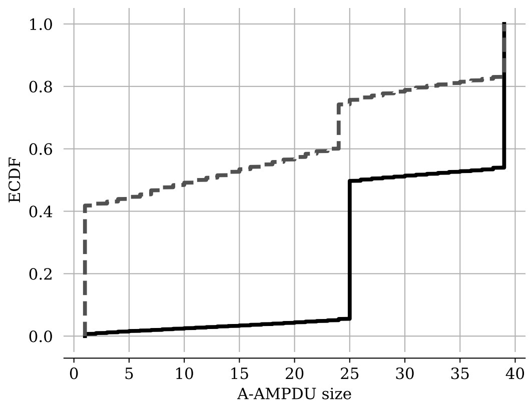
behaviour, where the transmit power remains unchanged. As can be seen, for a static channel, choosing a higher back-off value yields the best results. This is expected as when the channel does not change, probing only incurs overhead, and choosing a large maximum backoff value incurs less such overhead. The inverse is true for dynamic channels. A large maximum backoff value performs poorly as AASRC is "blind" to varying channel conditions. Based on the results in Figure 5.3, we choose a maximum backoff of $32ms$ as a compromise between maximising the goodput achieved on a static channel vs. on a dynamic channel.

5.5.2 Deficiencies of AASRC

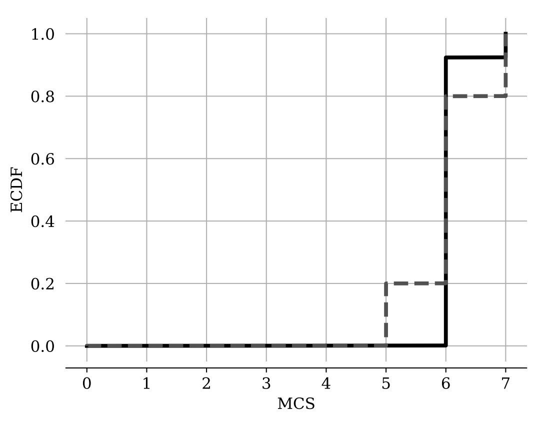
AASRC works well in relatively static environments. However, on more dynamic channels, the backoff mechanism will prevent AASRC from reacting in a timely fashion. In this section we show two examples from our empirical evaluation (see Section 5.9) that highlight two types of environments that illustrate AASRC's performance regimes. While these two examples are a small subset of all possible conditions, and not broadly representative, the purpose here is to highlight the benefits and disadvantages of AASRC when operating in these two contrasting environments.



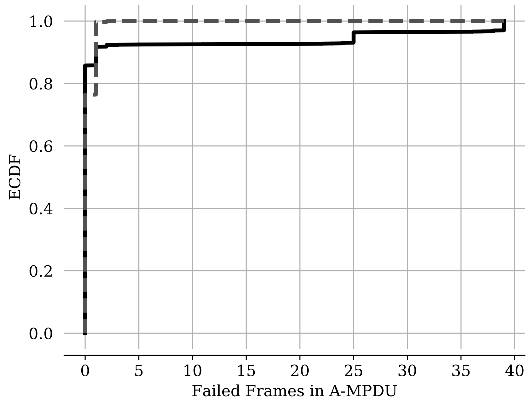
(a) One-second samples of goodput.



(b) Empirical CDF of A-MPDU size.



(c) Empirical CDF of MCS used.

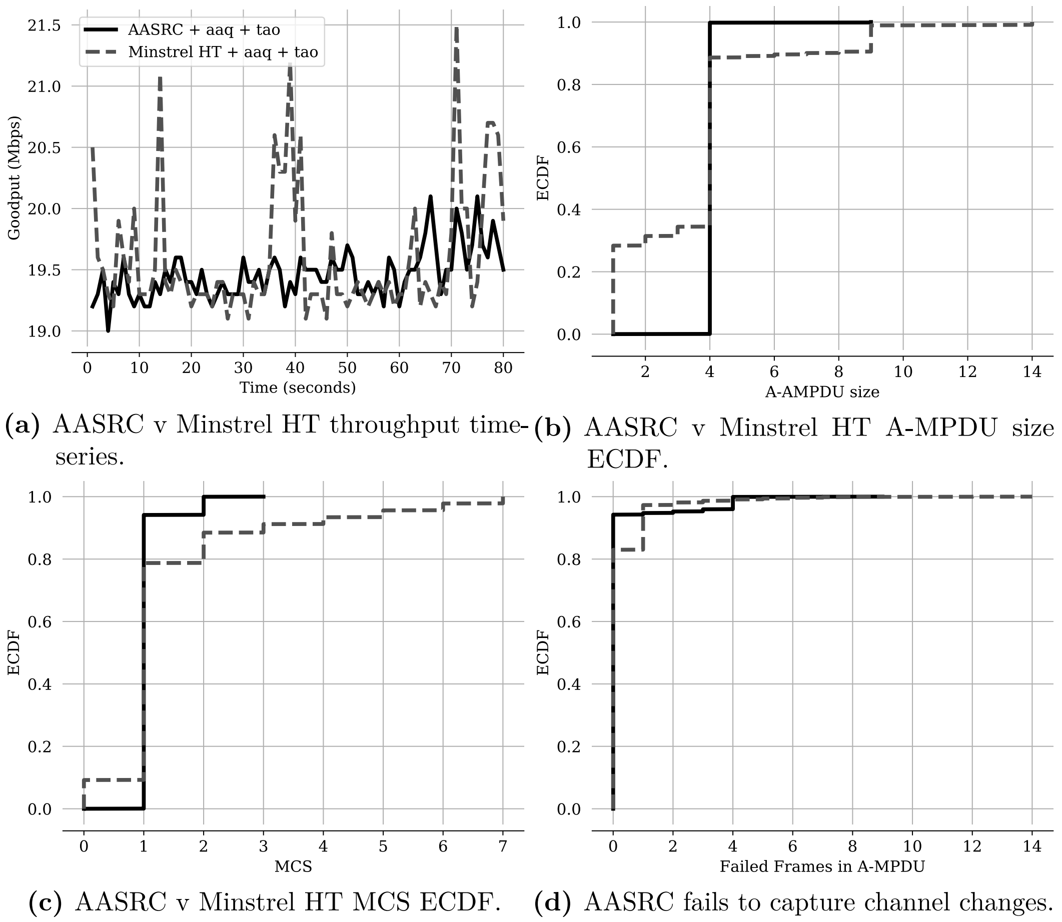


(d) Empirical CDF of number of failed frames in an A-MPDU.

Figure 5.4: AASRC v Minstrel HT under static channel conditions.

Figure 5.4 shows the case of a fairly static channel where the AP is transmitting to a single Wi-Fi station. Note that the y-axis does not start at zero and the purpose here is to highlight differences, and not compare performance. Figure 5.4a shows that because AASRC sends large A-MPDUs it achieves slightly (2%) higher goodput for one-second samples. Figure 5.4b shows that AASRC manages to send large A-MPDUs and virtually eliminates single-frame transmissions. Alternating between 39 and 25 frames in an A-MPDU is a result of double buffering used in the device driver (see Chapter 4, Section 4.6). On the other hand, Minstrel HT sends single frames more then 40% of the time. AASRC uses MCS-6 more then 90% (see Figure 5.4c) of the time and probes the higher MCS, MCS-7, about 7% of the time. This indicates that the backoff mechanism helps in reducing frame losses. This probing is reflected in the number of failed frames in an A-MPDU, shown in Figure 5.4d.

While AASRC manages to preform well under static channel conditions, in more dynamic environments (Figure 5.5), AASRC fails to adapt to channel conditions quickly enough. In this environment, Minstrel HT's random probing



(a) AASRC v Minstrel HT throughput time-series. **(b)** AASRC v Minstrel HT A-MPDU size ECDF. **(c)** AASRC v Minstrel HT MCS ECDF. **(d)** AASRC fails to capture channel changes.

Figure 5.5: AASRC fails to adapt to changing channel conditions.

manages to capture the channel's variability, but AASRC simply does not probe as a result of the backoff timer. This can be seen in Figure 5.5c.

In summary, AASRC performs well for channels that are relatively static as, it will probe using full A-MPDUs and reduce probing for rates that are not supported by the channel. However, the backoff mechanism prevents AASRC from being responsive enough under dynamic channels. Ideally we'd have an RC that has all the benefits of AASRC yet is able to react to sudden channel changes. In the next section we describe AARC and RC scheme with all the benefits of AASRC, but which further introduces a mechanism that will cancel the backoff timer in cases where channel conditions have improved.

5.6 Aggregation Aware Rate Control (AARC)

As explained in the last section, AASRC won't be able to detect improved channel quality until its backoff timer has expired. It is desirable that AASRC is able to switch to a better rate as soon as possible if a channel change is detected. In this section we present Aggregation Aware Rate Control (AARC), a RC algorithm that relies on receiver feedback to detect channel improvements, so that the sender can react to them before its backoff timer expires. To achieve this, AARC uses channel state information, fine-grained information that gives a detailed view of the status of the channel.

5.6.1 Channel State Information

Unlike RSSI/SNR, CSI is a fine-grained metric that shows channel gains for each of the OFDM sub-carriers used in IEEE 802.11n to transmit data [46]. CSI is used in the decoding process at the receiver to compensate for the effects of the wireless channel. While off-the-shelf hardware internally uses CSI to decode received frames, this information is not reported to the device driver, and it has remained information internal to hardware until recently. The first platform to extract CSI from off-the-shelf hardware used Intel wireless cards to export CSI from the hardware to the upper layers [47]. More recently, a similar platform was developed that enables wireless network interfaces based on the Atheros chipset to export CSI to the upper layers [96]. Figure 5.6 shows three examples of CSI information exposed by the hardware on the Atheros platform. The x-axis represent sub-carriers used to transmit data. A total of 114 sub-carriers are used when transmitting using 40Mhz channel bandwidth. Each sub-carrier's CSI is a complex number showing the power gain and the shift of the signal of that sub-carrier. The y-axis in Figure 5.6 shows the channel power in decibels for each of the sub-carriers. Unlike RSSI/SNR, CSI provides rich information about each of the sub-carriers used for data transmission. For example, with CSI1 we can see that sub-carrier 43 experiences deep fading, which is likely to cause frame errors. This effect is not captured by RSSI/SNR. Figure 5.6 shows CSI after it has been processed; in the next section we show

how we process raw CSI reported by the hardware.

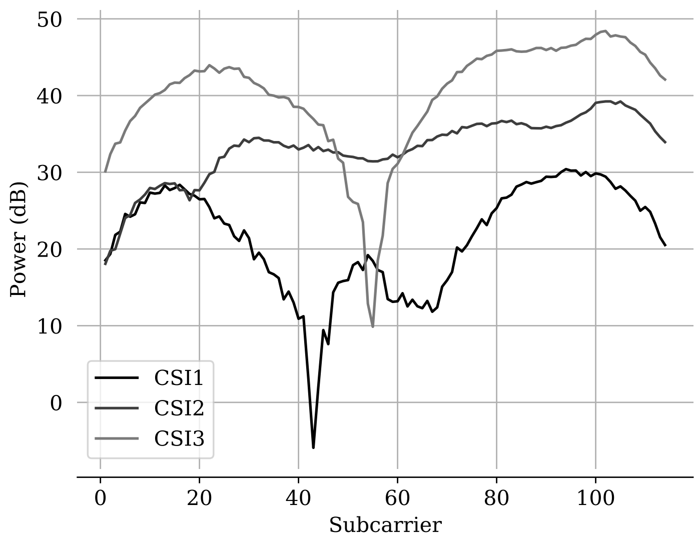
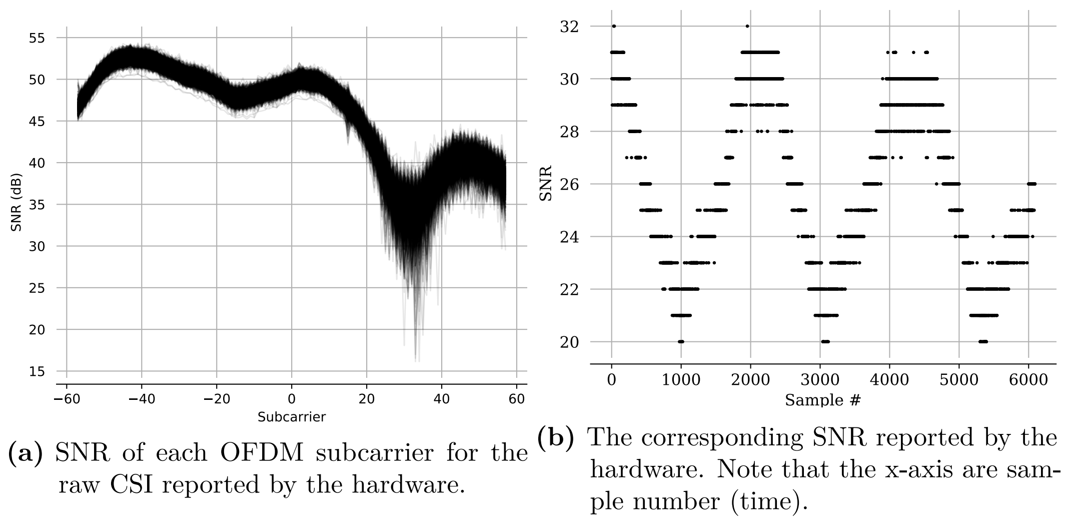


Figure 5.6: Example CSI

5.6.2 Processing CSI

To extract CSI from Atheros Wi-Fi hardware we use the ath9k-csitool device driver changes [96]. ath9k-csitool has three major components: the first component enables the correct registers to instruct the hardware to pass CSI up to the device driver; the second component is the code that parses raw bytes of CSI into a CSI matrix; and the last component is a module that exports CSI to user-level applications for processing. In our work we only use the first two components and write a Netlink [2] component, similar to the CSI tool for Intel cards [46], enabling user-level applications to download CSI from the device driver. The advantage of using the Netlink layer is that applications can use use the well-known socket API to receive CSI information. In addition, we also wrote code that allows CSI to be written in PCAP [17] format for offline processing. The reported CSI is a $N_{tx} \times N_{rx} \times N_{sc}$ complex matrix, where N_{tx} is the number of transmit antennas, N_{rx} is the number of receive antennas and N_{sc} is the number of OFDM sub-carriers used for transmission. For 20 Mhz channels, N_{sc} is 56 and for 40 Mhz channels N_{sc} is 114. For the purposes of this work we only consider a 1 by 1 antenna configuration and a 40 Mhz channel, therefore our raw CSI is a vector with 114 entries. While the techniques we use work for MIMO systems, we don't have documentation for how our hardware processes multiple streams and the spatial matrices it uses. This information must be used in processing of MIMO CSI. Each entry in the CSI matrix is a 20-bit complex number, using 10 bits each for the real and imaginary parts.

CSI provides rich information about power attenuation and the shifting of the signal as it propagates through the environment. For the purposes of AARC, we are only concerned with the power attenuation profile of the signal, as it is the only piece of information required by AARC to make decisions. CSI is an undocumented feature, and through experimentation and trial-and-error we determined how to process CSI. We observe that the reported power of raw CSI is with respect to an unknown reference, as the power of raw CSI does not change with changes in transmit power. To verify this, we set up



(a) SNR of each OFDM subcarrier for the raw CSI reported by the hardware.

(b) The corresponding SNR reported by the hardware. Note that the x-axis are sample number (time).

Figure 5.7: Changes in transmit power are not reflected on the raw CSI reported by the hardware.

an AP and a client and we instruct the AP to transmit to the client while we vary transmit power at the AP. At the client, we record the CSI and SNR information reported by the receiving hardware. Figure 5.7 shows that despite the transmit power changes being reflected in the measured SNR (Figure 5.7b), the CSI (Figure 5.7a) does not reflect these changes. This behaviour is similar to that reported in the Intel CSI project [46], and we use the same method to normalise and scale the reported CSI. The normalisation procedure is as follows: First we calculate the mean power, Power$_{mean}$, as follows:

$$Power_{mean} = \frac{1}{N_{sc}} \times \sum_{i=0}^{N_{sc}-1} |CSI_{raw}|^2$$

then the normalised CSI CSI$_{normalised}$ is :

$$CSI_{normalised} = \frac{CSI_{raw}}{\sqrt{Power_{mean}}}$$

The normalisation process has two effects on the raw CSI. First, it removes the unknown reference power, and second, it reduces noise present in the raw CSI. Note that in Figure 5.7a there is a $\sim 3dB$ band and the normalisation reduces the band to $\sim 1dB$ (see Figure 5.8).

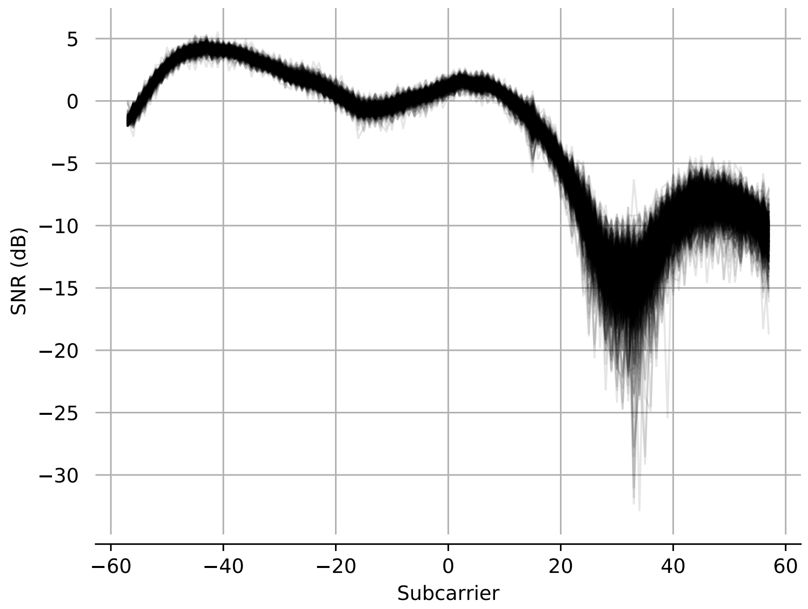


Figure 5.8: The result of applying normalisation to the raw CSI in Figure 5.7a

Then we calculate scaled CSI, CSI$_{scaled}$, as follows:

$$CSI_{scaled} = CSI_{normalised} \times \sqrt{SNR_{power}}$$

This procedure will scale the normalised CSI with respect to the reported SNR. SNR$_{power}$ (linear scale) is the signal-to-noise ratio reported by the hardware for each raw CSI. Unlike previous work that uses 20 Mhz channels [46, 96] we use 40 Mhz channels for AARC. When using a 40 Mhz channel the hardware reports two SNR values, SNR$_{ctrl}$ for the first (control) channel and SNR$_{ext}$ for the secondary (extended) channel. In this work SNR is the mean of SNR$_{ctrl}$ and SNR$_{ext}$. We decided to use the mean (in linear scale) as we found empirically that it gives the best results. The resulting CSI$_{scaled}$ is used as the input to calculate eSNR, as explained in the next section. AARC then uses eSNR to react to channel changes.

5.6.3 Computing effective SNR (eSNR)

We replicate here the steps required to compute eSNR [46]. Before computing eSNR, first the effective bit-error-rate (eBER) is computed. To compute eBER, power for each OFDM sub-carrier is calculated:

$$\rho[i] = |csi[i]|^2 \tag{5.6}$$

$csi[i]$ is the complex number representing CSI_{scaled} for sub-carrier i. From sub-carrier power one can calculate uncoded BER ρ for each sub-carrier i. BER is calculated using the well known textbook equations [88] shown in Table 5.2.

Modulation (m)	**Bits/Symbol**	$BER_m[i]$
BPSK	1	$Q(\sqrt{2\rho[i]})$
QPSK	2	$Q(\sqrt{\rho[i]})$
16QAM	4	$\frac{3}{4}Q(\sqrt{\frac{\rho[i]}{5}})$
64QAM	6	$\frac{7}{12}Q(\sqrt{\frac{\rho[i]}{21}})$

Table 5.2: Calculate BER from SNR ρ for sub-carrier i

Q (qfunction) is the standard normal CDF. After calculating BER for each sub-carrier, the effective BER is calculated as follows:

$$eBER_m = \frac{1}{N_{sc}} \sum_{i=0}^{N_{sc}-1} BER_m[i] \tag{5.7}$$

That is, eBER is the mean BER across OFDM sub-carriers. Note that eBER is calculated for each modulation, as different modulations have different constellation density. Finally effective SNR is calculated using the inverse BER_m^{-1}:

$$eSNR_m = BER^{-1}(eBER_m) \tag{5.8}$$

AARC uses eSNR_m to detect channel changes.

5.7 AARC Design

AARC uses eSNR to decide if the probing backoff timer should be canceled and probing initiated. Calculating eSNR from CSI results in four eSNR values, one for each supported modulation. The calculated eSNR values are for uncoded channels, as the equations in Table 5.2 for calculating BER do not take coding into account. Prior work has proposed a calibration process to map an uncoded channel to a coded channel [46]. That calibration process involves transmitting using various power levels and measuring eSNR values and their corresponding frame error rates (FER). Note that it is not possible to directly measure BER from off-the-shelf hardware, so FER is the closest metric that we can measure. The prior work's calibration process generates mapping tables that map eSNR to FER for each of the supported rates. These curves are then used to make rate decisions. When trying to generate our own calibration curves using our hardware we found that as a result of noisy SNR measurements the resulting curves were insufficiently accurate to use directly to set the MCS. We observed multiple overlaid curves depending on channel conditions. Instead, AARC uses the difference between two eSNR values measured at different points in time as an indicator of the trend in the channel's status.

Figure 5.9 shows the architecture of AARC. It consists of two main components. The user-level application (CSID) runs on both the AP and clients. CSID receives raw CSI from the device driver, processes the CSI and calculates the eSNR, which is then transmitted back to the transmitter via a feedback channel. On receiving eSNR feedback, CSID reports the value to the AARC module in the kernel. AARC operates in the same way as AASRC until a

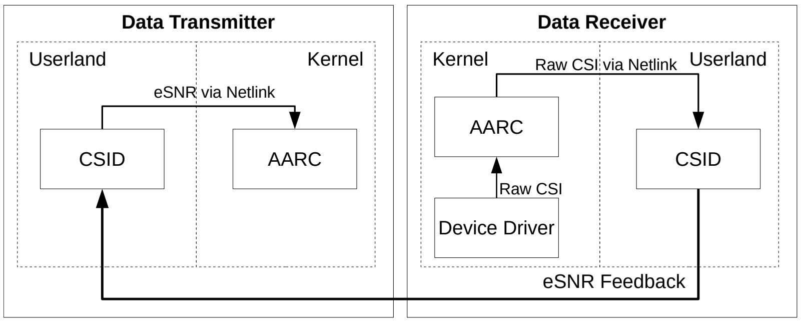


Figure 5.9: Architecture of AARC

probing rate fails, at which point AARC stores the current eSNR. For each subsequent transmission, eSNR is reported to the AARC module and the difference between the most recent eSNR and the eSNR recorded at the time of failure is calculated. If the difference calculated is bigger than a threshold (see Section 5.7.1), a probe is initiated without waiting for the backoff timer to expire. The motivation for this design is that AARC works well under stable channel conditions, where the backoff timer reduces number of probes sent. However, we want AARC to detect channel changes as soon as possible and react to them quickly.

5.7.1 Empirical Evaluation of eSNR difference threshold

We take a similar approach as we did in AASRC when we empirically evaluated the maximum backoff timer. To evaluate the eSNR difference threshold that will trigger a probe to a higher rate, we evaluate various eSNR differences under varying channel conditions. Figure 5.10 shows the heatmap of the goodput achieved. From the results in Figure 5.10 we choose an eSNR difference threshold of 1 dB, as that gives the best overall performance across the range of channel variations.

5.7.2 Empirical Evaluation of the Maximum Backoff Parameter

We used the same apporach we used for AASRC to determine the maximum backoff parameter (see Section 5.5.1). Figure 5.11 shows the heatmap for maximum backoff for AARC. We choose a value of 256ms as it gives the best

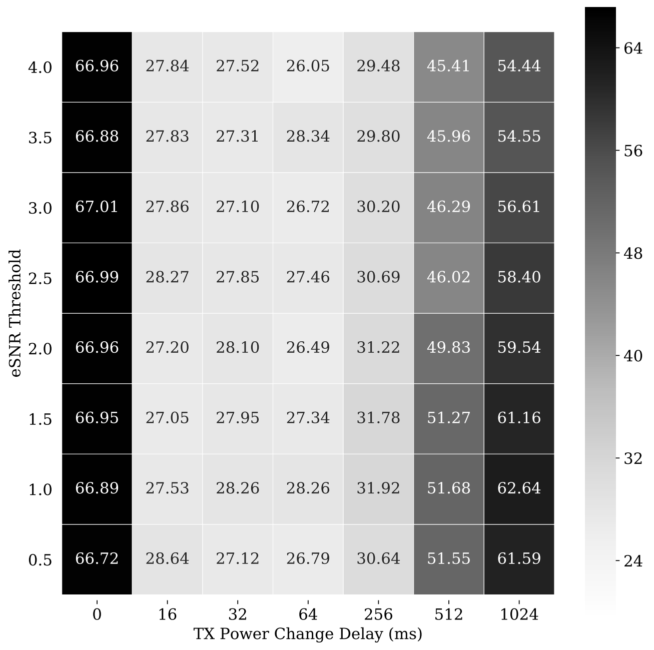


Figure 5.10: Heatmap of the goodput achieved (Mbps) as a function of eSNR difference under varying channel conditions

balanced throughput. Note that this is larger then AASRC's which we set to 32ms. Given that AARC reacts to channel changes, and cancels backoff when the channel improves, doing so allows for a larger maximum backoff which is beneficial when the channel remains static.

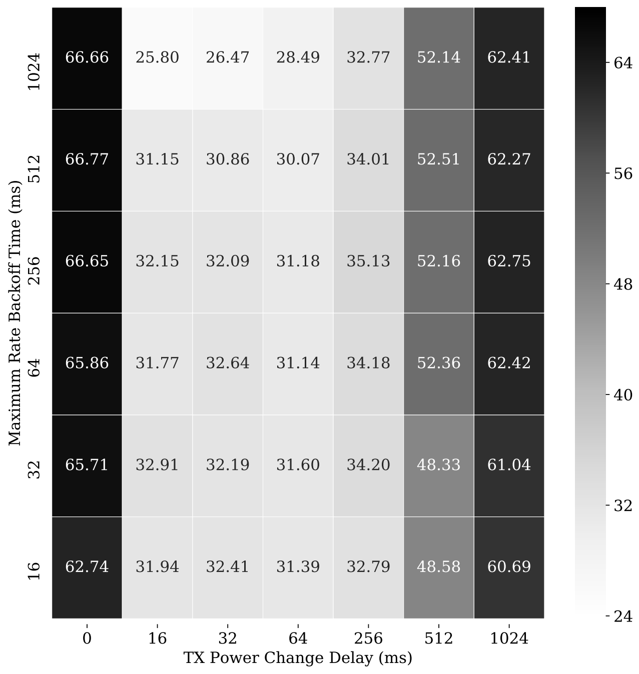


Figure 5.11: Heatmap of goodput achieved (Mbps) as a function of maximum backoff parameter under varying channel conditions for AARC

5.8 Implementation

We implement AARC as a mac80211 layer module in Linux kernel version 3.18.7 and implement CSID as a user-level application written in C and C++. Implementing CSID at user level simplified implementation and debugging. For production deployment, CSID functionality could be implemented in the kernel.

On the kernel side there are two main components of AARC. The first component handles the CSI data and the second implements the AARC algorithm.

5.8.1 Handling of CSI Data

The hardware at the receiver reports CSI information only if transmitted frames have been marked with the sounding flag. Furthermore, hardware registers at the receiver hardware have to be set as published in the ath9k-csitool [96]. The receiver's hardware uploads the CSI data to the device driver in two separate "transactions". The user data payload is uploaded first, followed by the CSI data. When uploading the user payload the hardware sets the *more_data* field of the receive descriptor. The device driver treats the data payload as part of a larger fragmented frame and buffers until the next "fragment", consisting of the CSI data, is uploaded. Once CSI data is transferred from the hardware, the device driver "de-fragments" both the user payload and the CSI data into a single packet. This "glued" packet is then passed up the networking stack for processing. The original ath9k-csitool only copies the CSI data into its internal buffer and leaves the "glued" packet intact. This leads to frames being dropped further up the stack, resulting in low throughput. We modify the original ath9k-csitool such that the payload data and CSI data are separated before forwarding to the upper layer. In order to do so, we store extra information in the control block of the *struct sk_buff*. We extend the *struct ieee80211_rx_status* and add a new structure that contains CSI-related data.

```
struct ieee80211_csi {
     u_int16_t csi_len;
     u_int8_t n_rx;
     u_int8_t n_tx;
     u_int8_t n_subcarriers;
     int8_t rssi_ctl;
     int8_t rssi_ext;
     s8 noise;
};
```

csi_len contains the length of the CSI data in bytes. *n_rx* and *n_tx* are the number of receive and transmit antennas, respectively. *n_subcarriers* is the number of OFDM sub-carriers. *rssi_ctl* holds RSSI for the primary channel, and for 40 Mhz channels, the secondary channel's RSSI is stored in *rssi_ext*. The received CSI data are passed up the stack with the data payload, and the *struct ieee80211_csi* entries are filled accordingly. When the *mac80211* sub-layer receives an *sk_buff* with fields indicating that CSI data is present it performs two actions. First, it splits data contained in the *sk_buff* at the CSI data boundary and the user data payload is forwarded up the stack as per default behaviour. Second, the CSI data is handed to the AARC module for further processing. When AARC receives CSI data, the first action it performs is to find the association ID (*aid*) of the Wi-Fi client that sent the data frame (using the MAC address). The *aid* and the CSI data tuple are inserted into the CSI queue managed by the AARC module. A kernel thread (tasklet) wakes up and processes the CSI data by sending each CSI to the NETLINK layer.

In a full implementation, we envision that eSNR feedback will be transmitted to the data sender via BlockAck control frames. However, as our hardware does not allow us to modify BlockAck content, we implement the feedback channel in CSID using UDP via an entirely separate Wi-Fi channel, using a second Wi-Fi NIC. We discuss the implications of this simplification in Section 5.8.4. The AARC module also handles reception of the CSI feedback sent from the receiving host via the NETLINK layer. The AARC module registers with the NETLINK layer to receive CSI feedback. Once the feedback is received, the *aid* is used to find the Wi-Fi client and the data for the Wi-Fi client and AARC uses the feedback as described in Section 5.6.

It is worth noting that while validating the ath9k-csitool we encountered a performance problem where goodput achieved was very low (i.e., 20 Mbps). After investigation we noticed that when enabling the CSI tool and the transmitted data is not 8-byte aligned, the receiver would drop frames. This behaviour only occurred when CSI reporting was enabled. We work around this bug in the mac80211 layer's transmit path: we examine the transmitted data length and if it is not multiple of 8-bytes, we pad the data. We informed the ath9k-csitool authors and made our fix available.

5.8.2 AARC

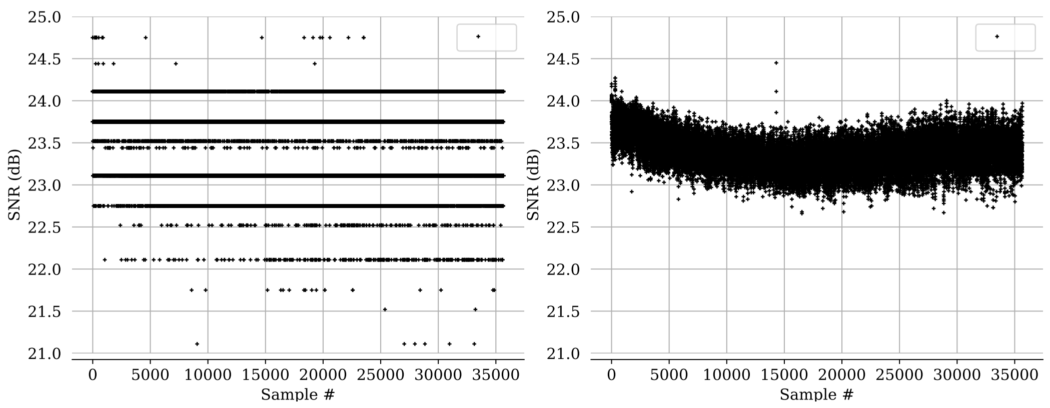
The mac80211 subsystem in Linux is modular, so adding an RC algorithm requires implementing the necessary functions that are called by the device driver. The two main functions are the transmit status report function, called when the device driver is notified about the status of a transmission, and the the function that sets the transmit rate, called by the device driver when preparing a transmission. Upon transmit completion the device driver informs AARC of the outcome of the transmission. It reports the size of the A-MPDU, the number of frames that failed in the A-MPDU, the number of transmission retries, and the MCS used to transmit the frame. For each MCS, AARC keeps track of the number of frames transmitted and number of frames that failed and uses this information to calculate success rate. In addition, AARC tracks the A-MPDU sizes for each MCS using an EWMA. The AARC module also registers with the NETLINK layer to receive CSI feedback. This received feedback is the eSNR calculated by the receiver.

5.8.3 CSID

CSID is a multi-threaded user-level application written in C++ and C. It uses the low-level library (*libcsi*), written in C, that handles NETLINK communication between the CSID application and the Linux kernel. CSID has three main components. The first component handles NETLINK communication using *libcsi*. The second component sends and receives CSI feedback using UDP transport. The final component manages the main buffer where CSI data are stored. All the components use an asynchronous API from the boost library [79]. CSID operates in two modes: AP mode and Wi-Fi client mode. When in AP mode, CSID maintains a list of all the Wi-Fi clients and uses the *aid* received from NETLINK to uniquely identify each Wi-Fi client. CSID sends and receives CSI feedback and as such needs IP addresses to communicate. When running in Wi-Fi client mode, the IP address of the AP is hard-coded, while in the AP mode, the IP addresses of Wi-Fi clients are learned from the received CSI feedback. CSID uses sockets to receive raw CSI data from the device driver. Upon receiving the CSI, CSID processes it as described in Section 5.6.2 and calculates eSNR (see Section 5.6.3). The calculated eSNR is sent back to the transmitter via the feedback channel using UDP.

5.8.4 Practical Limitations

eSNR Feedback would ideally be sent as part of the BlockAck control frame, however we cannot modify BlockAck control frames using our platform. Therefore, we instead send AARC feedback using UDP frames on a feedback channel. As mentioned earlier, in our implementation we used a separate Wi-Fi card running on a different Wi-Fi channel to transmit feedback information. Sending feedback via BlockAck control frames would incur extra overhead and our results would see a slight decrease in goodput. Sending eSNR feedback would require 9 bits per spatial stream to be added to the BlockAck control frame. We use a fixed-point encoding to encode eSNR with a scaling factor of $\frac{1}{100}$, yielding a maximum eSNR value of 51.2 dB. BlockAck control frames are sent at the highest basic rate that is less then or equal to the sending rate. In our setup BlockAck control frames are sent at 24 Mbps, resulting in 0.375μs overhead added (or 1.125μs when using three spatial streams). Using utilisation Equation 3.1 from Chapter 3 and accounting for extra BlockAck overhead, the total reduction in utilisation is 0.04%, a negligible overhead. It is worth mentioning that the IEEE 802.11n standard defines procedures for exchanging CSI feedback designed for transmit beam-forming [50]. However, that scheme exchanges multiple messages, adding unnecessary overhead for the purposes of AARC.



(a) SNR reported by the hardware. **(b)** EWMA of the SNR.

Figure 5.12: Raw mean SNR of the primary channel and the secondary channel and its EWMA

.

SNR reported by the hardware varies by 1 dB even on stable channels. We don't exactly know what causes this variation, but we speculate it to be either AGC imbalance, or the result of integer quantisation. To smooth the SNR reported by the hardware, we use an EWMA. Figure 5.12 shows the SNR reported by the hardware, and the EWMA version of it. Note that the SNR shown in Figure 5.12 is the mean of the two SNR values reported by the hardware for the two 20 MHz "halves" of the 40 MHz channel used.

Transmit Power was limited to 14dBm, as we noticed that when transmitting with full power (20dBM) the received power was clipped, resulting in noisy SNR and CSI measurements.

5.9 Evaluation

We evaluate the performance of AARC empirically, both synthetically and on a Wi-Fi testbed we set up on 7th floor of the UCL CS building, shown in Figure 5.13. There are a few questions we'd like to answer in our experiments. The

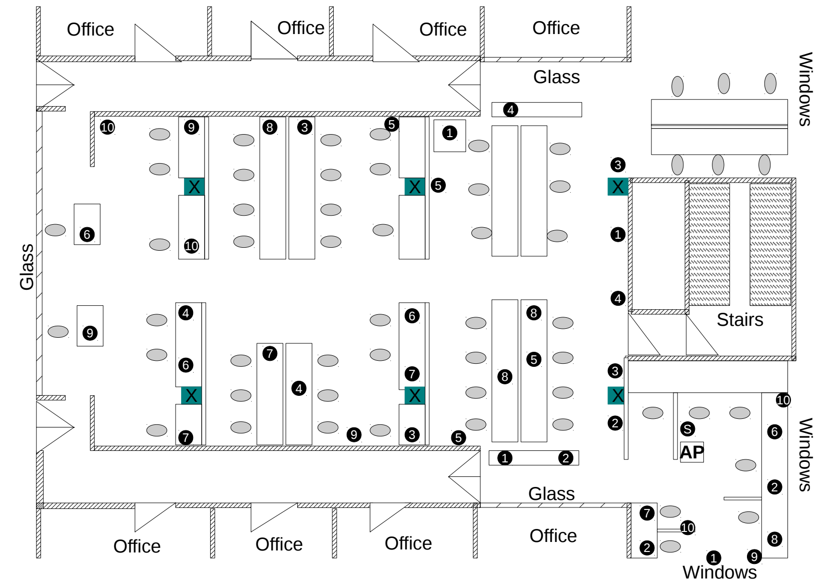


Figure 5.13: The indoor office environment and wireless topologies used to evaluate AARC. The access point is marked as **AP**. The Wi-Fi client marked as **S** is the desktop Wi-Fi client that remains in the same location in all experiments. Numbers on each Wi-Fi client indicate the topology used during the evaluation.

first question we address is whether our RC algorithms are practical. Second, we set the goal to create a RC algorithm that manages to send large A-MPDUs. How well does AARC achieve this goal? Note that in a real environment, when the dynamics of the channel change, the greatest A-MPDU size is largely determined by the transmit rate supported by the wireless channel and the frame loss rate. If AARC achieves greater A-MPDU sizes as compared to other schemes, does it also improve goodput? More specifically, is the aggregated achieved goodput greater as compared with other schemes? In Chapter 4 we showed that AAQ is fair to Wi-Fi clients when the transmit rate for all Wi-Fi clients is the same. Does AAQ remain fair, however, in a Wi-Fi network that uses diverse transmit rates? Finally, does AARC perform well in a mobile usage scenario?

5.9.1 Setup

To run AARC, Wi-Fi stations must be equipped with Wi-Fi network interface cards (NICs) that are capable of reporting CSI. In our experimental testbed we have six Wi-Fi NICs that support CSI. Therefore, we evaluate AAQ using five Wi-Fi clients and a Wi-Fi AP. Our AP runs on a desktop machine with an Intel i7 CPU and 12 GB RAM, equipped with a PCIe Atheros 9300 NIC. The Wi-Fi client marked with letter the **S** in Figure 5.13 is a similar desktop machine that runs one of the Wi-Fi clients, and its location does not change in different topologies. The other four Wi-Fi clients run on Intel NUC devices equipped with mini-PCIe Atheros 9300 Wi-Fi NICs, and are placed on various locations marked in Figure 5.13 for various topology configurations.

5.9.2 Methodology

We use the same setup as in Chapter 4, Section 4.8.3 and Figure 4.12. The AP and the traffic generator are connected using a 1 Gbps Ethernet segment. Wi-Fi clients associate to the AP and the traffic generator sends to Wi-Fi clients using *iperf* for 120 seconds. The AP operates on a 5 GHz band (channel 154) and uses 40 MHz bandwidth with the SGI enabled. We evaluate four different schemes: Minstrel HT, AASRC, and AARC with all using AAQ and TAO, and Minstrel HT with FIFO queues (the default configuration in Linux). When running AARC, the feedback channel runs over a separate Wi-Fi network that operates in the 2.4 GHz band.

An ideal evaluation testbed for RC algorithms will capture the dynamics of the wireless channel, test the internal mechanisms of the RC algorithm and at the same time be repeatable. While the first two properties can be achieved using a real testbed, achieving repeatability is challenging if not impossible. Still, we want an evaluation that is repeatable, as we must make sure that the RC algorithms are tested under the same wireless channel conditions. To achieve repeatability the research community has relied on using simulations. While simulations give repeatability, they fall short in capturing the real dynamics of the wireless channel and the interaction among various components in the systems. To have a better representation of the wireless channel, simulations are complemented using recorded traces. These traces record statistics from a real wireless channel and then the simulator is run using statistics from the traces. This trace-driven approach improves upon pure simulation-based approaches because frame losses are from a real wireless channel.

In attempting to capture the properties mentioned above, we use various evaluation methods to compare RC algorithms. We first use a synthetic approach, where we vary transmit power using a random walk that emulates rapid wireless channel changes. As AARC is a closed-loop algorithm, using this approach instead of a trace based approach allows us to account for the behaviour of the close-looped functionality. Using the synthetic approach, we compare RC algorithms under the same wireless channel conditions. Thus we produce an evaluation with repeatability, and to some degree, wireless channel variability. Secondly, we evaluate RC algorithms using the real indoor Wi-Fi network testbed shown in Figure 5.13. The advantage of this approach is that it captures the real dynamics of the wireless channel in an open plan office floor. However, as real testbed, it sacrifices repeatability. Finally, we use an indoor mobile scenario to measure the performance of AARC under mobile conditions. Channel fading is the dominant phenomenon under mobile conditions, and our synthetic methods fail to capture it. When running our synthetic evaluation we only change the transmit power and the underlying channel remains the same.

When running testbed experiments, we place four Wi-Fi clients at various locations shown in Figure 5.13, while the Wi-Fi client marked **S** remains static. For each physical placement of Wi-Fi clients we run experiments for all schemes where four Wi-Fi clients are active at the same time, resulting in five sub-topologies for each physical placement. In total, we run experiments on fifty (50) different topologies with four Wi-Fi clients active at the same time. We randomise the order in which the RC algorithms run on a topology and repeat each experiment four times, where each experiment runs for 120 seconds.

5.9.3 Synthetic Experiment Results

First we present results from our synthetic experiments. The goal of this evaluation is to compare RC algorithms on the same rapidly changing wireless channel. Using this methods allows us to also capture the dynamics of the CSI feedback loop. In these experiments we use a single Wi-Fi client and we change the transmit power at the AP during the course of the experiment. To change the transmit power we use a random walk. The AP transmits at a particular power for a duration of time (y-axis) before changing the transmit power again. We use the same random seed for all RC algorithms to ensure that the transmit power change is the same across all the RC algorithms compared. Each experiment runs for 60 seconds and is repeated 10 times. When plotting the data we show the mean goodput achieved across all one-second samples. We compare Minstrel HT, AASRC and AARC, with AAQ and TAO enabled for all RC schemes. The heatmap of goodput results is shown in Figure 5.14, with the x-axis showing RC algorithm and the y-axis the time delay between transmit power changes. Under these emulated conditions, AARC outperforms other RC schemes, with up to 12% improvement over Minstrel HT. These experiments served as a preliminary feasibility test for our RC algorithms. Next we show results from our experiments conducted in the indoor Wi-Fi network testbed shown in Figure 5.13.

5.9.4 Testbed Experiment Results

In this section we show results from our testbed Wi-Fi network experiments in which Wi-Fi clients remain static at the locations noted in Figure 5.13. Each Wi-Fi client is placed at desk height to emulate a typical office scenario where Wi-Fi clients remain static. We first present a high-level view of the goodput distribution for each RC algorithm across all topologies. Figure 5.15 shows the distribution of one-second goodput values each Wi-Fi client experiences on all topologies. To generate the figure we plot the empirical CDF (ECDF) of one-second goodput samples Wi-Fi clients experience across all topologies and runs. More then 70% of the time Wi-Fi clients get better goodput when

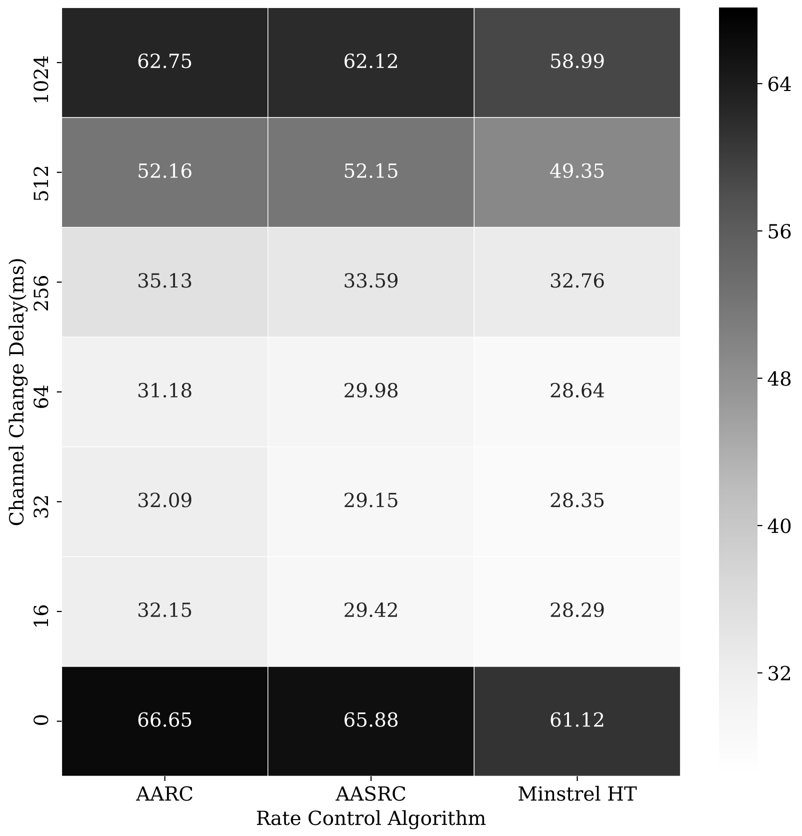


Figure 5.14: Heatmap of the goodput (Mbps) achieved by different rate control algorithm when transmit power varied by random walks with different time step duration.

using AARC. This suggests that for the majority of the time, in our testbed, choosing AARC is beneficial. Next, in Figure 5.16 we show the ECDF of the aggregated mean goodput each RC algorithm achieves. To generate the figure we calculate the mean goodput (among 80 one-second samples from the middle of the experimental run) each Wi-Fi client experiences and sum the values to obtain the aggregate goodput for a topology and a run. We then use these aggregated goodput data points for each run and topology to generate Figure 5.16. Most of the time AARC achieves higher aggregate mean goodput. The median goodput when using AARC is 6%, 10%, and 20% better then AASRC, Minstrel HT + AAQ + TAO, and Minstrel HT + FIFO, respectively. Table 5.3 shows 1st, 25th, 50th, 75th, and 99th percentiles for each of the RC algorithms. Figures 5.15 and 5.16 suggest that Wi-Fi clients benefit from using AARC. However, it does not show how each RC algorithm performs with

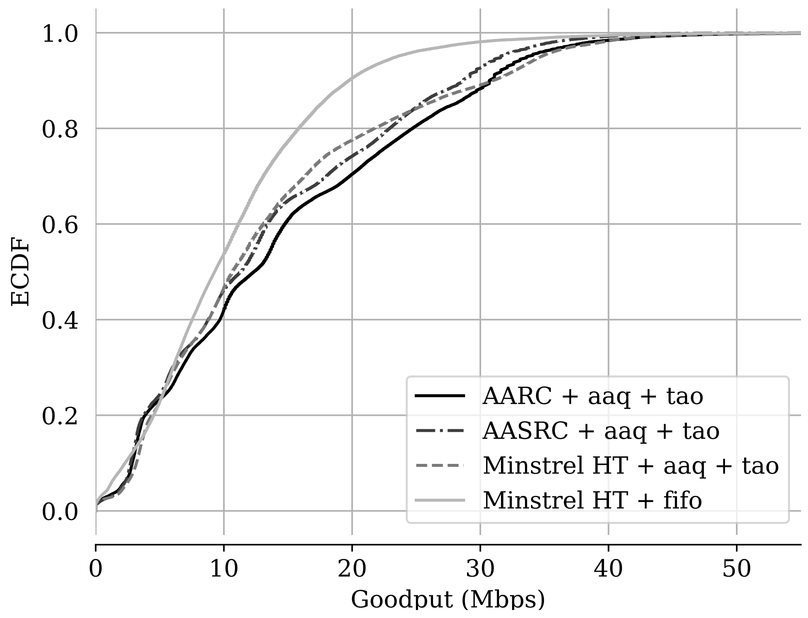


Figure 5.15: One second goodput (Mbps) distribution across all topologies, runs and Wi-Fi clients.

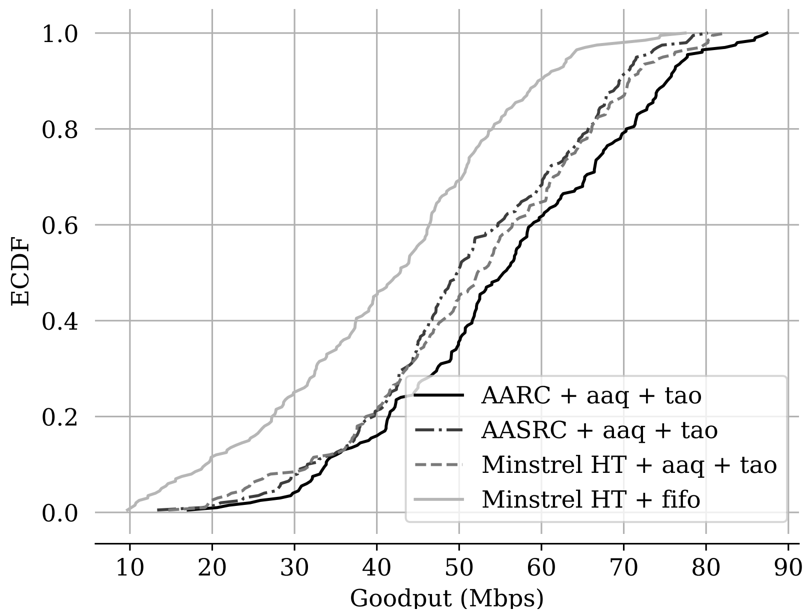


Figure 5.16: Aggregated goodput (Mbps) ECDF across all topologies.

RC	1st	25th	50th	75th	99th
AARC + aaq + tao	26.0	45.4	54.0	68.7	83.0
AASRC + aaq + tao	23.0	42.6	49.0	63.2	75.5
Minstrel HT + aaq + tao	21.0	42.7	51.1	63.8	77.6
Minstrel HT + fifo	19.0	31.1	42.0	49.2	69.7

Table 5.3: Aggregate goodput (Mbps) percentiles across all topologies

respect to the others on each topology. Next we explore a coherent comparison between RC algorithms, and show the benefits each algorithm offers, and under which circumstances.

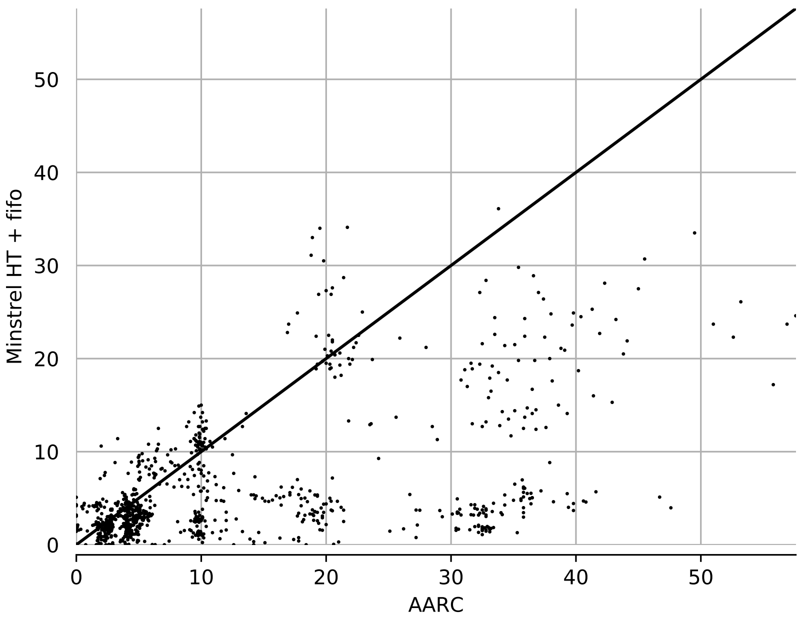


Figure 5.17: Scatter plot of AARC v Minstrel HT + FIFO for Topology 5.

Before we present these results we need to restate the challenge that comes when comparing RC algorithms using a real wireless network testbed. As mentioned earlier, ideally a comparison of RC algorithms would be realised under identical wireless channel conditions. However, arranging identical wireless channel conditions in a real wireless network testbed is hard, if not impossible. To mitigate this difficulty we repeat each experiment over multiple runs and randomise the order in which schemes execute. Given the random nature of the wireless channel, we do not expect AARC to outperform other RC algorithms for every one-second sample. For example, in Figure 5.17 we show a scatter plot of AARC vs Minstrel HT + FIFO for one-second goodput samples on one of the topologies. As can be seen in Figure 5.17 there are samples where Minstrel HT + FIFO achieves higher goodput. However, for the majority of data points, AARC performs better. When we evaluate RC algorithms we will calculate average goodput and compare RC algorithms using aggregate mean goodput achieved on each topology. To calculate the mean goodput for a topology we first calculate the mean goodput each Wi-Fi client achieves across all four runs. That is, we have 80 seconds worth of goodput samples for each run, with a total of 320 seconds of goodput data across all four runs. Then the aggregated average goodput for a topology is calculated by summing the mean goodputs of all Wi-Fi clients active in a topology.

First we look at the performance gains of AAQ and TAO using Minstrel HT as compared to Minstrel HT when running with FIFO queues. This will show the benefit of just running AAQ and TAO. We then continue the comparison by adding new techniques we developed, and compare the performance to the previous best system. This way we will be able to see the benefit each technique adds. Figure 5.18 shows a scatter plot comparing Minstrel HT + FIFO (x-axis)

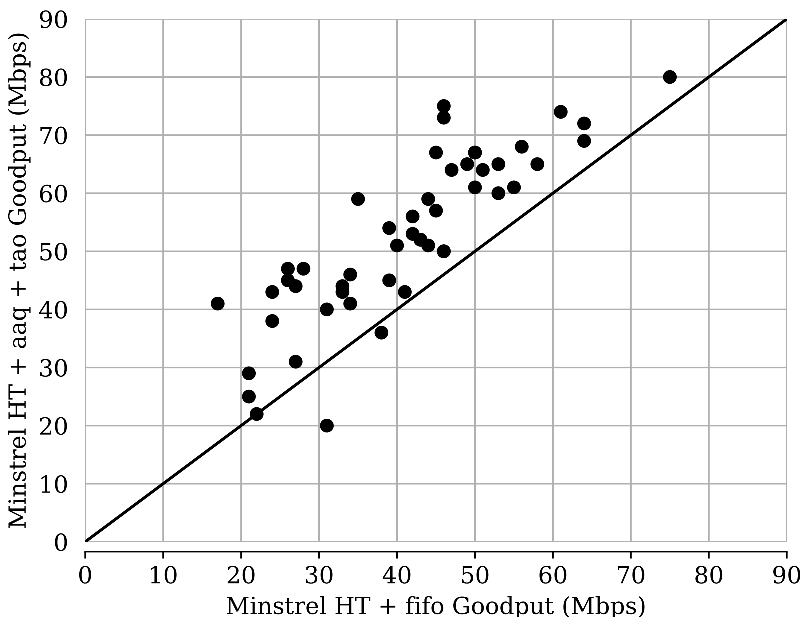


Figure 5.18: Aggregate goodput comparison of Minstrel HT + fifo and Minstrel HT + AAQ + TAO for all topologies.

with Minstrel HT + AAQ + TAO (y-axis). Each data point represents the aggregate mean goodput of a topology with fifty data points in total, one for each topology. Minstrel HT gains are significant when enabling AAQ and TAO, outperforming Minstrel HT + FIFO in 94% of the topologies with up to 230% goodput improvement and 30% goodput improvement on average. Given that the Minstrel HT algorithm is the same for both schemes, these improvements in goodput are the result of AAQ and TAO sending larger A-MPDUs. To show this we plot the ECDF of the A-MPDU sizes each scheme achieves across all topologies. In Figure 5.19 we see that enabling AAQ and TAO results in

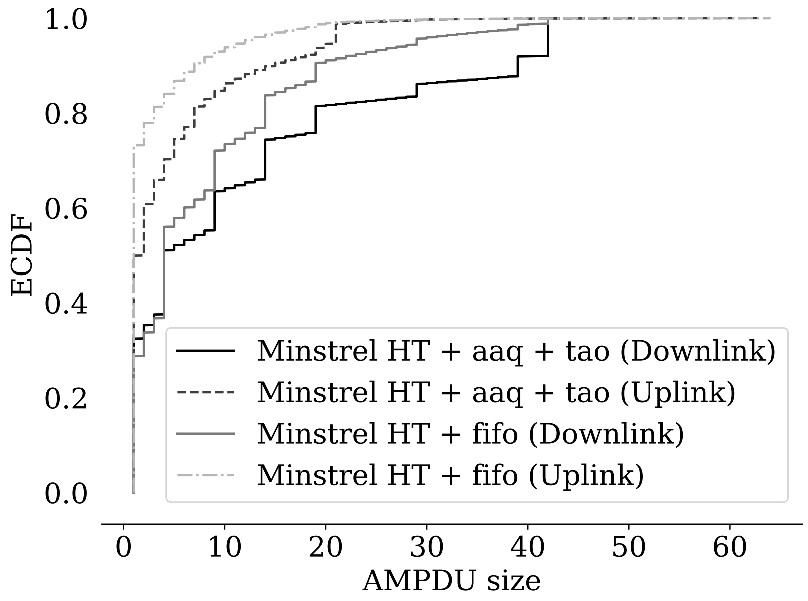


Figure 5.19: Distribution of A-MPDU size when the AP is transmitting (Downlink) to Wi-Fi clients and when the Wi-Fi clients transmit to the AP (Uplink) across all topologies.

Minstrel HT sending larger A-MPDUs. However, note that for both schemes there is still a large fraction of single-frame transmissions as a result of the probing frames used by Minstrel HT.

Next, we compare AASRC + AAQ + TAO and Minstrel HT + AAQ + TAO. We want to see if simply using AASRC gives performance improvements. In Figure 5.20 we show a similar scatter plot as in Figure 5.18. We see that both schemes achieve similar performance, with AASRC achieving an aggregate mean goodput of 51.3 Mbps across all topologies, and Minstrel HT achieving 52.6 Mbps. In Figure 5.22 we show the distribution of number of frames in an A-MPDU and in Figure 5.23 we show the number of bad frames in an A-MPDU across all topologies. AASRC + AAQ + TAO sends larger A-MPDUs. But, this is not reflected in the goodput achieved. There are two causinges, as we explained in Section 5.5. First, AASRC incurs more bad frames in an A-MPDU (Figure 5.23) as a result of probing with a whole A-MPDU, and

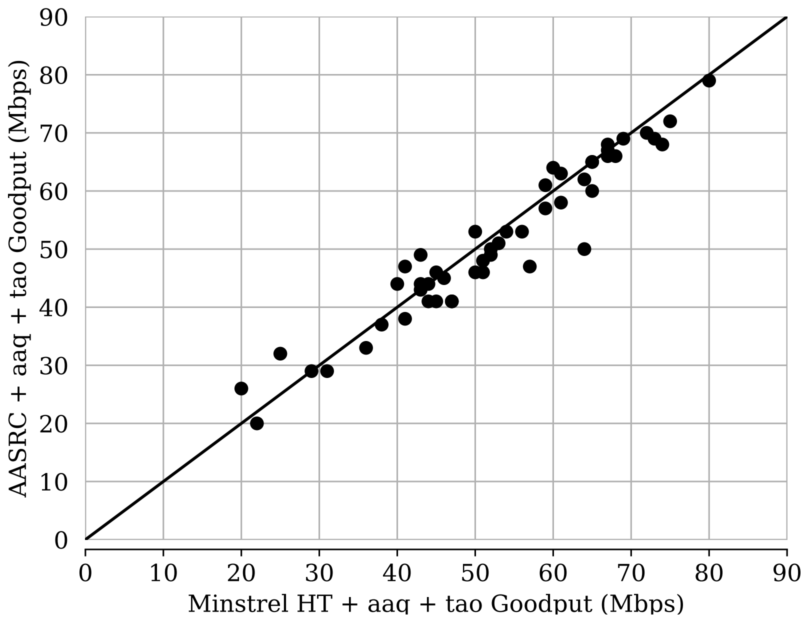


Figure 5.20: Aggregate goodput comparison of Minstrel HT + AAQ + TAO and AASRC + AAQ + TAO for all topologies.

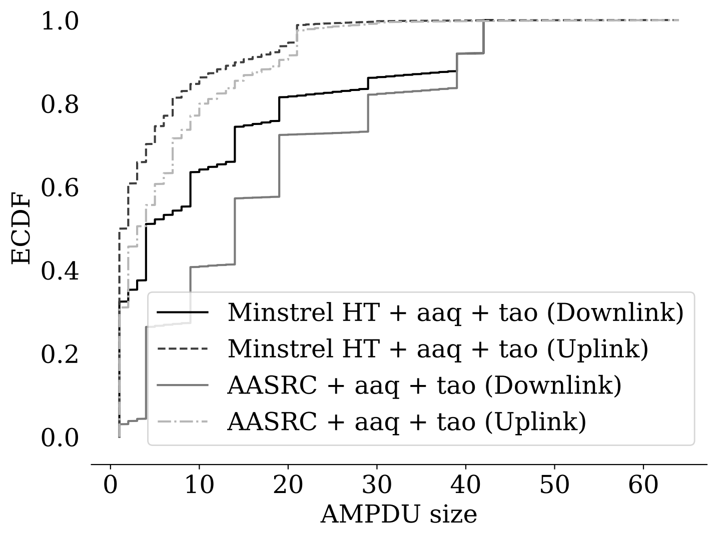


Figure 5.21: A-MPDU size

Figure 5.22: Distribution of A-MPDU size when the AP is transmitting (Downlink) to Wi-Fi clients and when the Wi-Fi clients transmit to the AP (Uplink) across all topologies.

second, it does not promptly react to channel changes. In Figure 5.25 we show the distribution of MCSes used across all topologies and note that Minstrel HT uses MCS-6 3% more often then AASRC. As discussed earlier, this drawback

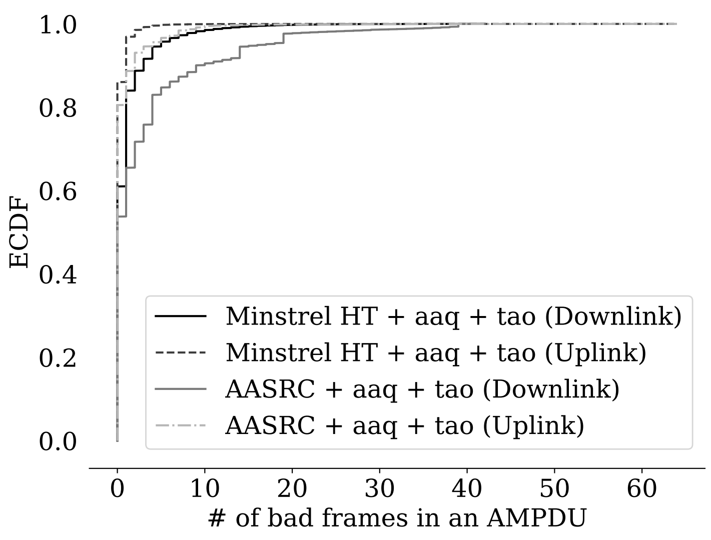


Figure 5.23: Bad frames in a A-MPDU

Figure 5.24: Distribution number of bad frames in a A-MPDU when the AP is transmitting (Downlink) to Wi-Fi clients and when the Wi-Fi clients transmit to the AP (Uplink) across all topologies.

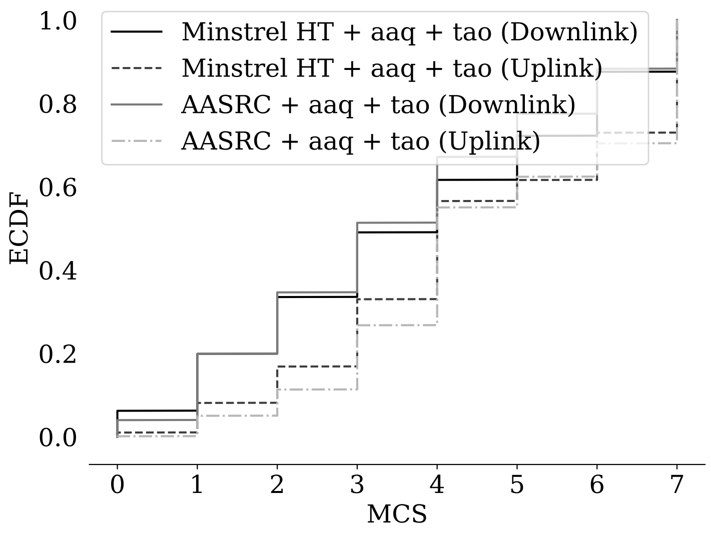


Figure 5.25: MCS used by Minstrel HT + AAQ + TAO and AASRC + AAQ + TAO on all topologies.

with AASRC motivated us to develop AARC. Next we compare AARC and AASRC. In Figure 5.26 we show a scatter plot comparison between the two. In 90% of topologies AARC performs better than AASRC. AARC gives up to

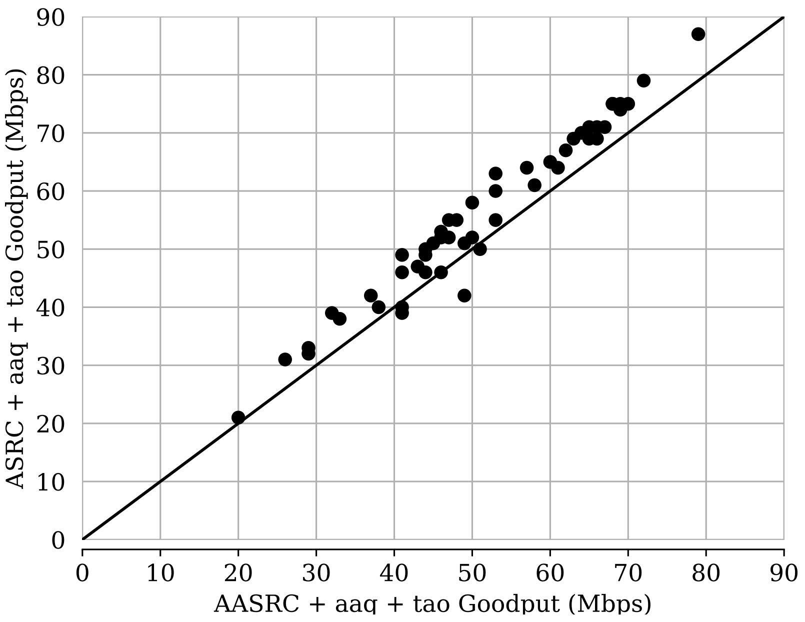


Figure 5.26: Aggregate goodput comparison between AASRC + AAQ + TAO and AARC + AAQ + TAO for all topologies.

21% aggregate mean goodput improvement with 8% improvement on average.

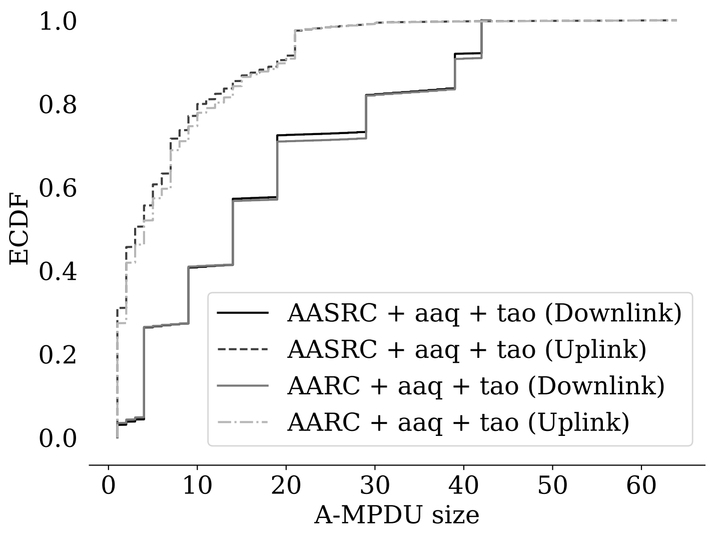


Figure 5.27: Distribution of A-MPDU size when the AP is transmitting (Downlink) to Wi-Fi clients and when the Wi-Fi clients transmit to the AP (Uplink) across all topologies.

Figure 5.27 shows that both AASRC and AARC generate the same A-MPDU sizes. However, in Figure 5.28 we see that AARC produces fewer bad frames in an A-MPDU. The reason is that AARC probes less often then AASRC, as

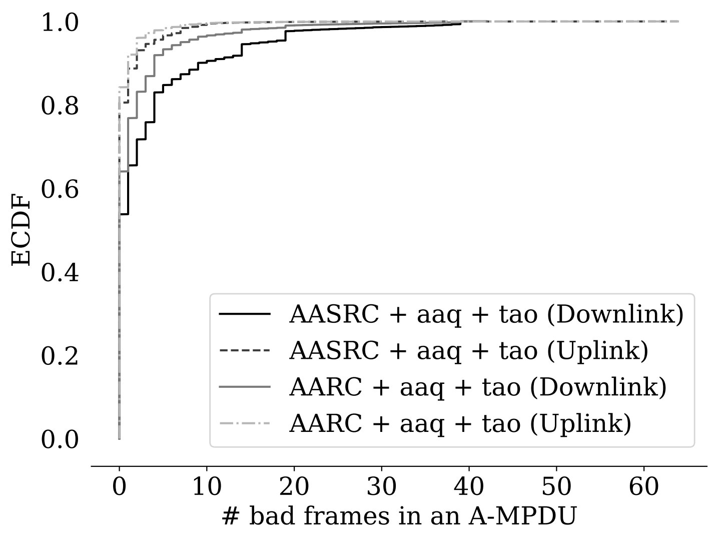


Figure 5.28: Distribution of bad frames in a A-MPDU when the AP is transmitting (Downlink) to Wi-Fi clients and when the Wi-Fi clients transmit to the AP (Uplink) across all topologies.

AARC uses a greater maximum backoff parameter. In Figure 5.29 we can see that AARC uses higher MCSs for some fraction of time more than AASRC. The reason is that AARC uses CSI to detect channel improvements, and cancel backoff early. We looked at the number of probes that AARC and AASRC

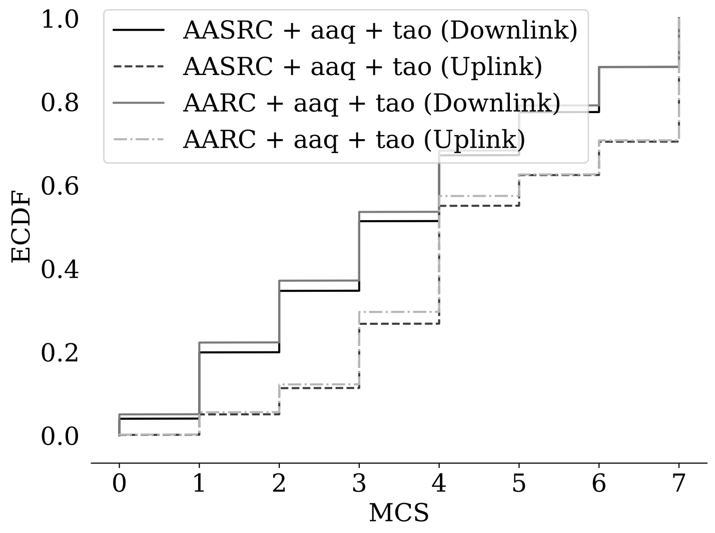


Figure 5.29: Distribution of MCS used by AASRC + aaq + tao and AARC + aaq + tao on all topologies

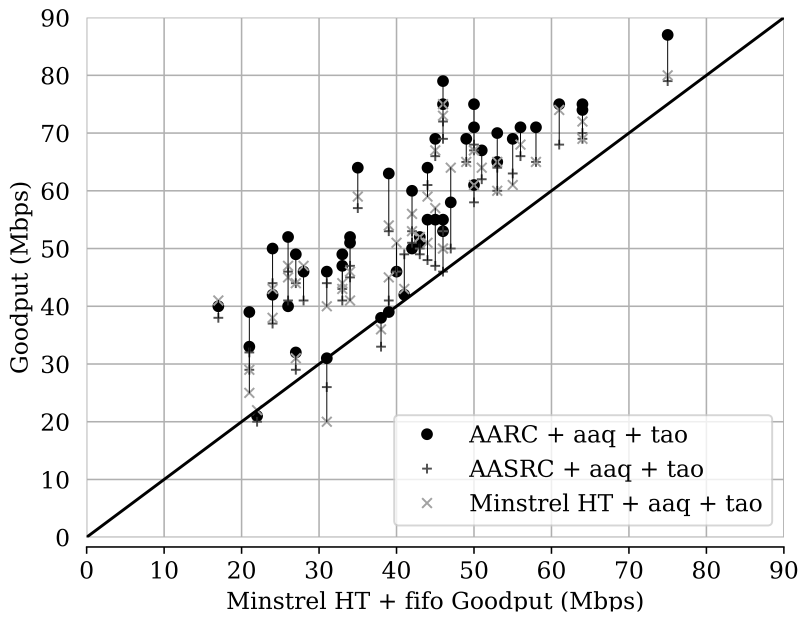
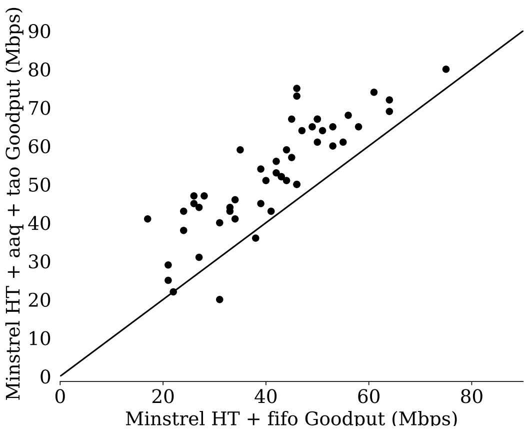


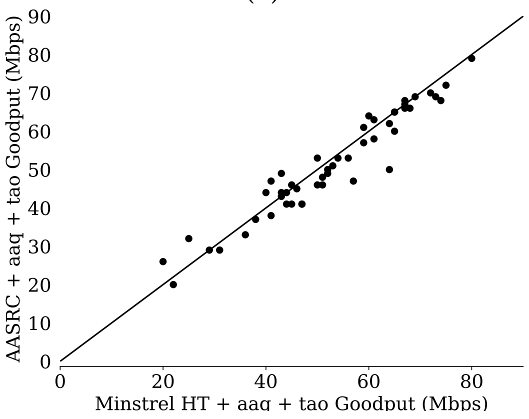
Figure 5.30: Aggregate goodput comparison of Minstrel HT + fifo and other schems for all topologies

send and note that across all topologies, on average, for AARC about 6.6% of transmitted frames were probing frames, out of which 7.1% were forced probes. Out of forced probes, 37% were successful (probing that resulted in the new rate being used). On the other hand, AASRC sent about 24% of frames as probes and 4% were successful.

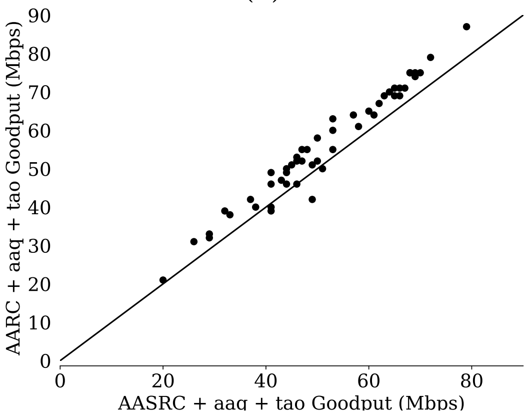
Finally, in Figure 5.30 we show a scatter plot comparing each of the schemes to Minstrel HT + FIFO. The vertical lines connect the points that belong to the same topology. AARC outperforms Minstrel HT + FIFO in 92% of topologies with up to 2.3x improvement in goodput. Compared to Minstrel HT + AAQ + TAO, AARC performs better on 72% topologies with up to 56% improvement and 12% improvement on average. In 6% of topologies AARC and Minstrel HT + AAQ + TAO have equal performance and in 12% of topologies AARC performs worse then Minstrel HT + AAQ + TAO, with 6% lower goodput on average. Lastly, to give a more intuitive comparison, we show all the scatter plots next to each other in Figure 5.31.



(a)



(b)



(c)

Figure 5.31: Per topology goodput comparison: Minstrel HT + FIFO vs Minstrel HT + TAO + AAQ (Figure 5.31a), Minstrel HT + AAQ + TAO vs AASRC + AAQ + TAO (Figure 5.31b) and AASRC + AAQ + TAO vs AARC + AAQ + TAO (Figure 5.31c)

5.9.5 Goodput Fairness

So far we have compared RC algorithms by mean aggregate goodput. We have not yet discussed the goodput each Wi-Fi client sees and if Wi-Fi clients get fair share of available goodput. We note that in some topologies some Wi-Fi clients achieve higher goodput when running Minstrel HT + FIFO. For example, in Figure 5.32 we show aggregate goodput for topology 13. We chose topology 13 for illustration purposes only; the same behaviour is observed in other topologies when this phenomenon occurs. When running Minstrel HT +

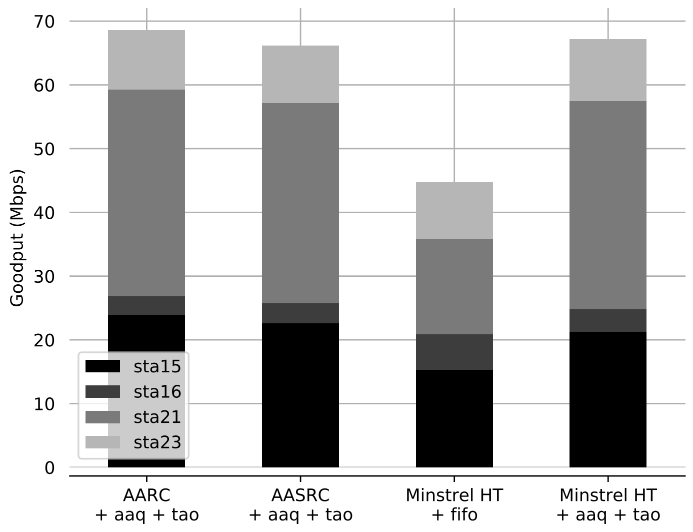
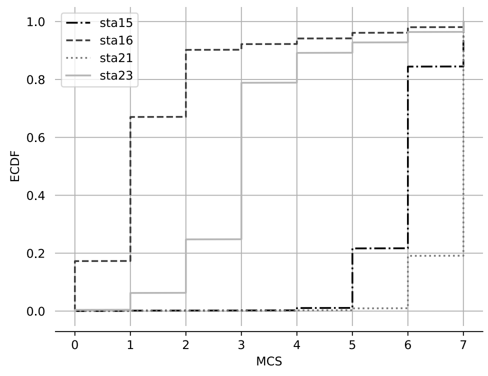
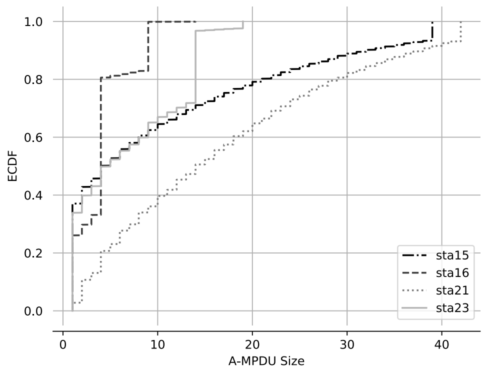


Figure 5.32: Average goodput each Wi-Fi client gets on topology 13 (topology 3, subtopology 3).

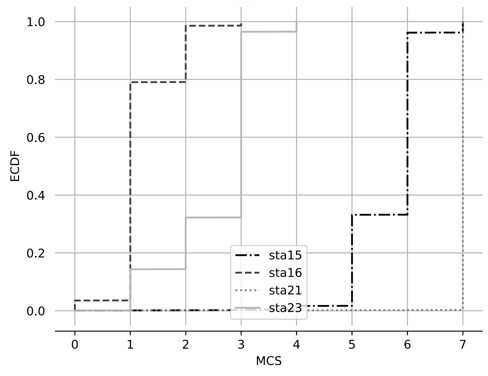
FIFO, the Wi-Fi client labeled sta16 achieves higher goodput as compared to when running with other schemes. In addition, Wi-Fi clients sta21 and sta15 see significantly lower goodput. From the point of view of aggregate goodput, RC algorithms running AAQ and TAO achieve significantly higher goodput. But, could it be that AAQ and TAO suffer from fairness problems when run under a RC algorithm? To analyse this behaviour we compare the distribution of MCSes used and A-MPDU sizes for each Wi-Fi client when running Minstrel HT + FIFO and AARC + AAQ + TAO. In Figure 5.33a and Figure 5.33c we see that the AP mostly uses MCS-1 when transmitting to Wi-Fi client sta16. The other three Wi-Fi clients use higher transmit rates, tough MCS-3, MCS-6, and MCS-7 most of the time. In Wi-Fi networks with transmit rate diversity, the over-the-air data transmission time of a Wi-Fi client that transmits slowly increases for the same amount of data, as compared to clients that transmit at higher rates. This leads to fairness problem where a slowly sending Wi-Fi client takes a larger fraction of wireless channel time. This is a well known problem [83]. To overcome it the Wi-Fi MAC protocol introduced time-based fairness through which the amount of time each Wi-Fi client can transmit with a single wireless channel acquisition is limited (4ms in our testbed). But despite the Wi-Fi MAC protocol's use of time-based fairness, why do we still observe unfairness? The problem is the interaction between A-MPDU size and the scheduling of A-MPDUs by the device driver. Recall that in our setup we use the ath9k device driver which schedules Wi-Fi clients for transmission using a round-robin scheduling discipline. In the case of Minstrel HT + FIFO, the



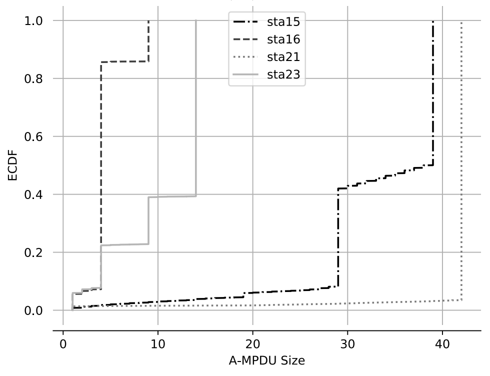
(a)



(b)



(c)



(d)

Figure 5.33: Distribution of MCS and A-MPDU size when the AP is transmitting (Downlink) to Wi-Fi clients when using Minstrel HT + FIFO (5.33a and 5.33b, respectively) and AARC + AAQ + TAO (5.33c and 5.33d, repsectively) on topology 13.

quickly sending Wi-Fi clients produce low A-MPDU sizes (see Figure 5.33b), which leads to these Wi-Fi clients' not fully utilising their share of transmit time. In turn slowly sending Wi-Fi clients obtain more than their fair share of transmit time.

To illustrate this problem, suppose that the AP only transmits to Wi-Fi clients sta16 and sta21. Say it first schedules Wi-Fi client sta21, which uses MCS-7, and it sends A-MPDUs with size 17 on average. The over-the-air time for sta21's transmission is about 1.5ms (much less then the 4ms it can use to transmit). After that, the device driver schedules sta16 for transmission using MCS-1 and an A-MPDU size of 4 frames on average. The over-the-air time for sta16's transmission is about 3.2ms, more then twice that of sta21. In addition, because sta21 does not use all of its fair share of time, sta16 gets to be scheduled more often than it should. This interaction with round-robin scheduling in the device driver leads to loss of the time-based-fairness provided by the MAC protocol. This is another argument why analysing cross-layer interactions is important to improve end-to-end performance.

When using AARC + AAQ + TAO, time-based fairness is restored, because quickly sending Wi-Fi clients send larger A-MPDUs (see Figure 5.33d), increasing the fraction of their share of time they utilise. Recall that AAQ receives explicit stop messages from the device driver when the number of dequeued packets for a Wi-Fi client equals the maximum number of frames that can be formed into an A-MPDU. Because maximum A-MPDU size is limited to 4ms, AAQ essentially achieves an approximation of time-based-fairness. To verify this we measure the fraction of time each Wi-Fi client uses for transmission as shown in Figure 5.34. We can see that when using Minstrel HT + FIFO, the slowly sending Wi-Fi client sta16 uses the wireless channel for a larger fraction of time. While using AAQ and TAO does not completely eliminate unfairness, it certainly helps in improving time-based fairness. To completely uphold time-based fairness, the AP must transmit to all Wi-Fi clients with maximum A-MPDU size. However, on a wireless channel where frames are lost, this is not possible, as frame losses will lower A-MPDU sizes.

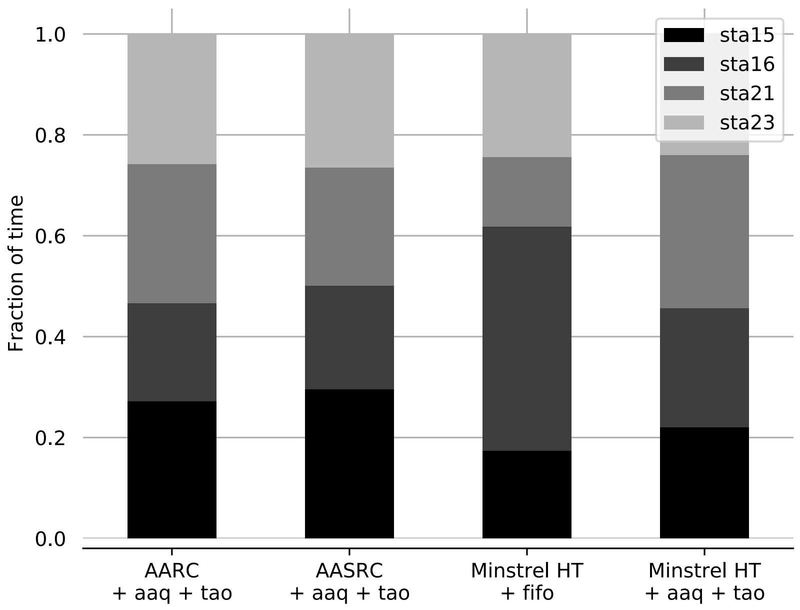


Figure 5.34: Fraction of transmit time each Wi-Fi client gets on topology 13.

5.9.6 Mobile Experiment Results

So far we have shown results from our synthetic experiments and from our real Wi-Fi network testbed. However, none of these experiments capture wireless channel fading. In synthetic experiments we compared the performance of RC algorithms on a dynamic channel by changing only the transmit power, which does not change the underlying wireless channel. In our real testbed experiments, Wi-Fi clients are static, so the wireless channel changes only when people move around. To better evaluate the performance of AARC under wireless channel fading, we measure it in a mobile scenario. Ensuring that the RC algorithms are run under same mobile scenario conditions is hard. One can have a mobile experiment run many times (i.e., a week worth of runs) while trying to follow the same mobility pattern, but doing so consistently is difficult. The approach we take is to compare AARC with a variant of AARC that uses SNR instead of eSNR to detect channel improvement. This way minimise changes in the algorithm, allowing us explore AARC's ability to react to rapid channel changes. In this experiment the AP forwards TCP segments by transmitting them to a single Wi-Fi client while it is moving along the predefined path shown in Figure 5.35. We use the same location as in Figure 5.13

with Figure 5.35 cropped to show only the area where the Wi-Fi client moves. When running the SNR variant, AARC uses SNR difference instead of eSNR

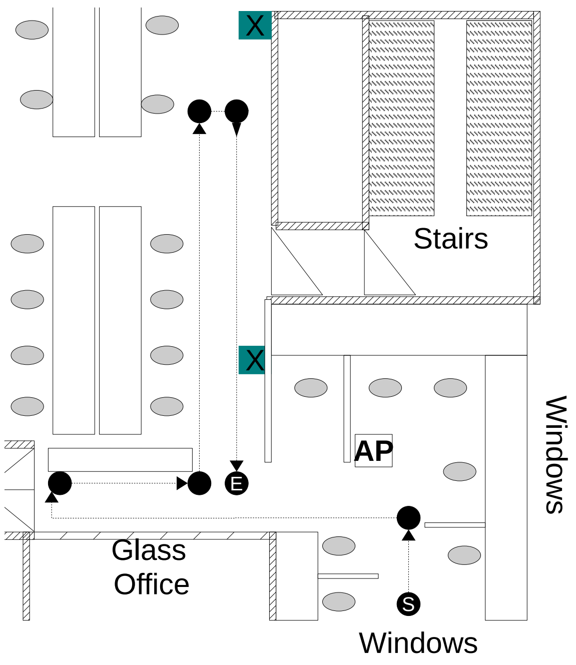


Figure 5.35: The walking path starting at the point marked **S** and ending at the point marked **E** for mobile experiments comparing eSNR and SNR.

difference, and the algorithm otherwise remains unchanged. The Wi-Fi client

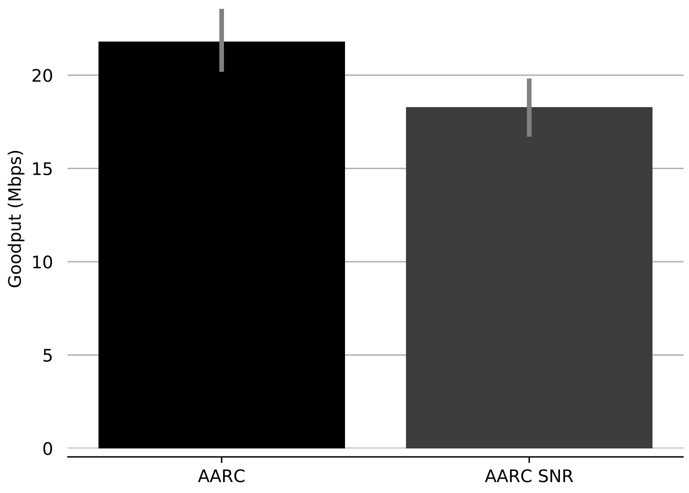


Figure 5.36: Mean goodput (with 95% confidence intervals error bars) AARC and AARC SNR achieve under a mobile scenario experiment.

is held in hand similarly to how a person holds a handheld device while typing and walking at normal speed. Each experiment starts at the location marked **S**, and the mobile device is moved along the path at walking speed until it reaches the location marked **E**. Each experiment is repeated 10 times and the order of the runs of variants is random. On average, each experiment runs for 40 seconds. AARC offers a mean goodput of 22 Mbps while AARC SNR achieves a mean goodput of 18 Mbps. AARC achieves a 20% higher goodput then AARC SNR. These experiments first show that AARC is able to adapt the transmit rate under rapidly changing wireless channels and second show the superiority of eSNR to SNR, as also shown previously [46]. Looking at the probe statistics, we note that for AARC 5.8% of the probes were forced probes were 86% of them resulted in success. For AARC SNR, 6.3% were forced probes and 72% resulted in success. AARC SNR triggers probing more often, which also results

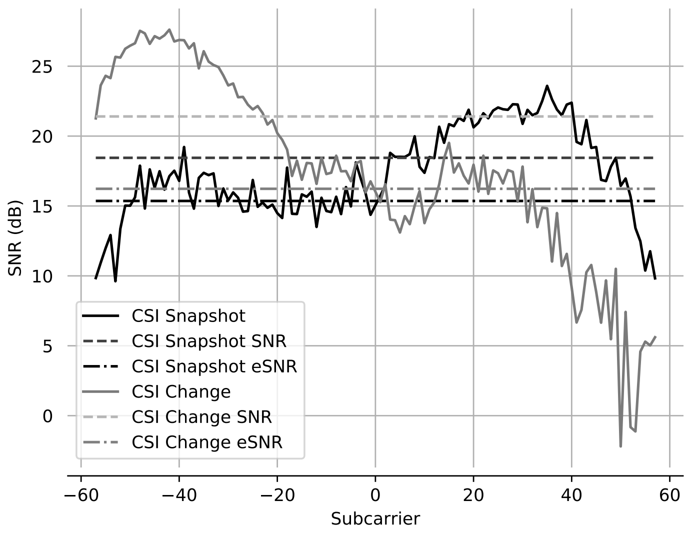


Figure 5.37: AARC SNR wrongly decides to probe in response to a wireless channel change.

in more failed probes. To understand why, consider an example (Figure 5.37) showing the CSI when AARC SNR takes a snapshot of the SNR (solid black line) and the CSI when AARC SNR decides that the wireless channel has improved (solid grey line). The black and grey dashed lines show the recorded

SNR for the respective CSI curves and the dotted black and grey lines show the eSNR recorded for the respective CSI, for comparison. From the CSI we can see that there has been a wireless channel change. This change resulted in improvement for lower frequency sub-carriers (the left-hand side subcarriers), while the higher frequency sub-carriers experienced a deep fade. As previously shown SNR is biased towards higher power sub-carriers [46], which in this example leads to a greate enough SNR change triggering an AARC SNR probe. However, because of deep fading the probe fails. For comparison, note that the eSNR change is not great enough to trigger a probe, so AARC would not probe. This leads to AARC SNR's triggering probes more often, resulting in more loss.

5.10 Discussion

In this chapter we presented shortcomings of the default rate control (RC) algorithm used in Linux, Minstrel HT. To explore new transmit rates Minstrel HT uses a probing mechanism. While this probing mechanism allows Minstrel HT to react quickly to losses it comes up short in sending large A-MPDUs. We developed AASRC and AARC RC algorithms that probe using full A-MPDUs thus, keeping the A-MPDU size large. Under loss, probing with full A-MPDUs is costly. AASRC and AARC use an exponential backoff mechanism that delays probing of transmit rates that experience high loss. When using AASRC, the transmitter makes decisions independently about which transmit rates to use, while AARC relies on the receiver's feedback to make transmit rate decisions. AASRC works well when the wireless channel rarely changes but the backoff mechanisms prevents it from reacting in time when the wireless channel improves. To overcome this problem with AASRC, we designed AARC, which uses channel state information (CSI) to detect wireless channel improvements. AARC uses effective SNR difference, measured at two different points in time, to detect channel improvements and trigger transmit rate probing. We evaluated the performance of AARC using a combination of synthetic wireless channel experiments, experiments on a real Wi-Fi network testbed, and a mobile Wi-Fi scenario. In our testbed AARC achieves up to 2x more aggregate mean goodput compared to the default Minstrel HT configuration in Linux. Our mobile experiments show that AARC is able to adapt to wireless channel changes in mobile scenarios.

To the best of our knowledge, AARC is the first practical rate control algorithm implementation that uses CSI. Nevertheless, as we discussed earlier, AARC as presented in this chapter used only one spatial stream. Today's Wi-Fi devices support multiple streams (MIMO), and for AARC to fully exploit that hardware, it must support multiple spatial streams. To make AARC work with MIMO, we need details of howw multiple streams are processed by the specific Wi-Fi card. We do not have those details for commodity cards.

While the math for calculating effective SNR and the techniques presented here work for MIMO streams, AARC's probing algorithm would need modifications for it to work using MIMO. Designing a probing algorithm for use in MIMO systems is left as future work. One approach is to use the same approach as in [72] when exploring spatial streams. Also, one can rely on Minstrel HT's approach to exploring spatial streams. Regarding the tracking of eSNR, AARC for MIMO would have to keep eSNR differences for each spatial stream, tough the exponential backoff design would remain unchanged.

As we noted we fixed the wireless channel to 40 MHz, and the guard interval to the short-guard-interval (SGI). By default, the AP will enable 40 MHz channel width if no neighbouring AP is heard on the secondary channel. Designing a channel width selection algorithm is out of scope for this thesis. The same goes with the guard interval. CSI can aid in designing a guard-interval algorithm, however. Using CSI one can calculate a power delay profile [96] and use it to decide on which guard interval to use. For example, if a large delay spread is observed, the long guard interval is used, otherwise the short guard interval can be used. We leave this as future work to explore.

Chapter 6

Conclusion and Future Work

When studying systems of networked computers, everyone is taught the advantages of layered design. A layered design promotes modularity, which offers several advantages, such as isolation of information and development independence. Each layer performs a specific task and provides a well defined interface to the layer it serves, sharing only necessary information. This allows for independent development of each layer without affecting other layers. But as we argue in this thesis, cross-layer interactions in Wi-Fi end-host stacks cause subtle problems that result in a performance penalty. We take a cross-layer approach in this thesis, where through empirical analysis and evaluation we show that available Wi-Fi capacity can be reclaimed by propagating information across layer boundaries. We then designed and implemented cross-layer techniques that enable the IEEE 802.11 MAC to improve channel utilisation and therefore goodput. Of course, the idea of cross-layer design is not new to this thesis: previously it has been identified that cross-layer hints provide performance benefits [78, 41]. The rest of this chapter is organised as follows: in Section 6.1 we discuss the cross-layer interactions we identified during the work in this thesis, and the techniques we developed that mitigate these interactions; in Section 6.3 we consider avenues for future work that explore promising directions we have not yet explored in this thesis; and we conclude this thesis in Section 6.2 with a discussion of our techniques and how they relate to other approaches introduced during the work in this thesis.

6.1 Cross-Layer Interactions in Wi-Fi Networks

In Section 1.1 we listed five hypotheses that show cross-layer interactions that might cause the IEEE 802.11 CSMA/CA MAC to achieve low channel utilisation. We found that cross-layer interactions impacting Wi-Fi performance occur across various layers of the networking stack. At the transport layer, protocols such as TCP interact with the Wi-Fi MAC in a way that causes unnecessary channel acquisition overheads. At the network layer, the IP protocol's queueing subsystem is oblivious to the frame aggregation used by the Wi-Fi MAC protocol, which causes small numbers of packets to be aggregated, significantly reducing achieved goodput in Wi-Fi networks. Finally, at the MAC layer, the interaction between the transmit rate control algorithm and frame aggregation by the Wi-Fi MAC can reduce achievable goodput.

At the transport layer we identify two problems when using TCP over Wi-Fi networks. First, TCP ACKs are treated as normal data by the Wi-Fi MAC protocol (hypothesis **H3**), and therefore go through the same channel access procedure, incurring overhead. Secondly, TCP ACKs are robust to loss and are not congestion-controlled by TCP, but as shown in Chapter 4, uncontrolled transmission of TCP ACKs over Wi-Fi networks can cause unnecessary channel acquisition overhead and can increase collisions (hypotheses **H4** and **H5**).

The first problem is addressed in Chapter 3, where we designed and implemented TCP/HACK, a system that eliminates most of the expensive medium acquisitions that TCP ACK packets require, improving TCP goodput for bulk downloads. TCP/HACK addresses hypotheses **H3** and **H5**.

We address the second problem in Chapter 4, where we presented TAO, a TCP optimisation technique that queues TCP ACKs generated by the receiving TCP stack until all TCP ACKs for an incoming data aggregate have been generated. Once all TCP ACKs have been generated, a large aggregate can be formed from them and transmitted. The reader may notice that TCP/HACK and TAO address the same problem: the interaction described in hypothesis

H5. We will compare these two approaches in Section 6.2.

At the network layer, the IP queueing subsystem is oblivious to modern Wi-Fi card's use of frame aggregation. In Chapter 4 we presented the design of AAQ and showed that by coordinating the IP queueing subsystem with the frame aggregation functionality of the Wi-Fi MAC, wireless channel utilisation can be increased. Chapter 4 addresses hypotheses **H1** and **H4**. AAQ is a queuing algorithm that enables information exchange between the IP queueing subsystem and the Wi-Fi driver to aid with frame aggregation. AAQ provides one bit of information to the Wi-Fi driver indicating if there are more packets in the queue for the given Wi-Fi station. This information is used by the device driver to decide if frame aggregation can be scheduled for that Wi-Fi station.

Finally, at the datalink layer, interactions within the Wi-Fi MAC protocol itself can reduce achieved goodput in Wi-Fi networks. While AAQ and TAO help in aggregating many packets in a single aggregate, the interaction between frame aggregation and the default rate control algorithm used in the Linux operating system results in a potentially large aggregate being broken into multiple smaller aggregates. The culprit is the probing frames transmitted by the rate control algorithm used for exploring new transmit rates: a relic of the legacy IEEE 802.11b rate adaptation system inherited by the modern Wi-Fi design. This interaction diminishes the performance gains offered by AAQ and TAO. In Chapter 5 we presented AARC, a rate control algorithm that is aggregate friendly. In AARC, Wi-Fi stations enlist the help of channel state information to aid in transmit rate decisions. Using AARC, Wi-Fi stations probe the channel using whole aggregates, as opposed to single-frame probing. AARC provides mechanisms to guard against the continuous loss of probing frames, allowing Wi-Fi stations to transmit the large aggregates achieved by AAQ and TAO.

6.2 Discussion

We now reflect on the relationship between TCP/HACK and TAO, which solve the same problem by different means, and discuss the generality of our techniques, as well as evolution of the Linux Wi-Fi stack that occurred concurrently with the work in this thesis.

As mentioned earlier, both TCP/HACK and TAO address the same problem: the overhead incurred by TCP ACKs when transmitted over the Wi-Fi network. These two systems were designed for different scenarios. TCP/HACK targets the case where we give ourselves the freedom to change the MAC protocol, whereas TAO targets the case where we limit ourselves to today's IEEE 802.11 MAC protocol compliance. Which approach is preferable?

While we don't provide empirical results comparing TCP/HACK and TAO, we believe that when MAC changes are feasible, TCP/HACK should be used. TCP/HACK eliminates most TCP ACK transmissions by encapsulating TCP ACK information in link-layer ACKs, reducing Wi-Fi channel overhead and collision probability. By contrast, TAO reduces TCP ACK Wi-Fi channel acquisitions, but it does not completely eliminate them. TCP ACKs are still transmitted over the wireless channel and incur channel access overhead. As the number of Wi-Fi stations increases, the probability of collisions increases, so TCP/HACK should provide better performance then TAO. TAO, on the other hand, is directly usable in scenarios where we cannot change the MAC protocol and need a design that works with today's unmodified hardware. As described in Chapter 4, TAO queues TCP ACKs until all TCP ACKs for the received aggregate have been generated. While keeping TCP ACKs in the queue adds delay, TAO's benefits outweigh its cost (Section 4.8.4). Sending large aggregates reduces the number of channel acquisitions, which reduces the channel acquisition overhead and collision probability.

This thesis presents TCP/HACK as implemented for TCP. We think that the TCP/HACK concept is general and can be applied to other reliable protocols. TCP carries the majority of today's Internet traffic, but new protocols such as QUIC [60] have gained popularity recently. QUIC is a user-level protocol that runs on top of UDP and is encrypted by default. If the QUIC application provided hints about acknowledgment packets down the stack, the TCP/HACK concept could be applied to QUIC in a similar way to TCP.

In this thesis we described TCP/HACK as an implementation for IEEE 802.11a, but TCP/HACK also works with the frame aggregation introduced in IEEE 802.11n. A key enhancement to the MAC protocol introduced in IEEE 802.11n, frame aggregation, provides better utilisation of the high PHY rates introduced in IEEE 802.11n. We evaluated TCP/HACK (see Section 3.4.4) with frame aggregation in simulations where we assumed that Wi-Fi stations send the maximum number of allowed frames in an aggregate, as this this maximises channel utilisation [75]. But was the assumption that Wi-Fi stations send the maximum number of allowed frames in an aggregate correct?

As presented in detail in Chapter 4, the aggregate size status quo for commercial Wi-Fi cards and the Linux Wi-Fi stack was unfortunately too small to fully utilise the available PHY capacity (see Section 4.5). Using the insights gathered from our experimental testbed motivated the design of AAQ and TAO, lightweight approaches that coordinate packet processing decisions in the TCP, IP queuing, end-host MAC, and Wi-Fi driver layers to ensure that a sender transmits full-sized aggregates.

Concurrent with the work in this thesis, the Linux community changed the design of the device driver and the *mac80211* layer in a way that affects the aggregate sizes the Linux system generates [61, 62]. Their new design introduces queues in the *mac80211* layer, and packets are pulled from the queues by the device driver instead of the *mac80211* layer pushing them to the device driver. Is there any advantage to this new design as compared to AAQ?

Each TID in the new design gets a queue managed by a *CoDeL* algorithm, and while we have not evaluated this new design, we believe the results will be similar to those we present for *CoDeL* in Chapter 4. *CoDeL* measures the time packets stay in queues regardless of the number of Wi-Fi stations the AP is serving. In Wi-Fi the time packets stay in queues for a Wi-Fi client is a function of how many Wi-Fi stations the AP serves. As a result, *CoDeL* is too aggressive in targeting low delay, resulting in few packets residing in queues and preventing formation of large aggregates. In contrast, AAQ learns the exact number of packets that can be dequeued for transmission from the device driver, resulting in large aggregates, which reduces wireless channel overheads. One could make *CoDeL* Wi-Fi-aware by keeping a count of the number of Wi-Fi clients being served by the AP, and measuring the average time it takes for a Wi-Fi station to transmit. Using these two measurements, one could potentially arrive at a target time for each Wi-Fi client, but it is not clear even if then *CoDeL* will keep enough packets in queues to form large aggregates. Also, their new design does not solve the problem of staggered TCP ACKs, which TAO solves. We believe that AAQ offers a simpler design for achieving high goodput for bulk traffic on Wi-Fi networks.

6.3 Future Work

Frame aggregation is an important feature that allows the IEEE 802.11 MAC protocol to keep up with ever-increasing PHY rates. While in IEEE 802.11n the use of frame aggregation was optional, it was made mandatory in IEEE 802.11ac, so all transmissions should be aggregated. The upcoming IEEE 802.11ax standard makes further enhancements to frame aggregation. Prior to IEEE 802.11ax, all frames in an aggregate belonged to the same traffic identifier (TID, indicating the data priority such as best-effort, voice, etc.). IEEE 802.11ax introduces the Multi-Traffic Identifier Aggregated MAC Protocol Data Unit (Multi-TID AMPDU), which allows packets with different priorities to be aggregated together, so that Wi-Fi stations can aggregate more efficiently. For example, in situations when there are not enough packets in the queue from a particular TID, the transmitter can group frames from other TIDs to form a large aggregate, improving channel utilisation. This extension in IEEE 802.11ax should further enhance the benefit of our techniques for producing large aggregates, though we leave that exploration of IEEE 802.11ax to future work.

Frame aggregation is an important technique that improves channel utilisation and will continue to be important in the future. Despite this importance, one of the side effects of using frame aggregation is traffic burstiness. Burstiness can negatively impact certain types of traffic, such as voice. We leave the analysis of these effects on TCP flows as future work.

In addition, currently all frames in an aggregate must belong to a single Wi-Fi MAC destination address. Improving frame aggregation so that it can accept frames for multiple MAC destinations could further improve aggregation efficiency, but this will require hardware changes. We leave this improvement as future work.

In Chapter 5 we describe AARC, though our implementation only supports one spatial stream. Extending AARC to support multiple spatial streams requires detailed knowledge of how the Wi-Fi hardware processes spatial streams. In addition, with multiple streams, the algorithm for exploration of new rates must be amended. One can use the same approach for exploring rates as Minstrel does in Linux, or use the zigzag approach taken in MiRA [72]. Furthermore, the receiver must report the effective SNR difference for each of the spatial streams and the transmitter must track the effective SNR difference for each stream. We leave these enhancements to AARC to future work.

In AARC we choose to probe using full aggregates, as opposed to using single-frame probing, as done by Minstrel. To guard against repetitive losses of full aggregates when probing we use an exponential backoff algorithm. A better approach would be to send full aggregates, but within an aggregate, mark single frames as probing frames by modulating them at the MCS we wish to probe. Status-quo IEEE 802.11 frame aggregation mandates that all packets in an aggregate are to be modulated using the same MCS, so MAC standard and hardware changes would be necessary for this approach to work.

The latest IEEE 802.11ax [23] standard, which postdates much of the work in this thesis, departs from the traditional CSMA/CA-only MAC protocol by using a centralised channel access protocol based on OFDMA. One of the primary goals of IEEE 802.11ax is to provide connectivity to Wi-Fi networks with large numbers of clients, such as IoT devices. Using OFDMA, IEEE 802.11ax divides the wireless channel into sub-carriers and groups them into chunks called Resource Units (RUs), which are assigned to Wi-Fi clients. The AP can transmit simultaneously to multiple Wi-Fi clients using assigned RUs. As mentioned earlier, frame aggregation is still used in IEEE 802.11ax, and *Machrouh et al.* show that sending large aggregates improves performance in IEEE 802.11ax Wi-Fi networks [63]. Our techniques presented in Chapter 4 will thus complement IEEE 802.11ax.

IEEE 802.11ax presents a particular challenge for rate-control algorithms as the success of a rate is also dependent on the RU assigned by the AP. This adds another extra dimension to the rate adaptation algorithm, adding complexity. Adapting AARC to work with IEEE 802.11ax would be an interesting future research direction.

Finally, during the implementation of our techniques using the *ath9k* open-source driver we unearthed interesting questions regarding the interface between the device driver and the hardware, and how they interact. Some of the problematic interactions we noticed are a relic of legacy systems that pre-date frame aggregation. For example, often Wi-Fi cards offer the ability to specify multiple transmit rate policies in the form of multi-rate retry tables. This allows the device driver to specify a chain of transmit rates that the Wi-Fi hardware will independently try in cases when the first transmit rate fails. While this makes sense for single-frame transmission, when using frame aggregation this potentially can lead to multiple problems. First, if the transmit rate specified by the rate adaptation algorithm fails, it is desirable to report back the failure as quickly as possible so rate adaptation reacts in a timely fashion. Secondly, retrying at a different transmit rate leads to fairness problems, as Wi-Fi stations get more then their share of transmission time. Another interaction concerns frame aggregation. In *ath9k*, aggregates are formed in the device driver. In addition the device driver is responsible for scheduling aggregate transmissions. One problem with this approach is that once an aggregate is scheduled for transmission it cannot be modified. A potentially superior approach is to move the process of frame aggregation into the *mac80211* software layer. This way, *mac80211* will maintain TID queues, and TID state such as the block ACK window would be updated accordingly by the device driver. Given that information, *mac80211* would mark TID queues that are ready for transmission i.e, TID queues that have enough frames enqueued (TID queues that have reached the block ACK window capacity). The device driver would only schedule TIDs marked ready for transmission. Potentially, the device driver could DMA all ready TID queues to the hardware and the hardware would schedule all DMAed queues.

The aforementioned approach would also eliminate the double-buffering problem we describe in Chapter 4. The IP queueing discipline, such as AAQ, would interact with the *mac80211* layer instead of the device driver. This approach simplifies the device driver's design, as the only responsibility of the device driver becomes to schedule (or rather just DMA queues and let the hardware schedule) transmissions. In addition, the multi-rate retry table would be removed from the hardware, so transmit failures could be reported to the rate adaptation algorithm immediately. This design would also allow for frame retransmission to be handled by the *mac80211* layer, further simplifying the device driver's design. We leave this refactoring of responsibilities between driver and hardware to future work.

To conclude, in this thesis we have shown that the nuances of cross-layer interactions incur extra overhead that reduces Wi-Fi network performance. We have shown that using a cross-layer approach in designing Wi-Fi systems can eliminate these overheads, and so improve Wi-Fi network performance.

Appendix A

IEEE 802.11n Analytical Goodput

In this chapter we show the goodput analytical goodput prediction for TCP over IEEE 802.11n DCF. Let $t_{U_{data}}$ and t_{Frame}, be the over-the-air time for a single user data packet and the corresponding IEEE 802.11n frame with added IEEE 802.11 headers, IP headers, and TCP headers, respectively. The DCF utilisation to send an A-MPDU of size n is defined as follows:

$$Utilisation = \frac{n \times t_{U_{data}}}{n \times t_{Frame} + \lfloor \frac{n}{2} \rfloor \times t_{T_{ACK}} + DCF_{overhead} + AMPD_{overhead}} \tag{A.1}$$

$DCF_{overhead}$ the IEEE 802.11's DCF overhead defined as follows:

$$DCF_{overhead} = 2 \times (DIFS + E_{backoff} + t_{ba} + t_{preamble} + SIFS + t_{rtscts}) \tag{A.2}$$

Two A-MPDUs have to be transmitted. First A-MPDU is for transmitting TCP segments and the second A-MPDU is for transmitting TCP ACKs, both incurring DCF's channel acquisition overhead.

$t_{preamble}$ is the time it takes to serialise the preambles prepended to each A-MPDU.

$$t_{preamble} = STF + LTF + SIG + (N_{ss} \times LTF1) \tag{A.3}$$

N_{ss} is the number of independent data streams used (i.e, the number of antennas).

$$AMPDU_{overhead} = \frac{3 \times n}{2} \times t_{delim} + \frac{3 \times n}{2} - 2 \times t_{pad} \tag{A.4}$$

Each subframe in A-MPDU starts with the delimiter, a header that marks the beginning of the subframe. In addition, all frames but the last one are padded to be multiple octets. In total there are n TCP data and $\frac{1}{2} \times n$ TCP ACKs, therefore in total there are $\frac{3 \times n}{2}$ subframes. Table A.1 show the values for the parameters used.

Parameter	**Value**	**Description**
$t_{U_{data}}$	$20\ \mu s - 2\ ms$	Time to transmit 1448 byte of user data for IEEE 802.11n bit-rates. We assume TCP Timestamp Options therefore after removing 20 bytes of TCP header, 12 bytes for TCP Timestamp options, and 20 byte IP header, total user data is 1448.
t_{Frame}	$20.4\ \mu s - 2\ ms$	Time to transmit 1536 byte frames (after adding TCP,IP, and MAC headers and FCS).
$SIFS$	$16\ \mu s$	Short Interframe Spacing time.
$SLOT$	$9\ \mu s$	Timeslot duration.
$DIFS$	$43\ \mu s$	DCF Interframe Spacing. $DIFS = SIFS + (AIFS \times SLOT)$ $AIFS = AIFSN[AC]$. We use Best Effort AIFSN which is 3.
$E_{backoff}$	$67.5\ \mu s$	Expected backoff wait time assuming $CW_{min} = 16 \times TSLOT$.
t_{ba}	$10.6\ \mu s$	Time to transmit link-layer Block Ack at 24 Mbps.
$t_{T_{ACK}}$	$0.7 - 69\ \mu s$	Time to transmit TCP ACKs.
STF	$8\ \mu s$	Short Training Field.
LTF	$8\ \mu s$	Long Training Field.
$LTF1$	$4\ \mu s$	Long Training Field for each spatial stream.
SIG	$8\ \mu s$	SIGNAL time.
t_{rtscts}	$27.2\ \mu s$	Time to complete the RTS/CTS exchange at 24 Mbps.

Table A.1: IEEE 802.11n goodput calculation parameters

Bibliography

[1] Linux mac80211 Layer. URL `https://wireless.wiki.kernel.org/en/developers/documentation/mac80211`.

[2] *netlink - communication between kernel and user space.* URL `http://man7.org/linux/man-pages/man7/netlink.7.html`.

[3] Internet Protocol. RFC 791, 1981. URL `https://www.ietf.org/rfc/rfc791.txt`.

[4] Transmission Control Protocol. RFC 793, 1981. URL `https://www.ietf.org/rfc/rfc793.txt`.

[5] IEEE Standard for Telecommunications and Information Exchange Between Systems - LAN/MAN Specific Requirements - Part 11: Wireless Medium Access Control (MAC) and physical layer (PHY) specifications: High Speed Physical Layer in the 5 GHz band. *IEEE Std 802.11a-1999*, pages 1–102, Dec 1999. doi: 10.1109/IEEESTD.1999.90606.

[6] IEEE Standard for Information Technology- Telecommunications and Information Exchange Between Systems- Local and Metropolitan Area Networks- Specific Requirements- Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. *ANSI/IEEE Std 802.11, 1999 Edition (R2003)*, pages i–513, 2003. doi: 10.1109/IEEESTD.2003.95617.

[7] IEEE Standard for Information Technology- Telecommunications and Information Exchange Between Systems- Local and Metropolitan Area

Networks- Specific Requirements Part Ii: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. *IEEE Std 802.11g-2003 (Amendment to IEEE Std 802.11, 1999 Edn. (Reaff 2003) as amended by IEEE Stds 802.11a-1999, 802.11b-1999, 802.11b-1999/Cor 1-2001, and 802.11d-2001)*, pages i–67, 2003. doi: 10.1109/IEEESTD.2003.94282.

[8] Ieee standard for information technology–local and metropolitan area networks–specific requirements–part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications - amendment 8: Medium access control (mac) quality of service enhancements. *IEEE Std 802.11e-2005 (Amendment to IEEE Std 802.11, 1999 Edition (Reaff 2003)*, pages 1–212, 2005.

[9] Ieee standard for local and metropolitan area networks part 16: Air interface for broadband wireless access systems amendment 3: Advanced air interface. *IEEE Std 802.16m-2011(Amendment to IEEE Std 802.16-2009)*, pages 1–1112, 2011.

[10] Ieee standard for information technology–telecommunications and information exchange between systems—local and metropolitan area networks–specific requirements–part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications–amendment 4: Enhancements for very high throughput for operation in bands below 6 ghz. *IEEE Std 802.11ac(TM)-2013 (Amendment to IEEE Std 802.11-2012, as amended by IEEE Std 802.11ae-2012, IEEE Std 802.11aa-2012, and IEEE Std 802.11ad-2012)*, pages 1–425, 2013.

[11] Network simulator, 2014. URL `https://www.nsnam.org/`.

[12] Atheros Open Source Driver, 2017. URL `https://wireless.kernel.org/en/users/Drivers/ath9k`.

[13] hostapd: IEEE 802.11 AP, IEEE 802.1X/WPA/WPA2/EAP/RADIUS Authenticator, 2017. URL `https://w1.fi/hostapd/`.

[14] iPerf - The ultimate speed test tool for TCP, UDP and SCTP, 2017. URL `https://iperf.fr/`.

[15] Linksys Cisco AP, 2017. URL `https://www.linksys.com/us/support-product?pid=01t80000003K7dYAAS`.

[16] Linux Operating System 3.18.7, 2017. URL `https://launchpad.net/linux/+milestone/3.18.7`.

[17] TCPDUMP and Libpcap, 2017. URL `https://www.tcpdump.org`.

[18] TCP Extensions for High Performance, 2017. URL `https://www.ietf.org/rfc/rfc1323.txt`.

[19] The NewReno Modification to TCP's Fast Recovery Algorithm, 2017. URL `https://tools.ietf.org/html/rfc3782`.

[20] TCP Selective Acknowledgment Options, 2017. URL `https://tools.ietf.org/html/rfc2018`.

[21] TCP small queues, 2017. URL `https://lwn.net/Articles/507065/`.

[22] Ubuntu Linux, 2017. URL `http://releases.ubuntu.com/14.04/`.

[23] Ieee draft standard for information technology – telecommunications and information exchange between systems local and metropolitan area networks – specific requirements part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications amendment enhancements for high efficiency wlan. *IEEE P802.11ax/D6.0, November 2019*, pages 1–780, 2019.

[24] Minstrel Linux, 2019. URL `https://wireless.wiki.kernel.org/en/developers/documentation/mac80211/ratecontrol/minstrel`.

[25] 3GPP. LTE-M, July 2020. URL `https://www.gsma.com/iot/long-term-evolution-machine-type-communication-lte-mtc-cat-m1/`.

[26] 3GPP. NB-IOT, July 2020. URL `https://www.3gpp.org/news-events/3gpp-news/1785-nb_iot_complete`.

[27] Norman Abramson. THE ALOHA SYSTEM: Another Alternative for Computer Communications. In *Proceedings of the November 17-19, 1970, Fall Joint Computer Conference*, AFIPS '70 (Fall), pages 281–285, New York, NY, USA, 1970. ACM. doi: 10.1145/1478462.1478502. URL `http://doi.acm.org/10.1145/1478462.1478502`.

[28] Victor Bahl, Ranveer Chandra, Thomas Moscibroda, Rohan Murty, and Matt Welsh. White space networking with wi-fi like connectivity. In *SIGCOMM 2009: ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Barcelona, Spain*. Association for Computing Machinery, Inc., August 2009. URL `https://www.microsoft.com/en-us/research/publication/white-space-networking-with-wi-fi-like-connectivity/`. SIGCOMM Best Paper Award.

[29] A. Bakre and B. R. Badrinath. I-TCP: indirect TCP for mobile hosts. In *Distributed Computing Systems, 1995., Proceedings of the 15th International Conference on*, pages 136–143, May 1995. doi: 10.1109/ICDCS.1995.500012.

[30] H. Balakrishnan, V.N. Padmanabhan, S. Seshan, and R.H. Katz. A comparison of mechanisms for improving TCP performance over wireless links. *Networking, IEEE/ACM Transactions on*, 5(6):756–769, Dec 1997. ISSN 1063-6692. doi: 10.1109/90.650137.

[31] Guy Bégin and David Haccoun. Performance of sequential decoding of high-rate punctured convolutional codes. *IEEE Trans. Communications*,

42(234):966–978, 1994. doi: 10.1109/TCOMM.1994.580205. URL `https://doi.org/10.1109/TCOMM.1994.580205`.

[32] Vaduvur Bharghavan, Alan Demers, Scott Shenker, and Lixia Zhang. Macaw: A media access protocol for wireless lan's. *SIGCOMM Comput. Commun. Rev.*, 24(4):212–225, October 1994. ISSN 0146-4833. doi: 10.1145/190809.190334. URL `https://doi.org/10.1145/190809.190334`.

[33] G. Bianchi. Performance analysis of the IEEE 802.11 distributed coordination function. *Selected Areas in Communications, IEEE Journal on*, 18(3):535–547, March 2000. ISSN 0733-8716. doi: 10.1109/49.840210.

[34] John Bicket. Bit-rate Selection in Wireless Networks. Master's thesis, MIT, 2 2005.

[35] John Bicket, Sanjit Biswas, Daniel Aguayo, and Robert Morris. Architecture and Evaluation of the MIT Roofnet Mesh Network .

[36] F. Cali, M. Conti, and E. Gregori. IEEE 802.11 wireless LAN: capacity analysis and protocol enhancement. In *INFOCOM '98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 142–149 vol.1, Mar 1998. doi: 10.1109/INFCOM.1998.659648.

[37] Frederico Calì, Marco Conti, and Enrico Gregori. Dynamic Tuning of the IEEE 802.11 Protocol to Achieve a Theoretical Throughput Limit. *IEEE/ACM Trans. Netw.*, 8(6):785–799, December 2000. ISSN 1063-6692. doi: 10.1109/90.893874. URL `http://dx.doi.org/10.1109/90.893874`.

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

[39] CISCO. Cisco Annual Internet Report (2018–2023) White Paper, March 2020. URL `https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html`.

[40] Lara Deek, Eduard Garcia-Villegas, Elizabeth Belding, Sung-Ju Lee, and Kevin Almeroth. A Practical Framework for 802.11 MIMO Rate Adaptation. *Comput. Netw.*, 83(C):332–348, June 2015. ISSN 1389-1286. doi: 10.1016/j.comnet.2015.03.015. URL `http://dx.doi.org/10.1016/j.comnet.2015.03.015`.

[41] Lars Eggert and Wesley Eddy. Towards more expressive transport-layer interfaces. 01 2006. doi: 10.1145/1186699.1186719.

[42] Cheng Peng Fu and S.C. Liew. TCP Veno: TCP enhancement for transmission over wireless access networks. *Selected Areas in Communications, IEEE Journal on*, 21(2):216–228, Feb 2003. ISSN 0733-8716. doi: 10.1109/JSAC.2002.807336.

[43] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark Buffers in the Internet. *Communications of the ACM*, 55(1), January 2012.

[44] B. Ginzburg and A. Kesselman. Performance analysis of A-MPDU and A-MSDU aggregation in IEEE 802.11n. In *Sarnoff Symposium, 2007 IEEE*, pages 1–5, April 2007. doi: 10.1109/SARNOF.2007.4567389.

[45] Andrea Goldsmith. *Wireless Communications*. Cambridge University Press, USA, 2005. ISBN 0521837162.

[46] Daniel Halperin, Wenjun Hu, Anmol Sheth, and David Wetherall. Predictable 802.11 packet delivery from wireless channel measurements. *SIGCOMM Comput. Commun. Rev.*, 41(4):–, August 2010. ISSN 0146-4833. URL `http://dl.acm.org/citation.cfm?id=2043164.1851203`.

[47] Daniel Halperin, Wenjun Hu, Anmol Sheth, and David Wetherall. Tool release: Gathering 802.11n traces with channel state information. *Computer Communication Review*, 41:53, 01 2011. doi: 10.1145/1925861.1925870.

[48] Martin Heusse, Franck Rousseau, Romaric Guillier, and Andrzej Duda. Idle Sense: An Optimal Access Method for High Throughput and Fairness in Rate Diverse Wireless LANs. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '05, pages 121–132, New York, NY, USA, 2005. ACM. ISBN 1-59593-009-4. doi: 10.1145/1080091.1080107. URL `http://doi.acm.org/10.1145/1080091.1080107`.

[49] IEEE Computer Society. Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 8: Medium Access Control (MAC) Quality of Service Enhancements. *IEEE Std 802.11e-2005 (Amendment to IEEE Std 802.11, 1999 Edition (Reaff 2003)*, 2005.

[50] IEEE Computer Society. Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 5: Enhancements for Higher Throughput. *IEEE Std 802.11n-2009 (Amendment to IEEE Std 802.11-2007 as amended by IEEE Std 802.11k-2008, IEEE Std 802.11r-2008, IEEE Std 802.11y-2008, and IEEE Std 802.11w-2009)*, pages 1–565, 2009.

[51] V. Jacobson. *Compressing TCP/IP Headers.* RFC 1144, Feb 1990.

[52] V. Jacobson, R. Braden, D. Borman, and Cray Research. *TCP Extensions for High Performance.* RFC 1323, May 1992.

[53] Mayank Jain, Jung Il Choi, Taemin Kim, Dinesh Bharadia, Siddharth Seth, Kannan Srinivasan, Philip Levis, Sachin Katti, and Prasun Sinha. Practical, real-time, full duplex wireless. In *Proceedings of the 17th Annual International Conference on Mobile Computing and Networking*,

MobiCom '11, page 301–312, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304924. doi: 10.1145/2030613.2030647. URL `https://doi.org/10.1145/2030613.2030647`.

[54] M.K. Kadiyala, D. Shikha, R. Pendse, and N. Jaggi. Semi-Markov process based model for performance analysis of wireless LANs. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2011 IEEE International Conference on*, pages 613–618, March 2011. doi: 10.1109/PERCOMW.2011.5766962.

[55] A. Kamerman and L. Monteban. WaveLAN-II: A high-performance wireless LAN for the unlicensed band. *Bell Labs Technical Journal*, 2(3): 118–133, Summer 1997. ISSN 1089-7089. doi: 10.1002/bltj.2069.

[56] Phil Karn. MACA - A New Channel Access Method for Packet Radio. 1990. URL `https://www.tapr.org/pub_cnc09.html`.

[57] Phil Karn. Maca : A new channel access method for packet radio. 1990.

[58] L. Kleinrock and F.A. Tobagi. Packet Switching in Radio Channels: Part I–Carrier Sense Multiple-Access Modes and Their Throughput-Delay Characteristics. *Communications, IEEE Transactions on*, 23(12):1400–1416, Dec 1975. ISSN 0090-6778. doi: 10.1109/TCOM.1975.1092768.

[59] Eddie Kohler, Mark Handley, and Sally Floyd. Designing DCCP: Congestion Control Without Reliability. In *SIGCOMM*, 2006.

[60] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17,

page 183–196, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346535. doi: 10.1145/3098822.3098842. URL https://doi.org/10.1145/3098822.3098842.

[61] Linux. mac80211: add an intermediate software queue implementation., April 2015. URL https://github.com/torvalds/linux/commit/ba8c3d6f16a1f9305c23ac1d2fd3992508c5ac03.

[62] Linux. ath9k: Switch to using mac80211 intermediate software queues., November 2016. URL https://github.com/torvalds/linux/commit/50f08edf98096a68f01ff4566b605a25bf8e42ce.

[63] Z. Machrouh and A. Najid. High efficiency wlans ieee 802.11ax performance evaluation. In *2018 International Conference on Control, Automation and Diagnosis (ICCAD)*, pages 1–5, 2018.

[64] E. Magistretti, K. Chintalapudi, B. Radunovic, and R. Ramjee. WiFi-Nano: Reclaiming WiFi Efficiency through 800 ns Slots. In *Proc. IEEE/ACM MobiCom*, Sep 2011.

[65] G. R. Mendez, M. A. Md Yunus, and S. C. Mukhopadhyay. A wifi based smart wireless sensor network for an agricultural environment. In *2011 Fifth International Conference on Sensing Technology*, pages 405–410, 2011.

[66] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Commun. ACM*, 19(7):395–404, July 1976. ISSN 0001-0782. doi: 10.1145/360248.360253. URL http://doi.acm.org/10.1145/360248.360253.

[67] Robert T. Morris, John C. Bicket, and John C. Bicket. Bit-rate selection in wireless networks. Technical report, Master's thesis, MIT, 2005.

[68] Kathleen Nichols and Van Jacobson. Controlling Queue Delay. *Queue*,

10(5):20:20–20:34, May 2012. ISSN 1542-7730. doi: 10.1145/2208917.2209336. URL `http://doi.acm.org/10.1145/2208917.2209336`.

[69] Brian O'Neill. Dab eureka-147: a european vision for digital radio. *New Media & Society*, 11(1-2):261–278, 2009. doi: 10.1177/1461444808099578. URL `https://doi.org/10.1177/1461444808099578`.

[70] M. Ozdemir, Daqing Gu, A.B. McDonald, and Jinyun Zhang. Enhancing MAC Performance with a Reverse Direction Protocol for High-Capacity Wireless LANs. In *Vehicular Technology Conference, 2006. VTC-2006 Fall. 2006 IEEE 64th*, pages 1–5, 2006. doi: 10.1109/VTCF.2006.461.

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

[72] Ioannis Pefkianakis, Yun Hu, Starsky H.Y. Wong, Hao Yang, and Songwu Lu. MIMO Rate Adaptation in 802.11N Wireless Networks. In *Proceedings of the Sixteenth Annual International Conference on Mobile Computing and Networking*, MobiCom '10, pages 257–268, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0181-7. doi: 10.1145/1859995.1860025. URL `http://doi.acm.org/10.1145/1859995.1860025`.

[73] G. Pelletier, InterDigital Communications, K. Sandlund, Ericsson, L-E. Jonsson, M. West, and Siemens/Roke Manor. *RObust Header Compression (ROHC): A Profile for TCP/IP (ROHC-TCP).* RFC 6846, January 2013.

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

[75] Lynne Salameh, Astrit Zhushi, Mark Handley, Kyle Jamieson, and Brad Karp. HACK: Hierarchical ACKs for Efficient Wireless Medium Utilization. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 359–370, Philadelphia, PA, June 2014. USENIX Association.

ISBN 978-1-931971-10-2. URL `https://www.usenix.org/conference/atc14/technical-sessions/presentation/salameh`.

[76] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4), Nov 1984.

[77] J. Salz. Digital transmission over cross-coupled linear channels. *AT T Technical Journal*, 64(6):1147–1159, 1985.

[78] Pasi Sarolahti. Transport-layer Considerations for Explicit Cross-layer Indications. Internet-Draft draft-sarolahti-tsvwg-crosslayer-01, Internet Engineering Task Force, March 2007. URL `https://datatracker.ietf.org/doc/html/draft-sarolahti-tsvwg-crosslayer-01`. Work in Progress.

[79] Boris Schling. *The Boost C++ Libraries*. XML Press, 2011. ISBN 0982219199.

[80] M. Shreedhar and George Varghese. Efficient Fair Queueing Using Deficit Round Robin. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '95, pages 231–242, New York, NY, USA, 1995. ACM. ISBN 0-89791-711-1. doi: 10.1145/217382.217453. URL `http://doi.acm.org/10.1145/217382.217453`.

[81] D. Skordoulis, Q. Ni, H. Chen, A. P. Stephens, C. Liu, and A. Jamalipour. IEEE 802.11n MAC frame aggregation mechanisms for next-generation high-throughput WLANs. *IEEE Wireless Communications*, 15(1):40–47, February 2008. ISSN 1536-1284. doi: 10.1109/MWC.2008.4454703.

[82] Randall Stewart. *Stream Control Transmission Protocol*. RFC 4960, September 2007.

[83] Godfrey Tan and John Guttag. Time-based Fairness Improves Performance in Multi-rate WLANs. In *Proceedings of the Annual Con-*

ference on USENIX Annual Technical Conference, ATEC '04, pages 23–23, Berkeley, CA, USA, 2004. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1247415.1247438`.

[84] K. Tan, J. Fang, Y. Zhang, S. Chen, L. Shi, J. Zhang, and Y. Zhang. Fine-grained Channel Access in Wireless LAN. In *Proc. ACM SIGCOMM*, Aug 2010.

[85] Kun Tan, Jiansong Zhang, Ji Fang, He Liu, Yusheng Ye, Shen Wang, Yongguang Zhang, Haitao Wu, Wei Wang, and Geoffrey Voelker. SoRa: High Performance Software Radio Using General Purpose Multi-core Processors. In *Proc. of the* NSDI *Conf.*, April 2009.

[86] Diane Tang and Mary Baker. Analysis of a local-area wireless network . In *Proceedings of the 6th annual international conference on Mobile computing and networking*, MobiCom '00, pages 1–10, New York, NY, USA, 2000. ACM. ISBN 1-58113-197-6. doi: 10.1145/345910.345912. URL `http://doi.acm.org/10.1145/345910.345912`.

[87] F.A. Tobagi and L. Kleinrock. Packet Switching in Radio Channels: Part II–The Hidden Terminal Problem in Carrier Sense Multiple-Access and the Busy-Tone Solution. *Communications, IEEE Transactions on*, 23 (12):1417–1433, Dec 1975. ISSN 0090-6778. doi: 10.1109/TCOM.1975. 1092767.

[88] David Tse and Pramod Viswanath. *Fundamentals of Wireless Communication.* Cambridge University Press, New York, NY, USA, 2005. ISBN 0-5218-4527-0.

[89] Andrew J. Viterbi. *CDMA: Principles of Spread Spectrum Communication.* Addison Wesley Longman Publishing Co., Inc., USA, 1995. ISBN 0201633744.

[90] Hai L. Vu and Taka Sakurai. Collision probability in saturated ieee 802.11

networks. In *In Australian Telecommunication Networks and Applications Conference*, 2006.

[91] Mythili Vutukuru, Hari Balakrishnan, and Kyle Jamieson. Cross-layer wireless bit rate adaptation. *SIGCOMM Comput. Commun. Rev.*, 39 (4):3–14, August 2009. ISSN 0146-4833. doi: 10.1145/1594977.1592571. URL `http://doi.acm.org/10.1145/1594977.1592571`.

[92] Chonggang Wang and Weiwen Tang. A probability-based algorithm to adjust contention window in IEEE 802.11 DCF. In *Communications, Circuits and Systems, 2004. ICCCAS 2004. 2004 International Conference on*, volume 1, pages 418–422 Vol.1, June 2004. doi: 10.1109/ICCCAS.2004.1346122.

[93] S. B. Weinstein. The history of orthogonal frequency-division multiplexing [history of communications]. *IEEE Communications Magazine*, 47(11): 26–35, 2009.

[94] Starsky H. Y. Wong, Hao Yang, Songwu Lu, and Vaduvur Bharghavan. Robust Rate Adaptation for 802.11 Wireless Networks. In *Proceedings of the 12th Annual International Conference on Mobile Computing and Networking*, MobiCom '06, pages 146–157, New York, NY, USA, 2006. ACM. ISBN 1-59593-286-0. doi: 10.1145/1161089.1161107. URL `http://doi.acm.org/10.1145/1161089.1161107`.

[95] Yang Xiao and J. Rosdahl. Throughput and delay limits of IEEE 802.11. *Communications Letters, IEEE*, 6(8):355–357, Aug 2002. ISSN 1089-7798. doi: 10.1109/LCOMM.2002.802035.

[96] Yaxiong Xie, Zhenjiang Li, and Mo Li. Precise power delay profiling with commodity wifi. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, page 53–64, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3619-2.

doi: 10.1145/2789168.2790124. URL `http://doi.acm.org/10.1145/2789168.2790124`.

[97] Xilinx. Xilinx FPGA, July 2016. URL `http://www.xilinx.com/products/silicon-devices/fpga.html`.

[98] R. Yavatkar and N. Bhagawat. Improving end-to-end performance of TCP over mobile internetworks. In *Mobile Computing Systems and Applications, 1994. Proceedings., Workshop on*, pages 146–152, Dec 1994. doi: 10.1109/MCSA.1994.513474.

[99] J. Zhang, K. Tan, J. Zhao, H. Wu, and Y. Zhang. A practical snr-guided rate adaptation. In *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*, pages 2083–2091, April 2008. doi: 10.1109/INFOCOM.2008.274.

[100] Z. Zhao, F. Zhang, S. Guo, X. Y. Li, and J. Han. RainbowRate: MIMO rate adaptation in 802.11n WiLD links. In *2014 IEEE 33rd International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, Dec 2014. doi: 10.1109/PCCC.2014.7017084.