Sharing is Caring: Throughput Fairness in Virtual Wireless LANs

Simon Kaufmann

4th Year Project Report
Computer Science and Mathematics
School of Informatics
University of Edinburgh

2021

Abstract

Network virtualisation can be a useful tool for wireless network providers to share costs for equipment and simplify deployment. Shared hardware costs can make a fair distribution of available throughput an important concern. C-VAP and AlphaAP are two related algorithms providing a fairness guarantee for uplink throughput and optimizing the total throughput using a dynamic control mechanism. This project provides the first implementation of C-VAP and AlphaAP on wireless hardware, thereby demonstrating the practical feasibility of the control-theoretic approach in making fairness guarantees using standard commercially available hardware components. The algorithm's underlying system model is extended to show how it can be used in connection with WLAN performance optimizations like frame aggregation and RTS handshakes. Finally, the implementation is thoroughly tested on a customisable, automated test bed, specificially designed for the project, and shown to outperform the conventional uncontrolled medium access mechanism in several cases while deviating from the desired fairness guarantee consistently by less than 2%. The project demonstrates how the firmware of publicly largely undocumented modern wireless chipsets can be used to provide low-level measurements of wireless channel conditions which form the basis for a controller based optimization of the network operation.

Acknowledgements

I would like to offer special thanks to my supervisor Dr. Paul Patras for his continuous guidance, feedback and support throughout the project and to Dr. Francesco Gringoli and Marco Cominelli for supporting me with their technical expertise and for their assistance with building up a large test setup to use during the project.

Contents

Chapter 1

Introduction

Global demand for mobile networking has more than tripled since 2017 and is expected to grow further at a similar rate over the next few years [12]. New mobile networking technologies like 5G can help to accommmodate the increased demand, but upgrades are costly to install and maintain [7].

Mobile data offloading from cellular networks to alternative WLANs (wireless local area networks) is an alternative which can help to lower costs for infrastructure providers and reduce the demand on cellular networks [14]. This is especially relevant in particularly crowded locations like shopping centres, event locations, and indoor venues.

Virtualisation is another technique which can be used to share WLAN equipment and thus operating costs between operators. It offers logical separation of networks which are operated on the same physical hardware and the virtual networks look to regular users indistinguishable from separated networks provided by physically different devices [13]. Virtualisation can help with the technical challenges of deploying physical hardware, such as finding suitable placement for antennas and connecting access points to the networking core etc.

One problem when operating virtualised WLANs is the distribution of the available data throughput, since virtual networks are required to share the total available bandwidth on a given channel. Especially when the network is operating under strain, a fair distribution of the throughput can become important for network operators who are sharing the operation costs.

Access points can use a modified queue management system to ensure that data traffic coming from the internet (downlink) is distributed fairly between the different stations or virtual networks, similar to the process described in [8]. For traffic originating from the different virtual networks (uplink), it is more difficult for the access point to guarantee a fair distribution of the available bandwidth. The decentralised access mechanism of WLANs gives equal access to every station connected to any of the virtual networks and therefore prioritises networks with more connected users.

C-VAP [3] and AlphaAP [21] are two related algorithms for solving this problem of fair distribution of uplink throughput between virtual WLANs. Both algorithms operate by dynamically modifying the channel access parameters for each virtual network based on the currently observed channel conditions. This allows the access point to slow down networks using more than their allocated share of total throughput and to increase traffic in networks using less throughput than allocated. Both algorithms are implemented completely at the access point and do not require clients to be modified.

Previously, C-VAP and AlphaAP were tested in a simulation setting, but the practical feasibility of deploying such a solution in real settings with off-the-shelf hardware has not been verified [3] [21]. Additionally, both C-VAP and AlphaAP do not consider different optimisations of the WLAN channel access mechanism.

This project starts by extending the system model of the C-VAP algorithm to account for extensions of the WLAN standard, such as frame aggregation and the RTS/CTS handshake mechanism, and to demonstrate how the algorithm can be used in scenarios where these extensions are used. Next, the feasibility of both algorithms is demonstrated by implementing it on standard commercially-available WLAN hardware (in particular, on router Asus RT-AC86U with wireless chipset Broadcom BCM4366), and finally the algorithm performance is evaluated in operation on a custom test bed under a range of network conditions both with and without the mentioned extensions.

We find that both algorithms are able to maximise network performance and ensure fairness between virtual networks under load, deviating less than 2% from the desired throughput distribution. Whenever a virtual network is not under full load, its full throughput demand will be served and the additional available bandwidth will be redistributed proportionally to the remaining virtual networks under full load. We further find that while the undocumented nature of the used WLAN chip led to some unsolved technical challenges regarding accurate measurements of the channel conditions when using frame aggregation or the RTS/CTS mechanism, the algorithm is able to ensure fair bandwidth distribution in these cases as well, though with a slightly lower total performance than using conventional network operation.

This work is structured as follows: Chapter 2 gives some background information on the WLAN technology and related work in the area, Chapter 3 outlines the changes to the C-VAP system model made for this project, Chapter 4 discusses the implementation of the algorithm on hardware followed by a description of the test bed used in Chapter 5. Finally, Chapter 6 ends with a discussion of the measured results of the algorithm in a range of settings.

Chapter 2

Background

The work in this project is based on wireless local area networks (WLAN). The technology behind WLANs was standardised by the Institute of Electrical and Electronics Engineers (IEEE) and released in the form of the IEEE 802.11 standard and its several amendments [10]. This chapter introduces the relevant parts of the 802.11 standard and concludes with a discussion of the C-VAP/AlphaAP algorithms for guaranteeing throughput fairness, comparing them to other approaches.

2.1 WLANs and Medium Access

A WLAN is a local computer network in which participating devices use wireless communication to exchange data. All WLANs in this project consist of an access point (AP) and regular participating devices (station / STA). The access point comes in the form of a WLAN router and is responsible for advertising the WLAN (using beacon frames which include the network configuration and are transmitted in regular intervals) and allowing stations to connect to the network [10].

It is possible for an access point to advertise several different logical/virtual networks. Each virtual network has a different identifier (SSID) and can have different advertised properties (like encryption etc.) but is operating at the same frequency as the other virtual networks provided by the same access point [10]. We refer to a virtual network as a virtual access point (VAP).

Medium access is an important consideration in wireless networking because all participating devices are accessing the network through the same medium (radio waves in the air at the same frequency) and cannot transmit data at the same time without causing interference. This applies to devices in all VAPs and means that the total time available for transmission has to be shared among all devices in all VAPs advertised by a single AP.

When two devices start sending at the same time, the data cannot be received properly by the access point or other stations. Such a failed transmission is called a collision. In wireless networks it is technologically challenging for a transmitting device to detect a collision during the transmission (which is different to, for example, a wired Ethernet

network). This means that during a collision the full duration of the intended transmission is wasted. The resulting high cost of collisions is the reason why the WLAN protocol was designed with avoidance of collisions in mind. CSMA/CA (Carrier-sense multiple access with collision avoidance) is the mechanism used by the IEEE 802.11 standard that aims to reduce the number of collisions occurring on the medium.

The following section discusses how CSMA/CA is implemented in the 802.11 standard and how stations determine when to transmit frames without causing a large number of collisions.

2.1.1 Contention Mechanism (DCF)

The medium access control mechanism for 802.11 WLANs is called DCF (Distributed Coordination Function). Stations with pending data frames continuously check whether the physical medium is occupied. After the medium is sensed free, stations are required to wait for a duration referred to as DIFS (DCF Interframe Space) [10]. If the medium is still free after DIFS has passed, the station starts a counting mechanism (the so-called backoff process) which needs to be completed before a frame can be sent.

The backoff process involves selecting a random integer from a uniform distribution of values between [1, CW] where CW refers to the contention window currently selected at this station. The station then runs a counter which decrements the randomly selected number every time slot (the duration of this basic time slot is specified by the standard). Once the number reaches zero, the station starts to transmit the next frame. If at any point during the countdown, the medium is sensed busy again, the station will store the current counter value and resume counting down once the medium has been sensed free again for the duration of DIFS.

After a station has transmitted a frame, it will wait for an acknowledgement frame (ACK) which is sent by the receiving station and confirms a successful transmission. A missing ACK frame indicates that a problem has occurred (e.g. a collision), in which case the frame has to be retransmitted. The station doubles the currently selected contention window and then chooses a new random number for countdown from the larger interval [1, CW_{double}].

The doubling of the contention window after every failed transmission leads to an automatic increase of the time that each station will wait on average before transmitting the next frame, thereby decreasing the chance of another collision occurring and self-regulating their behaviour to keep the number of collisions low. The standard defines a maximum contention window (CW_{max}) and a minimum contention window (CW_{min}) limiting the range of values that the selected CW can take. After having transmitted a frame successfully, a station resets its contention window to CW_{min}.

Figure 2.1 illustrates the described contention process. In addition to duration DIFS, the standard also defines SIFS (Short Interframe Space) which is (among other uses) the time between the end of a frame transmission and the start of the corresponding ACK frame.

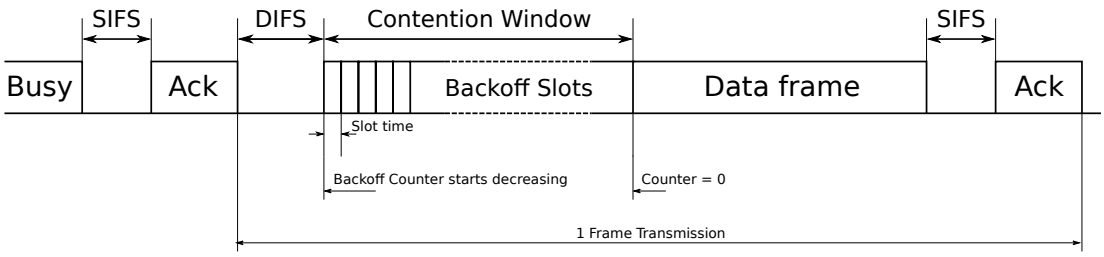

Figure 2.1: 802.11 Contention Mechanism (DCF)

One drawback of the original DCF definition is that all traffic is handled in the same way. In practice, it may be desirable to configure a WLAN to prioritise certain types of transmissions, such as live video and audio streams, compared to less time-critical data like regular file downloads. The next section describes the 802.11e Quality of Service amendment to the 802.11 standard which allows the access point to define different values for CW_{min} and CW_{max} depending on the type of data to be transmitted, a mechanism which will be crucial for both C-VAP and AlphaAP.

2.1.2 Quality of Service Enhancement (EDCA)

IEEE 802.11e is an amendment to the WLAN standard introducing a mechanism for Quality of Service and was published in 2005. The new mechanism allows WLANs to be configured in a way that treats certain types of traffic preferentially.

The amendment defines three different access categories (ACs): Background (AC_BK), Best Effort (AC_BE), Video (AC_VI) and Voice (AC_VO). Stations now maintain separate transmission queues and backoff counters for each AC and the access point adds an additional section containing the EDCA (Enhanced Distributed Channel Access) parameter set in the beacon frame of each VAP. This parameter set allows values for CW_{min} and CW_{max} to be configured for each AC and gives the network operator the opportunity to choose which ACs to prioritise and by how much. Smaller minimum values for the contention window increase the throughput for data of a specific access category, while larger values decrease it relative to other ACs.

Crucially, this mechanism also enables access points to prioritise some virtual networks over others by changing their respective contention parameters. C-VAP and AlphaAP make use of this option in order to dynamically control throughput for VAPs.

The CW_{min} and CW_{max} cannot be set to any arbitrary value and are instead represented in the beacon frame as two 4-bit values referred to as ECW_{min} and ECW_{max}. The corresponding CW value can be calculated using the formula in equation 2.1

$$CW_{min/max} = 2^{ECW_{min/max}} - 1 \tag{2.1}$$

The Quality of Service enhancement additionally defines the value AIFSN which indicates the number of empty slots that stations have to wait for in a certain access category before the backoff process starts. When using regular DCF, stations wait duration DIFS after the medium was sensed free. Instead when using EDCA, stations

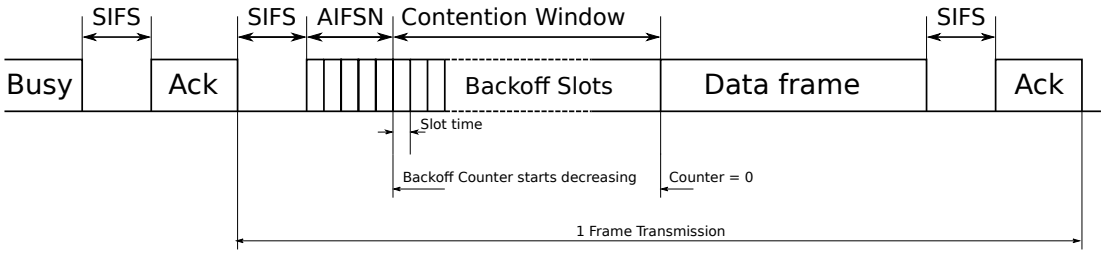

Figure 2.2: 802.11e QoS Contention Mechanism (EDCA)

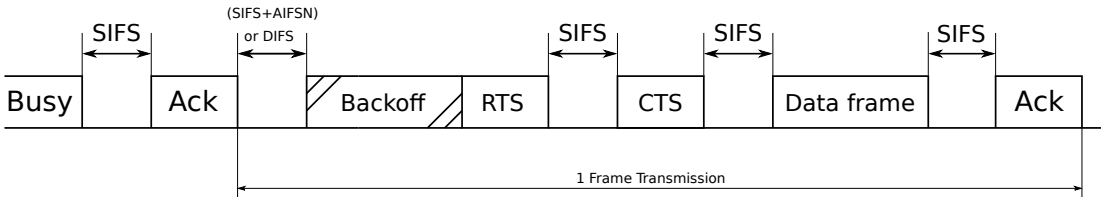

Figure 2.3: 802.11 RTS/CTS Mechanism

wait for the duration of $\text{SIFS} + (\text{AIFSN} \times \text{duration\_time\_slot})$ thereby giving the network operator another opportunity to prioritise data frames in certain access categories over others. The full EDCA contention process is illustrated in Figure 2.2.

The next section discusses the RTS/CTS mechanism whereby stations explicitly exchange information about medium access using a handshake mechanism.

2.1.3 RTS/CTS

The RTS/CTS mechanism (Request To Send / Clear To Send) is an optional mechanism included in the 802.11 standard designed to improve performance in networks where some stations cannot receive the transmissions of all other stations (hidden-node problem). In such a scenario there is a higher chance of a collision occurring because some stations cannot sense accurately whether the medium is free and might start a transmission despite another station being in the transmission process.

When a station uses the RTS/CTS mechanism, it starts by contending for the medium in the usual way (DCF or EDCA). Once the station is allowed to access the medium, it transmits a short RTS frame (request to send) to which the access point responds with a CTS frame (clear to send). The CTS frame informs the initiating station and all other stations that the medium is now reserved for a certain duration. Any station missing the original RTS frame due to being too far away will be able to receive the CTS frame from the access point and learn that the medium has become busy. Following the CTS frame, the initial station transmits the pending data frame. While the RTS/CTS frames add additional overhead, the mechanism can be advantageous because of the often significantly reduced duration of collisions. The described process is outlined in Figure 2.3.

2.2 Frame Aggregation

RTS/CTS frames and the backoff process before every single transmissions add additional overhead to the network operation. Frame aggregation can help to reduce this overhead by packing multiple data frames into a single transmission. Instead of a regular ACK frame, the access point will then respond using a so-called Block ACK acknowledging receipt of the aggregation of frames.

This concludes the discussion of the 802.11 standard. The following sections explore different approaches for how the described contention parameters can be used to optimise the performance of a WLAN and to make various fairness guarantees.

2.3 C-VAP Algorithm

The C-VAP algorithm is, to our knowledge, the first proposed algorithm to provide a fairness guarantee for different VAPs in terms of their uplink throughput and to use the controller for throughput optimisation [3]. Additionally, it does not require any software modifications or extensions on the client side.

The algorithm has two main goals: maximising the total throughput, and providing the same throughput to all VAPs under full load. In operation, an access point running the C-VAP algorithm will continuously measure the probability P_e of a time slot remaining empty and the probability S_i that a slot contains a successful transmission from VAP number i.

These values are then used to compute an error signal for each VAP, taking into account the difference between the individual VAP throughput compared to the average throughput for all other VAPs and the deviation of the measured P_e compared to an ideal setpoint P_e^*. Notably, the optimal value for P_e (in approximation) only depends on the length of an empty slot and the average length of an occupied slot, but not on the number VAPs or stations connected to VAPs [3]. Each error signal is used by the corresponding PI controller to modify the advertised contention window parameter of VAP. A visualisation of the control loop is provided in Section 4.1.

2.4 AlphaAP Algorithm

One limitation of the C-VAP algorithm is the fact that it only allows throughput to be distributed equally between VAPs, but does not consider the case where some VAPs should receive a larger share of the throughput than others. This issue was addressed in [21] where the system model of the algorithm is extended by a set of weights α_i which represent the share of total throughput that VAP i should receive. Apart from this addition, the algorithm works in a very similar manner to C-VAP. The main difference lies in the way the error signal for the PI controller is calculated (based on the introduced weights).

2.5 Other WLAN Optimisation and Fairness Algorithms

In this section, we briefly visit a number of alternative and related projects concerned with optimising throughput or providing fairness guarantees.

Generally, the minimum and maximum contention window are set statically and do not change during operation. Several projects like [2] and [6] examine effects of statically-defined EDCA contention parameter configuration on network performance and make suggestions on how to tune parameters for optimised performance.

Other projects examine ways of changing the EDCA parameter set dynamically in order to tune network operation. For example, the CAC algorithm proposed in [15] aims to optimise network performance by updating the advertised contention parameters at the access point based on the measured probability of a frame transmission colliding with another one. The DAC algorithm [16], in contrast, tunes WLAN performance not at the access point, but by calculating and selecting optimal contention parameters at every client station individually.

SplitAP [5] is an algorithm to solve the problem of providing a fair share of uplink airtime for groups of devices in virtual networks. In order to function correctly, the algorithm requires a software extension to be installed on every device connected to the wireless network.

2.6 Previous Prototyping Efforts

In [19] both the CAC and DAC algorithms were implemented and evaluated on hardware. The entire control mechanism was run in userspace on an off-the-shelf device and frames were captured using the wireless interface's monitor mode and then used to estimate the number of collisions occurring. Using these estimates, the optimal contention parameters were calculated and transferred to the driver. Since capturing and counting frames can be done at the userspace level, neither hardware nor firmware had to be modified.

Important work for modifying the firmware of Broadcom wireless chipsets has been set out in the Nexmon project [18]. Nexmon is a firmware patching framework for Broadcom wireless chipsets. To achieve this, a plugin for the GCC compiler was created which enables the programmer to create patches in the C language and inject C code snippets in the required location of the firmware binary.

Besides Nexmon, the Open Firmware Project [9] provides an open-source firmware for several older Broadcom wireless chips and was created using reverse-engineered documentation special processor developed by Broadcom [4].

This concludes the analysis of related work. The next section introduces the system model for C-VAP in more detail and extends the representation for use with RTS/CTS and frame aggregation.

Chapter 3

System Model

In this section, we look at the C-VAP system model used as a basis for the algorithm analysis in [3] and extend the model to account for an average duration of collisions different to the average duration of successful transmissions. This extension will be used to show how the algorithm can be used together with the RTS/CTS mechanism and frame aggregation (both described in Chapter 2).

For the analysis, we assume ideal channel conditions where transmissions never fail except if a collision occurs. This implies that all stations are close enough to receive each other's transmissions correctly and can therefore sense accurately whether the medium is free. These assumptions were previously shown to approximate performance in experimental settings well [20].

We denote the duration of an empty slot as T_e, the average duration of a slot containing a collision as T_c, and the average duration of a slot containing a successful transmission as T_s.

We assume that the access point is advertising N different VAPs and that n_i stations are connected to VAP i. Each station will only use a single transmission queue (corresponding to one 802.11e access category) and the contention window for this access category will always be set to a single value $CW_i = CW_{i,\min} = CW_{i,\max}$. By setting minimum and maximum value for the contention window equal, we disable the exponential backoff mechanism.

We define τ_i to be the probability that a station connected to VAP i starts transmission during a randomly chosen slot and can express this probability using the contention window as derived in [6] using

$$\tau_i = \frac{2}{1+CW_i}. \tag{3.1}$$

Each slot can either be empty, contain a collision, or contain a successful transmission. Similar to the derivation in [3], we define P_e to be the probability of a slot being empty and write it as

$$P_e = \prod_{k=1}^{N} (1-\tau_k)^{n_k}. \tag{3.2}$$

The probability of a slot containing a successful transmission from VAP i can be written as

$$S_i = n_i\tau_i(1-\tau_i)^{n_i-1}\prod_{k=1,k\neq i}^{N}(1-\tau_k)^{n_k} = \frac{n_i\tau_i}{1-\tau_i}P_e. \tag{3.3}$$

Since we are now looking at a different average duration for collisions and transmissions, we deviate from the analysis in [3] and split the probability of a slot not being empty $(1-P_e)$ into two separate cases: the probability of a slot containing a successful transmission from any VAP, written as

$$P_s = \sum_{k=1}^{N} S_i = P_e\sum_{k=1}^{N}\frac{n_k\tau_k}{1-\tau_k}, \tag{3.4}$$

and the probability of a slot containing a collision, expressed as

$$P_c = 1-P_e-P_s. \tag{3.5}$$

We then adjust the formula for the throughput from [3] taking into consideration the new assumption $T_s \neq T_c$ and obtain

$$R_i = \frac{\mathbb{E}[\text{paylod VAP}_\text{i}/\text{slot}]}{\mathbb{E}[\text{slot length}]} = \frac{S_iL}{P_eT_e+P_sT_s+P_cT_c} \tag{3.6}$$

with L being the number of payload data bytes transmitted during a transmission.

The algorithm objectives remain unchanged from [3]. We aim for all VAPs to have the same throughput ($R_i = R_j \quad \forall i,j$), and for the total throughput to be maximised ($\max\sum R_i$).

After rewriting the first objective as

$$\frac{n_i\tau_i}{1-\tau_i} = \frac{n_j\tau_j}{1-\tau_j}, \tag{3.7}$$

and assuming (as in [3]) $\tau_i \ll 1 \ \ \forall\, i$, we find that the first objective is satisfied when

$$n_i\tau_i = n_j\tau_j. \tag{3.8}$$

In order to address the second objective, we note that under the assumption of the first objective being achieved, maximising total throughput is equal to maximising throughput of an individual VAP.

To analyse how the throughput of an individual VAP can be maximised, we simplify the expression for throughput R_i of VAP i using the simplification from equation 3.8 which gives

$$R_i \approx \frac{n_i\tau_iP_eL}{P_eT_e+P_sT_s+P_cT_c} = \frac{n_i\tau_iL}{T_e+\left(\sum_{k=1}^{N}n_i\tau_i\right)T_s+\left(\frac{1}{P_e}-1-\sum_{k=1}^{N}n_i\tau_i\right)T_c}, \tag{3.9}$$

and obtain after some rearranging an expression for the throughput

$$R_i \approx \frac{n_i \tau_i L}{\left(\prod_k (1-\tau_k)^{-n_k}\right) T_c - \left(\left(1+\sum_{k=1}^{N} n_i \tau_i\right) T_c - T_e - \left(\sum_{k=1}^{N} n_i \tau_i\right) T_s\right)}. \tag{3.10}$$

Using the approximation $P_e \approx e^{-\sum_{k=1}^{N} n_i \tau_i}$ we obtain

$$R_i \approx \frac{n_i \tau_i L}{e^{\sum_{k=1}^{N} n_i \tau_i} T_c - \left(\left(1+\sum_{k=1}^{N} n_i \tau_i\right) T_c - T_e - \left(\sum_{k=1}^{N} n_i \tau_i\right) T_s\right)}. \tag{3.11}$$

Taking into account the condition from the first objective, we can further substitute $\sum_{k=1}^{N} n_i \tau_i = N n_i \tau_i$ to receive the following expression

$$R_i \approx \frac{n_i \tau_i L}{e^{N n_i \tau_i} T_c - \left(\left(1+N n_i \tau_i\right) T_c - T_e - \left(N n_i \tau_i\right) T_s\right)}. \tag{3.12}$$

To determine the value for τ_i maximising the throughput, we calculate the root of the partial derivative

$$\frac{\partial R_i}{\partial \tau_i} = 0 \tag{3.13}$$

which leads to the following non-linear equation

$$n_i \left[T_c e^{N \tau_i n_i} - (T_c - T_e)\right] - (n_i \tau_i) T_c e^{N \tau_i n_i} N n_i = 0. \tag{3.14}$$

Notably, this is the same equation as obtained in [3] when T_o (duration of an occupied slot containing either a collision or a successful transmission) is replaced by T_c. It can be solved in a similar way by using a Taylor expansion and ignoring τ_i terms of a higher than second order which leads to τ_i^* for optimising the throughput

$$\tau_i^* = \frac{1}{N n_i} \sqrt{\frac{2 T_e}{T_c}}, \tag{3.15}$$

and to a corresponding optimal contention window of

$$CW_i = \frac{2}{\tau_i^*} - 1. \tag{3.16}$$

This means that the same optimal value for P_e still applies as well:

$$P_e^* \approx \prod_k e^{-\frac{1}{N}\sqrt{\frac{2T_e}{T_c}}}. \tag{3.17}$$

Following the further control theoretic analysis in [3], we conclude that the C-VAP algorithm can be adapted for use in scenarios where $T_c \neq T_s$ by replacing T_o from the original System Model with T_c from the updated system model described in this section.

Chapter 4

Implementation

This chapter starts with a high-level description of the different components for the implementation of C-VAP and AlphaAP on a commercial WLAN router. We introduce the software architecture and outline the choices made during the implementation followed by a more detailed description of each individual part implemented.

4.1 Algorithm Components

Both C-VAP and AlphaAP consist of three basic components: a measurement system sensing the channel conditions, a PI controller which transforms an error signal e_i into an output signal o_i, and finally a mechanism to set the contention window based on the controller output signal. An overview is shown in Figure 4.1. The main difference between C-VAP and AlphaAP lies in the way the error signal is computed from the channel measurements.

Both algorithms use one controller per VAP; the error signal for each controller is calculated based on the goal state (P_e^*) and the channel measurements (P_e, S_i). At the end, the output signal of the controller is converted to one of the 16 possible contention window values (stored as a 4-bit number).

Each of these 3 components need to be implemented in software. In the next two sections, we look at the chosen router for the implementation and discuss its different software layers.

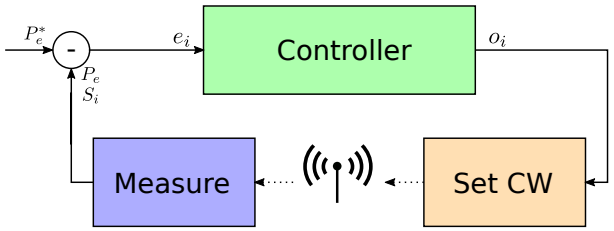

Figure 4.1: C-VAP/AlphaAP Algorithm Control Loop

4.2 Router Asus RT-AC86U

In this project we choose the router Asus RT-AC86U as basis for the implementation. The device is particularly suitable for the experiments carried out because the manufacturer published the usually proprietary firmware source code for the WLAN chipset on its website [11].

The router was released in 2018 and is advertised as a *Gigabit WiFi Gaming Router* [11]. It is compatible with IEEE 802.11a/b/g/n/ac standards and by default supports both the 2.4GHz and 5GHz frequency bands with the option of creating up to three guest networks in each band, which are internally implemented as VAPs. The main CPU of the device is a Broadcom BCM4906 with a dual-core ARM Cortex-A53 processor implementing the 64-bit ARMv8 instruction set. The router also contains two wireless SoCs Broadcom BCM4366E (5GHz) and BCM4365E (2.4GHz) [1].

Public official documentation about the included chipset is very limited, but some reverse-engineered resources are available online, for example the information listed in [4] and the description of Broadcom WLAN chips in [18]. The default firmware of the Asus RT-AC86U is called asuswrt and is used in most of the recent Asus routers. Most of the asuswrt source code is published online which, importantly, also includes the proprietary firmware code for the BCM4366/4365 wireless chipset.

4.3 Software Architecture

The router Asus RT-AC86U consists of several processors: the main processor is an ARM core which runs a version of the Linux operating system and includes the hardware driver for the Broadcom BCM4366/4365 wireless chipset. The wireless chipset itself consists of another ARM core running what we refer to as *firmware* in this document, and an additional proprietary processor known as *D11 core*. This D11 core is responsible for the most time critical tasks and runs code written in a special purpose assembly language often referred to as *uCode* [18], whereas the firmware is tasked with higher level tasks like configuring the Beacon frame and communicating with the Linux driver.

Implementation Layers

In summary, this means there are four different levels in the architecture for the algorithm implementation: *Linux userspace* on the main processor, *wireless driver* also on the main processor, *firmware* on the ARM core of the wireless chipset, and *uCode* on the D11 core. The code for the wireless driver is not included in the source code provided by Asus, the driver will therefore be left unmodified. Modifying the uCode is relatively complicated as well because the code is only available in a raw disassembled form of a custom assembly language without further documentation. Algorithm parts will therefore only be implemented at the level of the uCode if no other layer is able to perform the same task.

This leaves the Linux userspace layer and the firmware layer for implementing the largest parts of the algorithm. Code at both of these layers is written in the C language.

Userspace programs are simply compiled to executable files which can be run at any time. In contrast, the firmware code is compiled into a kernel object (packed together with the driver) which can be loaded by the Linux operating system, but requires a restart of the whole wireless system, making it less flexible. Firmware code can exchange data with the uCode through shared memory and with the main CPU through a mechanism called ioctl (described in more detail in Section 4.4). Programs on the main CPU can use floating point operation which are not supported for the firmware code.

PI controller

The PI controller part (see Figure 4.1) requires a number of multiplications to be carried out. While these could in principal be implemented using fixed point numbers at the firmware level, access to floating point multiplications simplifies the program development. The controller will therefore be implemented at the userspace level, which leads to the additional benefit that various settings like controller parameters can be changed more easily (e.g. using command line arguments) compared to the firmware.

Setting Contention Parameters

Contention window parameters are advertised in the beacon frame sent out by the access point. These beacon frames are managed at the firmware level which means that this component is best implemented at the firmware layer. Communication with the controller running in userspace is done through the ioctl mechanism which will be discussed in Section 4.4.

Measurement System

The measurement component is tasked with observing the channel conditions. This is a time-critical task which requires, for example, the counting of WLAN time slots with a duration of only a few microseconds. This is a major challenge and would be very difficult to implement at the firmware layer. Instead, this task is best suited for the uCode since time slot management is performed by the D11 processor.

As we will see in Section 4.5, the raw time slot counts need to be processed before they can be used by the PI controller to apply different corrections. We want to minimize any changes to the uCode and therefore any further processing will be done at the level of the firmware (which can access the counted values from the uCode using shared memory) and the userspace. This leaves the measurement system distributed over multiple layers.

An overview of the different processors, software layers and the mapping of algorithm components onto those layers is shown in the architecture diagram in Figure 4.2. This concludes the general description of the architecture; the next few sections will discuss the implementation of each subcomponent from Figure 4.2 in more detail.

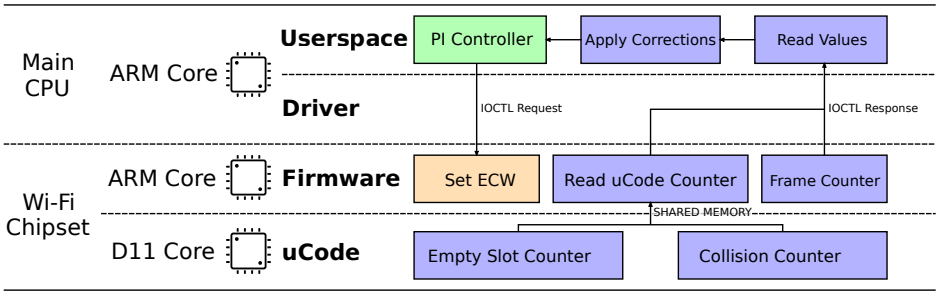

Figure 4.2: Architecture Diagram of C-VAP/AlphaAP Implementation

4.4 Firmware

As outlined in the previous section, when talking about *firmware*, we are referring to the code running on the ARM core in the wireless chipset BCM4366/4365 of the Asus router. The firmware can communicate with the Linux WLAN driver on the main processor using ioctl calls and with the D11 core by directly accessing the D11's special purpose registers and the shared memory.

Communication using ioctl

The term *ioctl* is short for input/output control and commonly refers to a mechanism for configuring input/output devices using a system call provided by the operating system. The ioctl system call allows programs to access functionality that is not well represented by any of the standard Unix operations (write, read, open, close). Which specific capabilities are offered is defined by the hardware driver of the device.

For network interfaces managed by the Broadcom wireless driver (BCMDHD), the ioctl implementation usually passes a message from the driver on the main processor to the firmware on the wireless chipset. The firmware will then execute the requested command and has the option to respond with a return message.

For this project, several custom ioctl calls were defined in order to configure the contention parameters, to retrieve the current measurements of the observed channel conditions, and to print the contention parameters and measurements to the system console for debugging purposes. The custom ioctl commands are listed in Table 4.1.

Ioctl Command	Description
ECW	Set EDCA parameter set of specified VAP for all access categories.
READ_PID	Return channel measurements and reset measurement counters for next cycle.
PRINT_ PARAMETER	Print EDCA parameter set and current channel measurements to system console.

Table 4.1: Custom ioctl commands

Setting Contention Parameters

The EDCA parameter set (refer to Section 2.1.2) allows us to configure contention parameters for every VAP and thereby control how aggressively stations are accessing the channel. Since we do not want to make use of the different access categories, the parameters for each access category of a VAP will be set to the same value. Furthermore, as described in Chapter 3, CW_{min} will be set to the same value as CW_{max} to disable the exponential backoff mechanism.

The fact that the contention window is stored as a 4-bit value means that the PI controller is not able to set each VAP to the optimal contention window value in most circumstances. Instead, we expect the selected contention window to oscillate between the next-smallest and next-largest value around the optimal point.

When the firmware receives an ioctl command `IOCTL_ECW`, the general handler function for ioctl commands `wlc_ioctl()` will be called in the firmware. Another case is added to this handler (see Listing 4.1) which will update the beacon with the new contention window value. In order to do this, the new handler case uses the data structure of type `wlc_bsscfg_t` corresponding to the network interface for the requested VAP. It updates the parameter set count in this data structure, which is important since devices connected to the network may only be checking the beacon contention parameters when a change is detected in the parameter set count. Afterwards, the three parameters ECW, TXOP and AIFSN are extracted from the ioctl request buffer and used to update the same parameters in the `wlc_bsscfg_t` data structure for all access categories. Finally, a call of `wlc_bss_update_beacon()` instructs the firmware to update the beacon with the new parameters.

Counting received frames

The purpose of the measurement system (Figure 4.1) is to obtain estimates for the probability of a slot being empty (P_e) and the probability of a slot containing a successful transmission from VAP i (S_i). A slot can either be an empty slot, a collision, or a successful transmission. Therefore, we calculate an estimate for S_i using

$$S_i = \frac{\text{Number of received frames for VAP } i}{\text{Total number of slots}}, \tag{4.1}$$

where the total number of slots for a time period is given as

$$\text{Total slots} = \text{Empty slots} + \left(\sum_{i \in \text{VAPs}} \text{Frames VAP } i \right) + \text{Collisions} \tag{4.2}$$

In the chipset, all received frames are sent from the D11 processor to the firmware and processed in `wlc.c` and `wlc_rx.c`. For counting frames, a function `increment_frame()` is added to the firmware which takes the VAP number as argument and is called inside `wlc_rx.c` and `wlc.c` where the received frames can be inspected to check the VAP they belong to before incrementing the corresponding count.

```
case IOCTL_ECW:
{
    uint8 *buffer = arg;

    /* increment parameter set count */
    uint8 parameter_set_count =
        cfg->wme->wme_param_ie_ad->qosinfo & 0x0f;
    parameter_set_count = (parameter_set_count + 1) & 0x0f;
    cfg->wme->wme_param_ie_ad->qosinfo &= 0xf0;
    cfg->wme->wme_param_ie_ad->qosinfo |= parameter_set_count;

    /* extract EDCA parameters from ioctl input */
    uint8 ecw = buffer[IOCTL_ECW_BUFFER_ECW_INDEX];
    uint16 txop = buffer[IOCTL_ECW_BUFFER_TXOP_INDEX] |
                  (buffer[IOCTL_ECW_BUFFER_TXOP_INDEX + 1] << 8);
    uint16 aifsn = buffer[IOCTL_ECW_BUFFER_AIFSN_INDEX] & 0xf;

    /* update internal data structure with EDCA parameters */
    int i = 0;
    for (i = 0; i < NUMBER_ACCESS_CATEGORIES; i++) {
        cfg->wme->wme_param_ie_ad->acparam[i].ECW = ecw;
        cfg->wme->wme_param_ie_ad->acparam[i].TXOP = txop;
        cfg->wme->wme_param_ie_ad->acparam[i].ACI =
            (i << 4) | aifsn;
    }

    /* Update Beacon */
    wlc_bss_update_beacon(wlc, cfg);

    bcmerror = 0;
    break;
}
```

Listing 4.1: Handler for ioctl to modify contention window

4.5 uCode

The proprietary D11 core manages the low-level and timing related tasks for receiving and transmitting frames on the medium. Since Broadcom has published no documentation about the D11's instruction set and special registers, we instead refer to reverse-engineered resources online [4]. As discussed in Section 4.3, the D11 core will only be used to count the number of empty slots (because no other processor has access to this time-critical information) and collisions (collisions lead to corrupted data which is therefore not forwarded to the firmware). These counts are required to obtain a measurement for the total number of slots in a certain time period.

Counting empty slots

The D11 core provides a special purpose register `IFS_IDLE_COUNTER` which counts idle slots and starts counting two slots after the medium becomes free [4]. Analysis of the disassembled uCode shows that the register is reset by the processor frequently, but also shows that the reset always occurs at the same place in the code. This makes it

possible to add instructions at this position to accumulate the register count in a shared memory cell before each reset. The firmware code can then read this shared memory cell and obtain the accumulated count for the number of empty slots that have occurred. Additionally, the firmware is able to reset the shared memory cell at any point to reset the counter (this happens after each call of `READ_PARAMETER` ioctl command).

The lines added to the uCode to accumulate the `IFS_IDLE_COUNTER` register are shown in Listing 4.2. Instruction `xadd` is used to add the content of the 16-bit register `IFS_IDLE_COUNTER` (referenced as `SPR_IFS_0x0e` in the code) to the value stored in shared memory location 0x1700. The second line additionally increments the content of memory address 0x1702 in case the first add instructions leads to an overflow and enables us to keep track of the accumulated count using 32-bit instead. This is important since a 16-bit counter would overflow in some scenarios after around 0.3 seconds.

```
1 xadd.   [SHM(0x1700)], SPR_IFS_0x0e, [SHM(0x1700)]
2 xaddc   [SHM(0x1702)], 0x0, [SHM(0x1702)]
```

Listing 4.2: Additional uCode instructions to accumulate IFS_IDLE_COUNTER

Lack of official documentation leads to some uncertainty around the precise behaviour of the counter register. Therefore, experiments were carried out to verify the counted values and compare them with an estimation based on the description of the contention process in Section 2.1.2. For this experiment, only one device was connected to the wireless network and the network was configured with a fixed contention window.

The tests, repeated for different contention window values, show that in contradiction with the reverse-engineered documentation, the register counts all empty slots that are part of the contention window plus approximately three additional slots for each frame transmitted over the channel.

Therefore, in order to obtain a more accurate measure of the empty slots on the channel the following correction mechanism will be applied:

$$\text{Empty slots (corrected)} = \text{IFS\_IDLE\_COUNTER} - 3 * \text{Received Frames} \tag{4.3}$$

Using this correction mechanism the measured empty slots during tests deviate consistently by less than 2% from the estimated count of empty slots across all possible contention windows between 3 (ECW = 2) and 32,767 (ECW = 15). This is compared to a deviation from the estimate between 20% and 200% observed for small contention windows without the correction mechanism.

Counting Collisions

Several approaches for measuring the number of collisions are explored:

1. Count how often the status register on D11 switches from idle to busy.
2. Use the `IFS_BUSY_COUNTER` to measure the total time the medium is busy.
3. Count the number of received frames with a wrong checksum.

For the first approach, we note that according to [4], there exists a status register which indicates whether or not the medium is busy at the current moment. Since every successful transmission and every collision start with a transition from an idle medium to a busy medium, we can attempt to count the number of changes to the status register bit indicating the idle status. Subtracting the number of correctly received frames would then lead to an estimate for the number of collisions occurring. Experiments show that this leads to unreliable results which are difficult to reproduce.

Possible explanations for this behaviour are problems in reliably tracking every change of the register (which may be the case if the iterative loop in the uCode does not execute the counting code frequently enough), or alternatively it might be the case that the actual behaviour of the status register is different from the behaviour according to the (unofficial) documentation [4].

The second approach is based upon the register `IFS_BUSY_COUNTER`, which counts the number of microseconds the medium was busy. Similarly to the counter for empty slots, this register is reset by the uCode frequently but the reset happens always at the same position in the code. This means that the value can be accumulated in the shared memory by adding a corresponding add instruction to this part of the uCode.

Under the assumptions that all successfully transmitted data frames and all collisions occupy the medium for the same amount of time, we can estimate the number of collisions using

$$\text{Collisions} = \frac{\text{Total busy time}}{\text{Average duration of data frame}} - \sum_{i \in \text{VAPs}} \text{Received frames VAP } i. \quad (4.4)$$

While this approach, according to experiments carried out for this project, leads to more reproducible results with a clear correlation between the number of collisions measured and the number of collisions estimated based on the theoretical model, the approach is sensitive to an accurate estimate for the average duration of frames/collisions. This means the approach will not be very reliable in practice and leads to problems particularly when mechanisms like RTS/CTS or frame aggregation are used (which changes the duration of collisions and successful transmissions).

The third mechanism links into the a check of the validity of the checksum carried out in the uCode. We add code to increment a counter in the shared memory whenever the checksum is invalid. This proved to be the most reliable way to estimate the number of collisions based on the experience with RTS/CTS and Frame Aggregation since the count does not depend the duration of any collision.

4.6 Userspace

As described in Section 4.3, two benefits for implementing functionality at the userspace level are the simple access to floating point operations and easier adaptability of the code (new programs can be compiled and executed without having to restart the entire wireless system). For this reason, the PI controller and the logic for processing the

raw measurement values from firmware and uCode are implemented at the userspace level. This section will describe the implemented functionality in more detail.

A good starting point in creating userspace programs for the Asus RT-AC86U can be found in [17] under section *bcm4366c0*, which includes a description of the tools used for compiling *nexutil*, a userspace program developed for the Nexmon project [18]. The userspace implementation for this project makes use of the *libnexmon* library for sending ioctl requests to the wireless firmware.

PI Controller

C-VAP and AlphaAP are based on a separate PI controller run for each VAP. The PI controller's main loop is executed every 500ms. This time was chosen based on the fact that beacon frames are sent out every 100ms and that the WLAN standard recommends not to change the contention parameters for every new beacon frame in order to give devices time to adjust to the new parameters.

In each iteration, the current counts for empty slots, collisions and received frames are requested from the firmware, and used to calculate the error signal for each VAP. The control loop then generates an output signal per VAP which will be converted and rounded to the nearest possible contention window for the VAP. These calculated contention window values are then transferred to the firmware via additional ioctl calls.

The operation of the controller program can be configured using the command line arguments described in Table 4.2. The controller parameters (K_P, K_I and P_e^*) can be configured. Note that also the number of devices connected to each VAP has to be set. While this information could be extracted from the firmware by keeping track of association and disassociation information, this feature was omitted due to time constraints. The program offers output in JSON format to allow for machine processing of the measured data. It is also possible to configure a minimum value for the ECW set at VAPs. Very low contention window values led to unreliable behaviour during experiments for this project (a minimum value of ECW = 2 avoids most unreliable behaviour while giving the controller a large enough range to operate; similar assumptions have been made in [3]).

Listing 4.3 shows the function used to transfer the EDCA parameters to the firmware. Values ECW, TXOP and AIFSN are copied into a buffer array and function `nex_ioctl()` from libnexmon is used to call the custom ioctl command for setting the ECW. The `nex_ioctl()` function expects a `nexio` data structure which can be obtained using the `nex_init_ioctl()` function (also part of libnexmon). Internally, this init function opens a socket for the corresponding VAP interface and any ioctl commands will then be called with reference to this socket.

The general functionality of the PI controller will be demonstrated using pseudo code, in order to focus on what the code does while removing some distractions based on the specifics of the C language.

Listing 4.4 shows the general structure of the program: at the start, the integral error signals will be set to zero for all VAPs; then the program enters an infinite loop to read

Option	Description
--time	Set time interval of one control loop iteration (in ms)
--devices[1-4]	Flags for setting number of devices connected to each VAP
--alpha[1-4]	Flags for setting AlphaAP throughput weights for each VAP
--kp	Configure controller parameter K_P
--ki	Configure controller parameter K_I
--pe_star	Configure desired probability of time slot being empty (P_e^*)
–-json	Enable output in JSON format for better automated machine-readability
--duration	Set duration (in seconds) until controller automatically terminates
--min_ecw	Set minimum value for ECW. Contention parameters will never be set below this value. Useful in order to prevent very small ECW values that easily block any network access through constant collisions

Table 4.2: Command Line Options for PI controller tool

the measurement (`read_values()`), to apply certain corrections to the raw measurements (`process_values()`), to run the control loop (either `run_controller_cvap()` or `run_controller_alpha_ap()`), and finally to set the contention window parameter for all VAPs (`set_contention_window()`).

Raw measurements are provided by the firmware through an ioctl call as described in Section 4.4. To obtain the latest measurements, the function `read_values()` (Listing 4.5) calls the ioctl in userspace which will be forwarded through the Linux driver to the wireless chip firmware. The firmware will send a response message containing the number of empty slots, collisions and frames per VAP counted since the last time the values were read by the controller program.

```
1 function read_values():
2     call_ioctl()
3     extract_values_from_response_buffer()
4     return empty_slots, collisions, frame_counts
```

Listing 4.5: Pseudo code for read_values()

C-VAP and AlphaAP require probability values P_e and S_i (for every VAP) as input. Function `process_values()` (Listing 4.6) calculates these probabilities using the raw measurements obtained through `read_values()`. As described in Section 4.5, the

```
void set_ecw_value(struct nexio *nexio, uint8_t ecw) {
    uint8_t buffer[BUFFER_SIZE];

    uint16_t txop = 0x00;
    uint8_t aifsn = 0x02; // Minimum allowed value according to ↩
        standard

    memset(buffer, 0, BUFFER_SIZE);
    buffer[0] = ecw;
    buffer[1] = txop & 0xff;
    buffer[2] = (txop >> 8) & 0xff;
    buffer[3] = aifsn;
    nex_ioctl(nexio, IOCTL_ECW, buffer, BUFFER_SIZE, true);
}
```

Listing 4.3: Function for setting EDCA parameters of a VAP

```
for i in 1 to N:
    error_integral[i] = 0
while True:
    read_values()
    process_values()
    run_controller_cvap() # for AlphaAP: run_controller_alpha_ap()
    set_contention_window()
```

Listing 4.4: Pseudo code for PI controller

measured number of empty slots is not completely accurate and three slots need to be subtracted for each frame received to obtain a more precise measurement.

The function calculates the total number of slots (empty slots together with one slot per received frame and one slot per collision). This number is then used together with count for empty slots and for frames received to obtain an estimate for P_e and S_i.

```
function process_values():
    total_frames = sum(frame_counts)
    empty_slots_corrected =
        empty_slots + CORRECTION_FACTOR * total_frames
    total_slots =
        empty_slots_corrected + total_frames + collisions
    Pe = empty_slots / total_slots
    for i in 1 to N:
        S[i] = frame_counts[i] / total_slots
    return Pe, S
```

Listing 4.6: Pseudo code for process_values()

The control loop for the C-VAP and AlphaAP algorithms requires the processed measurements calculated by `process_values()` in addition to the goal state P_e^*. For each VAP, the control loop function `run_controller_cvap()` (Listing 4.7) is calculating an error signal based on the deviation of the measured P_e from the aim P_e^* and based on the measured share of the throughput for a VAP compared to its allocated share.

The obtained error signal is then added to the integral of the error signal and both error and integral together with controller parameters K_P and K_I are used to obtain an output value. Finally, the output value multiplied by the number of devices connected to the VAP gives the contention window.

```
function run_controller_cvap():
    error_optimal = Pe* - Pe
    for i in 1 to N:
        error_fair[i] = (N - 1) * S[i] - (sum(S) - S[i])
        error[i] = error_optimal + error_fair[i]
        error_integral[i] += error_integral[i]
        output[i] = Kp * error[i] + Ki * error_integral[i]
        CW[i] = output[i] * connected_devices[i]
    return CW
```

Listing 4.7: Pseudo code for run_controller_cvap()

Function `run_controller_alpha_ap()` (Listing 4.8) is operating in a similar way as `run_controller_cvap()`. Again, an error signal is calculated for each VAP, but this time the weight α_i is taken into consideration as well. The error signal is again used to calculate an output signal and the contention window value for every VAP.

```
function run_controller_cvap():
    error_optimal = Pe* - Pe
    for i in 1 to N:
        error_fair[i] = S[i] / alpha[i] - sum(S)
        error[i] = error_optimal + error_fair[i]
        error_integral[i] += error[i]
        output[i] = Kp * error[i] + Ki * error_integral[i]
        CW[i] = output[i] * connected_devices[i] / alpha[i]
    return CW
```

Listing 4.8: Pseudo code for run_controller_alpha_ap()

Finally, the value for the contention window needs to be transferred to the wireless chipset. Ultimately, the contention window can only be set to 16 different values (ECW). Function `set_contention_window()` (Listing 4.9) receives the contention window calculated by the control loop and rounds it to the closest ECW value. The ECW value for each VAP is transferred to the wireless chip using the corresponding ioctl call.

```
function set_contention_window():
    for i in 1 to N:
        ECW = round_to_integer(log2(CW[i] + 1))
        ECW = clamp(ECW)
        send_ioctl(ECW)
```

Listing 4.9: Pseudo code for set_contention_window()

Chapter 5

Test Setup

The test setup for evaluating the algorithm performance consists of a router Asus RT-AC86U, a server to receive data, and a number of clients (8 Raspberry Pi devices) to send data. The router advertises several VAPs (up to four) and the Raspberry Pi clients (mix of models 3B+ and 4) are each connected to one of the VAPs using their default internal wireless interface. The server is connected to the router using a wired Ethernet connection. An overview of the test setup is given in Figure 5.1.

Raspberry Pi devices are used for the throughput measurements because of their common availability and low price which enables the creation of a test bed with a larger number of devices of the same type than would otherwise be possible. Using the same device type makes it more likely that all devices have the same behaviour and that any measured differences are not the result of a different implementation of the WLAN standard.

Throughput measurements are done using the network bandwidth measurement tool iperf3. The program is started on the server in server mode to receive data from different clients (using different ports). Each Raspberry Pi then runs iperf3 in client mode, continuously transferring data to the server at the highest speed possible. After the test has finished, the average throughput per device can be calculated from the reported number of received data frames at the server.

Tests are run for 100 seconds with 10 iterations each time; the mean throughput over these 10 iterations is ultimately reported. A test duration of 100 seconds is chosen to provide a balance between giving the algorithm enough time to work properly and keeping the total time required to collect results manageable. Tests show that the results after a duration of 100 seconds are very similar compared to results obtained after a test duration of 300 seconds. Before each test, the Raspberry Pi devices are connected to the correct VAP depending on the desired network configuration.

5.1 Router Configuration

The router is configured to operate in AP (Access Point) with three Guest Networks set up in the 5GHz frequency band.

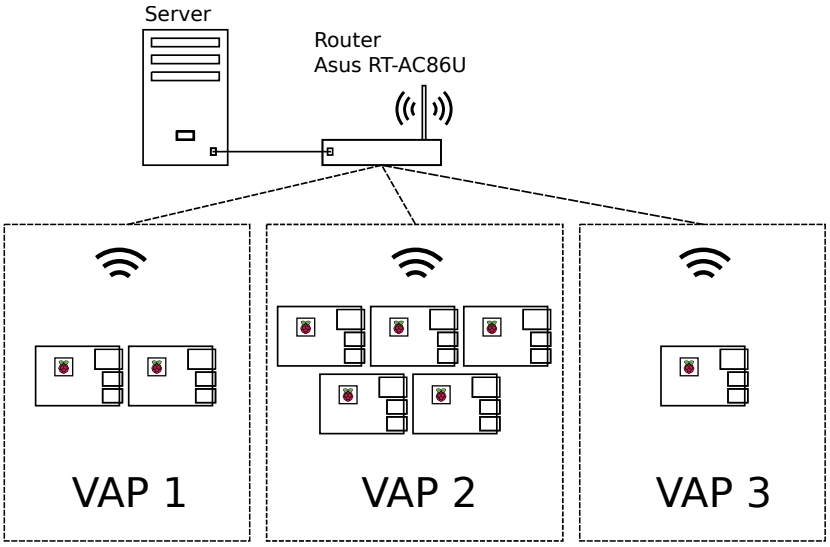

Figure 5.1: Test Setup for Throughput Measurements

Before running tests, a configuration script (shown in Listing 5.1) is executed which loads the custom firmware (from file dhd.ko) and limits allowed bitrates to 54Mbps only to ensure that all frames are being sent at the same bitrate. If this was not limited, the network devices can vary the bitrate over time depending on the channel conditions, which makes it difficult to compare throughput measurements. The guest networks are separated into different data link layer bridges. This is required in order to ensure proper separation between different VAPs.

```
#!/bin/bash

# Load modified firmware driver
/sbin/rmmod /jffs/dhd.ko
/sbin/insmod /jffs/dhd.ko

/sbin/restart_wireless

# VAP 1
/usr/sbin/wl -i wl1.1 down
/usr/sbin/wl -i wl1.1 rateset 54b       # Limit to OFDM rate 54
/usr/sbin/wl -i wl1.1 rateset -v 0      # Turn off VHT
/usr/sbin/wl -i wl1.1 rateset -m 0      # Turn off HT
/usr/sbin/wl -i wl1.1 up

# VAP 2
/usr/sbin/wl -i wl1.2 down
/usr/sbin/wl -i wl1.2 rateset 54b       # Limit to OFDM rate 54
/usr/sbin/wl -i wl1.2 rateset -v 0      # Turn off VHT
/usr/sbin/wl -i wl1.2 rateset -m 0      # Turn off HT
/usr/sbin/wl -i wl1.2 up

# VAP 3
/usr/sbin/wl -i wl1.2 down
/usr/sbin/wl -i wl1.2 rateset 54b       # Limit to OFDM rate 54
```

```
/usr/sbin/wl -i wl1.2 rateset -v 0       # Turn off VHT
/usr/sbin/wl -i wl1.2 rateset -m 0       # Turn off HT
/usr/sbin/wl -i wl1.2 up

# Separate bridge for VAP 1
brctl addbr br1
brctl delif br0 wl1.1
brctl addif br1 wl1.1

/sbin/ifconfig br1 10.0.1.1 netmask 255.255.255.0 up

# Separate bridge for VAP 2
brctl addbr br2
brctl delif br0 wl1.2
brctl addif br2 wl1.2

/sbin/ifconfig br2 10.0.2.1 netmask 255.255.255.0 up

# Separate bridge for VAP 3
brctl addbr br3
brctl delif br0 wl1.3
brctl addif br3 wl1.3

/sbin/ifconfig br3 10.0.3.1 netmask 255.255.255.0 up

# Disable hardware switching between VAPs
ethswctl -c hw-switching -o disable

# Enable routing between LANs
echo 1 > /proc/sys/net/ipv4/ip_forward
```

Listing 5.1: Router configuration script

5.2 Automated Testing Scripts

In order to simplify data collection, a set of Python scripts was created to automatically connect to all devices (server, router, clients) via SSH, connect the clients to the right VAP and start the required programs for throughput measurement. At the end of each test the resulting data is collected and stored (in JSON format) for further processing and visualisation.

The scripts are able to read an input file containing the test parameter information about the tests to be carried out. In the case of an error (for example if a device was unable to connect to the wireless network and is unavailable), the test scripts can independently restart the entire test setup and continue at the last successfully completed test.

5.3 Control Parameter Estimation

Running a test requires us to specify values for the controller parameters P_e^*, K_P and K_I. The analysis in [3] comes with formulas depending on T_e and T_o (which we substitute

here by T_c as discussed in Chapter 3). For tests in this project, we use the UDP transport layer protocol, which is particularly suitable because of its simplicity and its lack of any sophisticated throughput regulation mechanisms, which could interfere with C-VAP or AlphAP. The payload size is 1000 Bytes; together with UDP Header (8 bytes), IP Header (20 bytes) and Ethernet header and tail (total 40 bytes), we obtain a total frame size of 1066 bytes. For OFDM data rate of 54Mbps, the required transmission time is 180us.

The duration of an average collision (T_c) is calculated, according to the standard, using the duration of the frame and the ACK timeout value [10]. The ACK timeout value is calculated using SIFS (16us), plus a single slot time (9us), plus a start delay (20us) [10]. This leads to an average duration of a collision of $T_c = 225$us.

The results in the estimated control parameters for C-VAP (and AlphaAP, since the analysis in [21] is equivalent) are:

$$P_e^* \approx e^{-\sqrt{\frac{2T_e}{T_o}}} \approx 0.75$$

$$K_P = 0.4\frac{T_o}{P_e^* T_e} \approx 13.3$$

$$K_I = \frac{0.2}{0.85}\frac{T_o}{P_e^* T_e} \approx 7.8$$

RTS/CTS

When RTS/CTS or frame aggregation is enabled, the average duration for a collision changes substantially compared to the non RTS/CTS case and is calculated using the duration of an RTS frame together with the CTS timeout value. An RTS frame consists of 20 bytes which corresponds to a duration of 24us at a data rate of 54Mbps. Usually, RTS frames would be transmitted at a maximum data rate of 24Mbps, but here, due to the limitation of the router to only accept 54Mbps frames, RTS frames are also transmitted at 54Mbps. The CTS timeout value is defined in the same way as the ACK timeout value and is calculated using SIFS (16us), plus a single slot time (9us), plus a start delay (20us). This leads to an average duration of a collision of $T_{c,RTS} = 69$us.

The resulting estimated control parameters for RTS/CTS are:

$$P_{e,RTS}^* \approx e^{-\sqrt{\frac{2T_e}{T_o}}} \approx 0.60$$

$$K_{P,RTS} = 0.4\frac{T_o}{P_e^* T_e} \approx 5.1$$

$$K_{I,RTS} = \frac{0.2}{0.85}\frac{T_o}{P_e^* T_e} \approx 3.0$$

Chapter 6

Evaluation

This chapter provides results and interpretation for a range of experiments using the C-VAP and AlphaAP implementation on the test bed described in Chapter 5.

6.1 C-VAP Performance

First, we evaluate the performance of the C-VAP algorithm without RTS/CTS and frame aggregation. This corresponds to a similar scenario as the one simulated in [3]. The test parameters are outlined in Table 6.1 with each test having a duration of 100 seconds and the results given as the mean over a sample size of 10.

Transport Layer Protocol	UDP
Bitrate	54 Mbps
Payload per frame	1000 Bytes
Test duration	100s
Sample Size	10
K_P	13.27
K_I	7.81
P_e^*	0.75

Table 6.1: Test Parameters C-VAP Performance Measurement

We obtain the result visualised in Figure 6.1, with the first row showing the experiments run using the C-VAP algorithm for different numbers of devices and the second row showing the same experiment using the default EDCA contention mechanism using the default static contention window of $\mathrm{CW}_{\min} = 15$ and $\mathrm{CW}_{\max} = 1023$.

As we expect from the theoretical analysis, the results confirm that the EDCA mechanism distributes throughput approximately equal between devices, leading to very unequal results on a per-VAP basis. In contrast, the C-VAP algorithm consistently achieves as similar throughput and in some cases outperforms regular EDCA, while

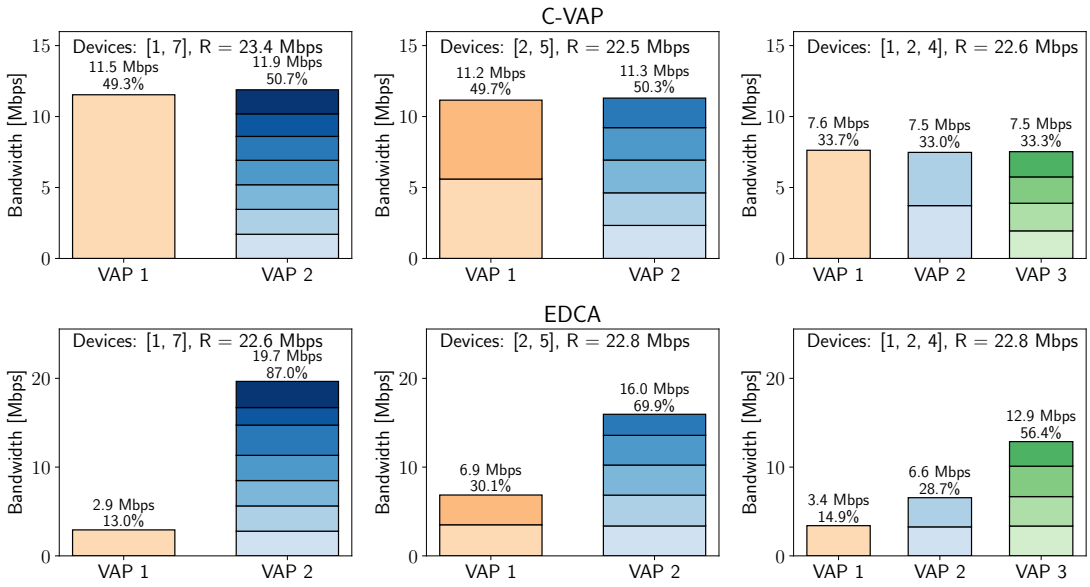

Figure 6.1: Throughput of EDCA contention and C-VAP algorithm in comparison

also distributing the total throughput equally across different VAPs with a deviation of less than 1.5%.

While in [3] the C-VAP algorithm always provides a higher total throughput than EDCA, this is only the case here in one out of three settings. One possible reason for this difference is the fact that in this project the contention window is limited to only 16 different values (4-bit number) whereas [3] allowed a more fine-grained selection of the contention window. Depending on the number of devices in each VAP, this means that the optimal contention window value cannot be set directly, but that the algorithm has to oscillate between the two closest possible contention windows and therefore not operate at the optimal point.

6.2 AlphaAP Performance

Next, we evaluate the performance of the AlphaAP algorithm and compare it to EDCA and C-VAP. We run a similar set of experiments as in the previous section and choose the AlphaAP weights to allocate a larger proportion of the throughput to the VAPs with a smaller number of devices in order to demonstrate the algorithm's effect compared to regular EDCA operation most clearly.

The results are shown in Figure 6.2 and demonstrate that, compared to C-VAP and EDCA, the AlphaAP algorithm achieves a similar and, in two out of three experiments, higher total throughput. It manages to distribute total bandwidth on a per-VAP basis according to the weights set in the algorithm with again a deviation of less then 1.5% compared to the desired weights.

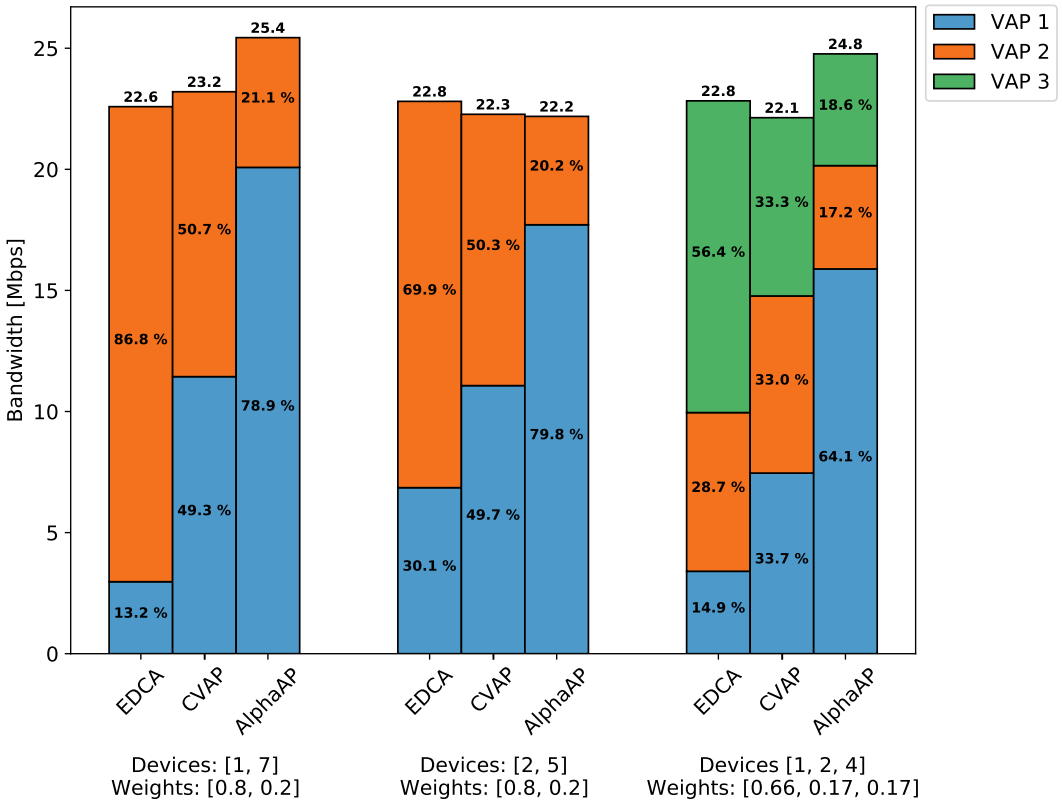

Figure 6.2: Throughput of AlphaAP, C-VAP and EDCA in comparison

6.3 Control Parameter Optimisation

In this section, we examine the impact of the controller parameters on the performance of the C-VAP algorithm. Again, samples are taken with a test of duration 100 seconds, the remaining parameters that are not modified for the current graph are set to the estimated optimum from Section 5.3.

Controller Setpoint: P_e^*

The P_e^* parameter represents the point of operation that the controller is aiming to achieve. It should be set to a well-balanced value in order to achieve a high total throughput. If the value is too high, the channel is mostly empty, and if it is too low, too many collisions will occur on the channel. Somewhat surprisingly, the derivation in [3] shows that the approximation of P_e^* is (at least in its approximated form) independent of the number of devices connected to the network and only depends on the average duration of a collision and the duration of an empty slot.

We measure throughput across a range of values for P_e^* and present the results in Figure 6.3. Maximum throughput achieved was 23.1 Mbps for $P_e^* = 0.8$ compared to 22.4 Mbps achieved for the estimated optimal value of $P_e^* = 0.75$. A possible reason for the disparity could either lie in an imperfect measurement of empty slots and collisions (both of which impact the measured P_e) or in the fact that some of the mathematical

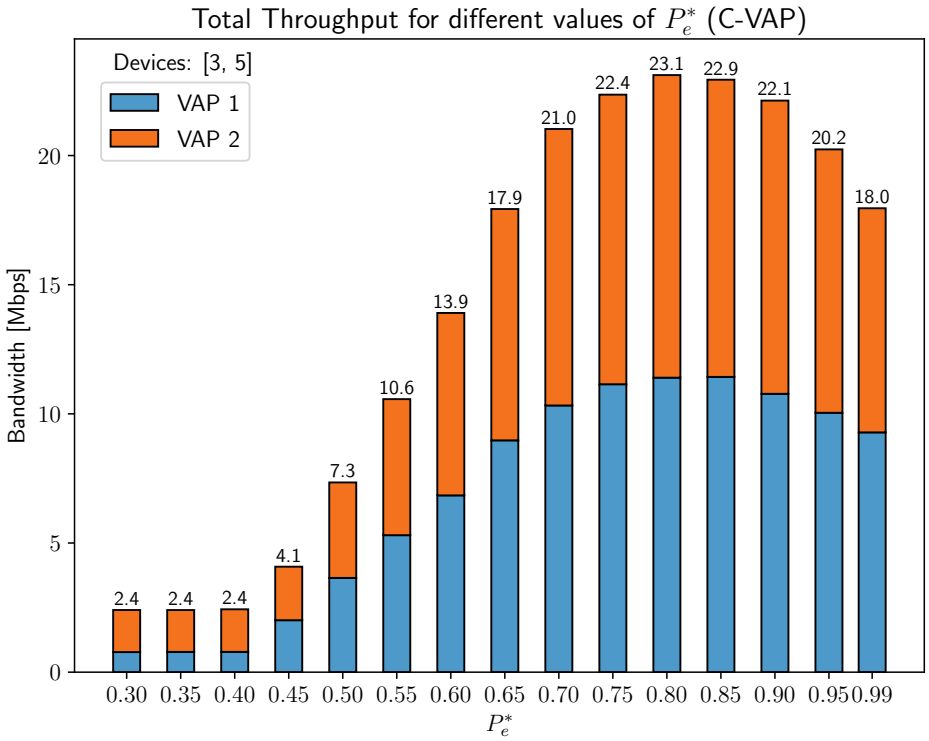

Figure 6.3: Throughput achieved for different P_e^* values

simplifications made in [3] are less accurate for a smaller number of devices.

Controller Integration Part: K_I

Similarly to the optimization of P_e^*, we carry out experiments using different values for the integral part of the PI controller K_I. The results shown in Figure 6.4 indicate that a throughput close to the optimum can be achieved in the range of around $K_I = 5$ up to $K_I = 50$. Outside this range the performance begins to deteriorate. The estimated value of $K_I = 7.81$ lies within the optimal range, but is located more closely towards the lower end of the range and a slightly higher value might enable the controller to adjust to changes faster while still maintaining stable behaviour.

Additionally, we examine the behaviour of the algorithm using a timing diagram. For the estimated optimal value ($K_I = 7.81$) we observe a reasonably quick introduction phase of around 5 to 10 seconds until the controller operations reaches the operating point. From this point on the contention window values keeps oscillating around the optimal value, causing the ECW values to flip frequently (Figure 6.5).

When choosing a value for K_I 10 times smaller than the estimate, the controller reacts much more slowly to any deviations in channel state from the setpoint. The final point of operation is only reached after around 60 seconds and it takes a similar amount of time until the total throughput reaches the desired optimum. This is demonstrated in Figure 6.6.

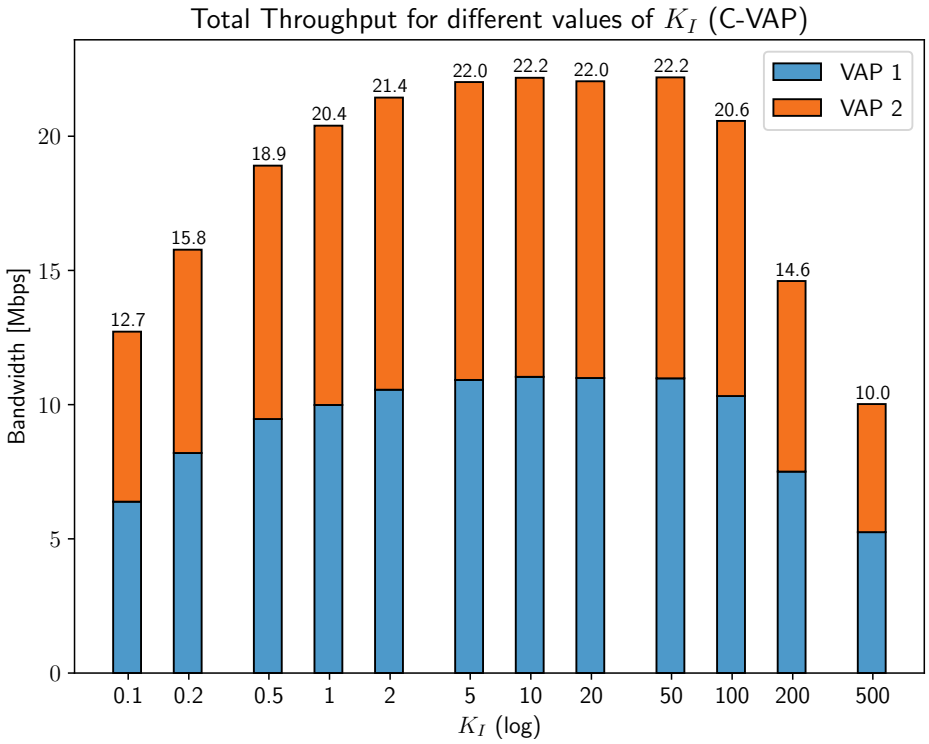

Figure 6.4: Throughput for different values of K_I

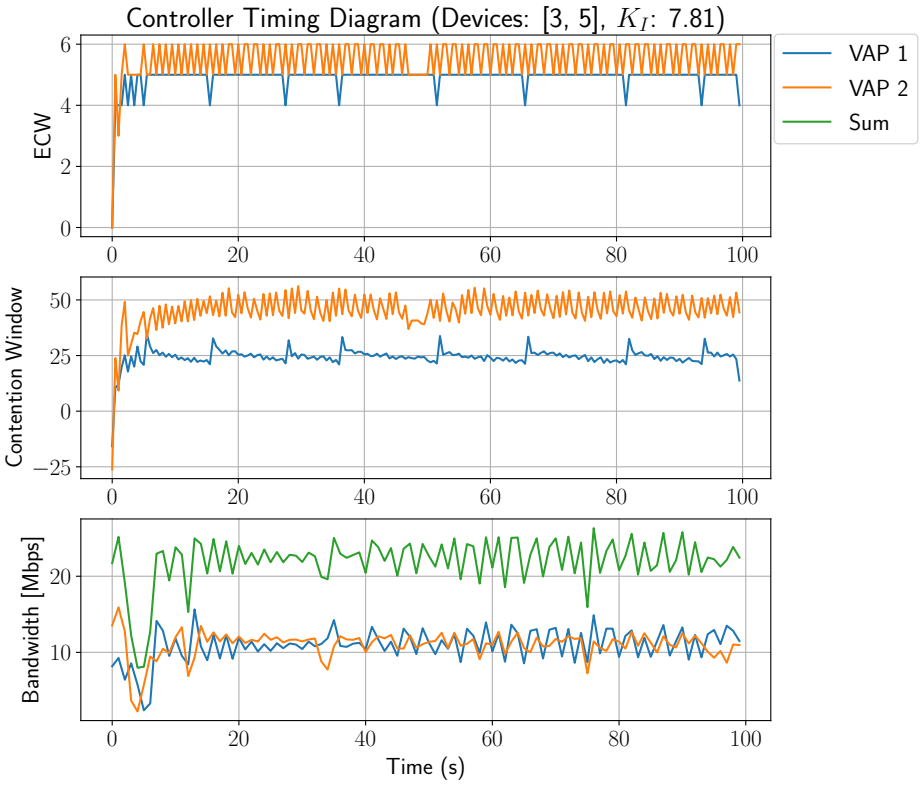

Figure 6.5: Timing diagram for estimated K_I

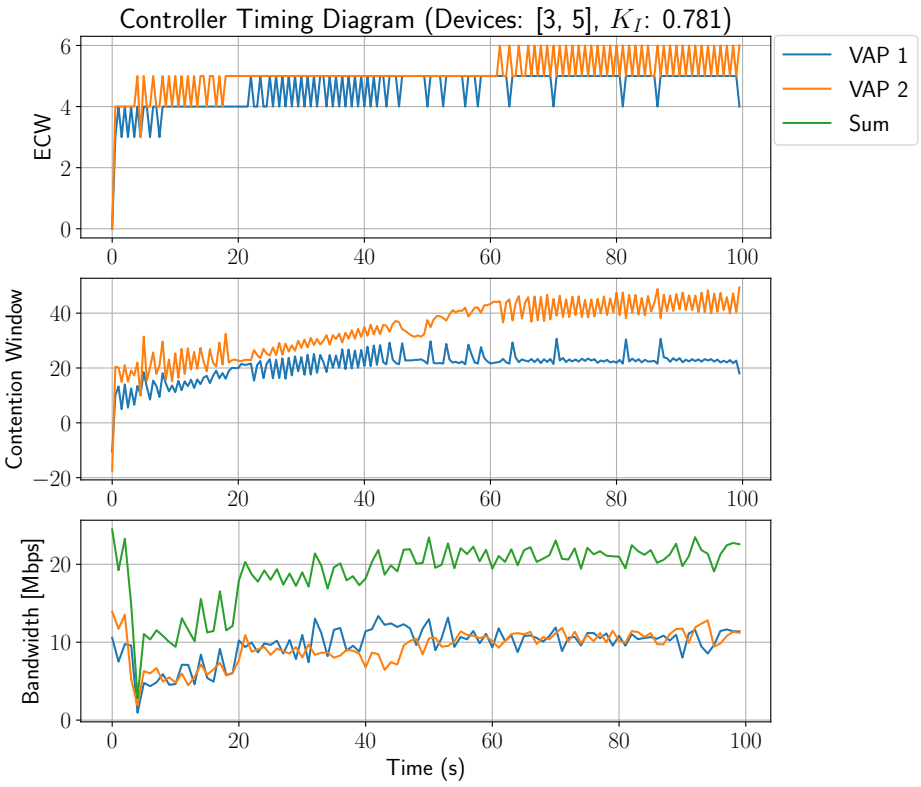

Figure 6.6: Timing diagram for $K_I/10$

Controller Proportional Part: K_P

We now repeat a similar experiment for the controller parameter K_P with the results shown in Figure 6.7. Interestingly, while high values for K_P, starting at around 50, lead to unstable behaviour and a deterioration of the total throughput, very low values of K_P do not seem to have a large impact on the performance. In those cases the integral part of the controller will ensure a good point of operation is reached eventually.

In terms of timing diagrams, we see a relatively stable behaviour for the estimate of $K_P = 13.27$ (see Figure 6.8). Again, the contention window values oscillate around the optimal values for each VAP. In contrast, increasing K_P by a factor of 10 causes unstable behaviour. The magnitude of oscillation in the contention window values increases and the ECW stops flipping between the two values closest to the optimum, instead oscillating more widely. This causes the throughput to degrade significantly.

6.4 Unsaturated Case

Up to this point, all stations in the experiments used saturated transmission queues and made use of the full throughput available to them. Next, we examine the behaviour of C-VAP and AlphaAP based on the unsaturated case where the station connected to VAP 1 has limits on the maximum transmission throughput used. In this case, we would like the algorithm to ensure that the requested traffic is served for all unsaturated VAPs and that the remaining throughput is divided proportionally between the remaining stations.

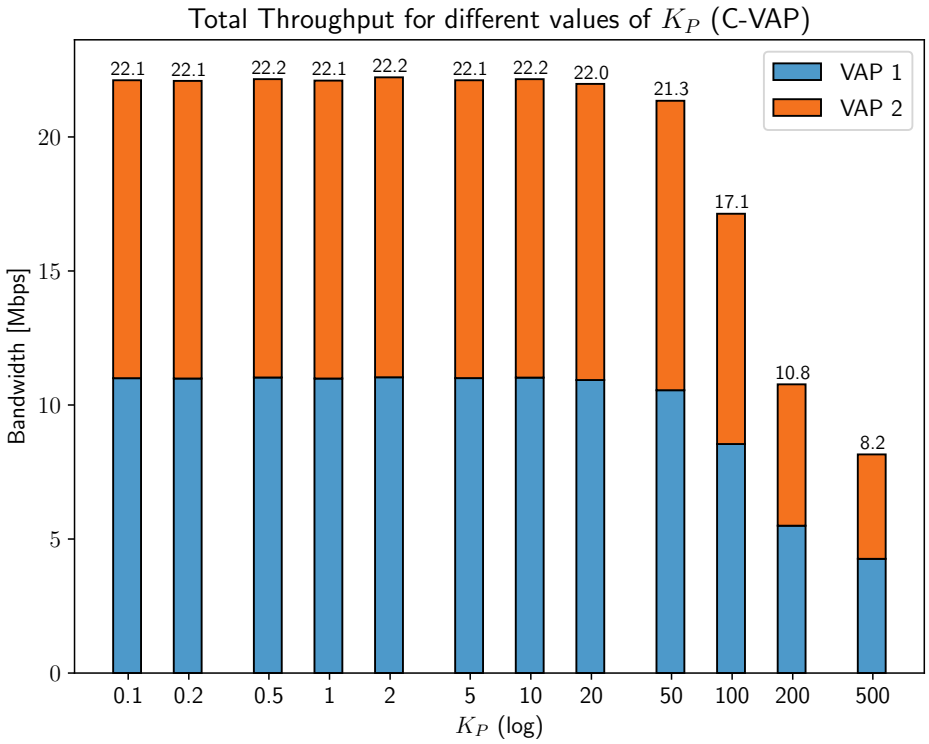

Figure 6.7: Throughput for different values of K_P

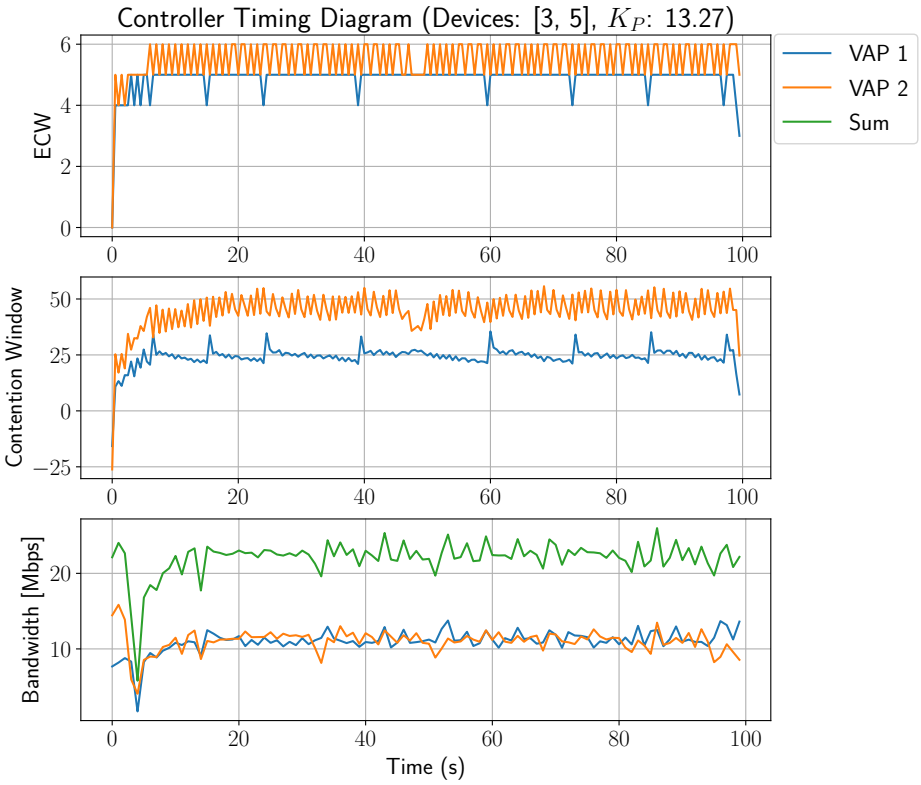

Figure 6.8: Timing Diagram for estimated K_P

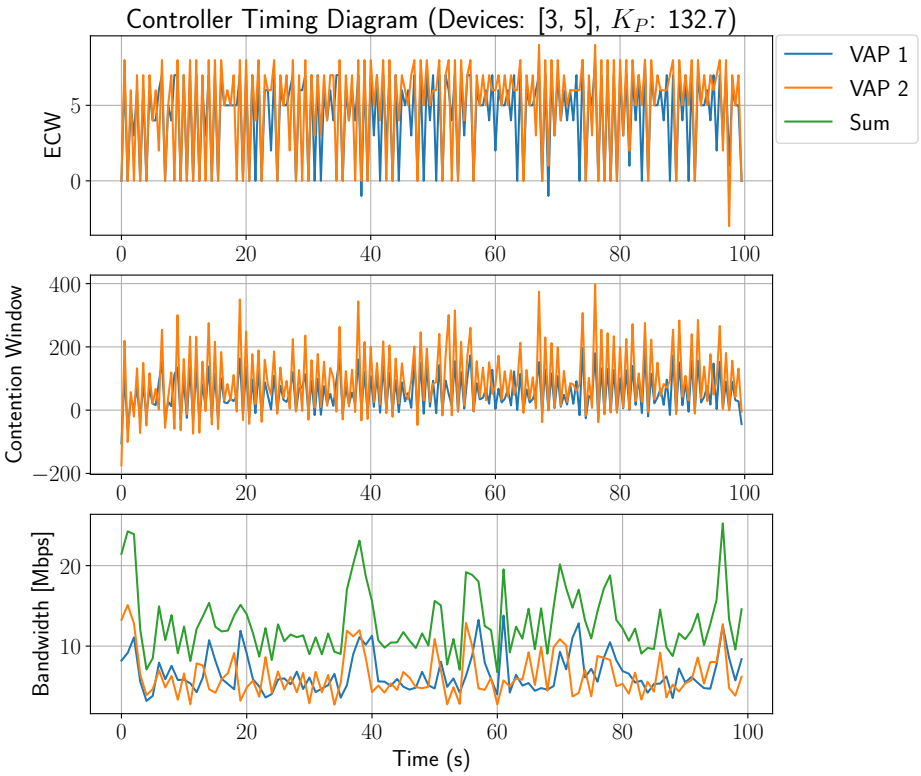

Figure 6.9: Timing Diagram for $10 * K_P$

The results are shown in Figure 6.10. The first experiment on the left side includes no throughput limitations and we observe a regular 50% split between both VAPs. The other experiments limit the throughput for VAP 1 to 2 Mbps. We see that the total throughput deviates slightly depending on the exact scenario (up and down) but stays in the same range as in the saturated case. The remaining throughput is indeed taken up by the saturated VAPs as desired and fairness guarantee is maintained except for a small deviation of less than 1%. In comparison, the EDCA experiments shown in the bottom row lead to slightly higher total throughputs and do not provide any fairness guarantees between the VAPs.

6.5 RTS/CTS and Frame Aggregation

In Chapter 3, we extend the C-VAP system model to support scenarios in which the duration of an average transmission is different from the duration of an average collision. For all experiments so far, stations did not make use of the RTS/CTS transmission mechanism (described in Section 2.1.3) which allows stations to reserve the channel using a special handshake mechanism.

In this section, we test the conclusion from Chapter 3 that the C-VAP algorithm would also perform well in cases where the average duration of a collision differs significantly from the average duration of a successful transmission.

For the experiment, we configure stations to make use of the RTS/CTS process by

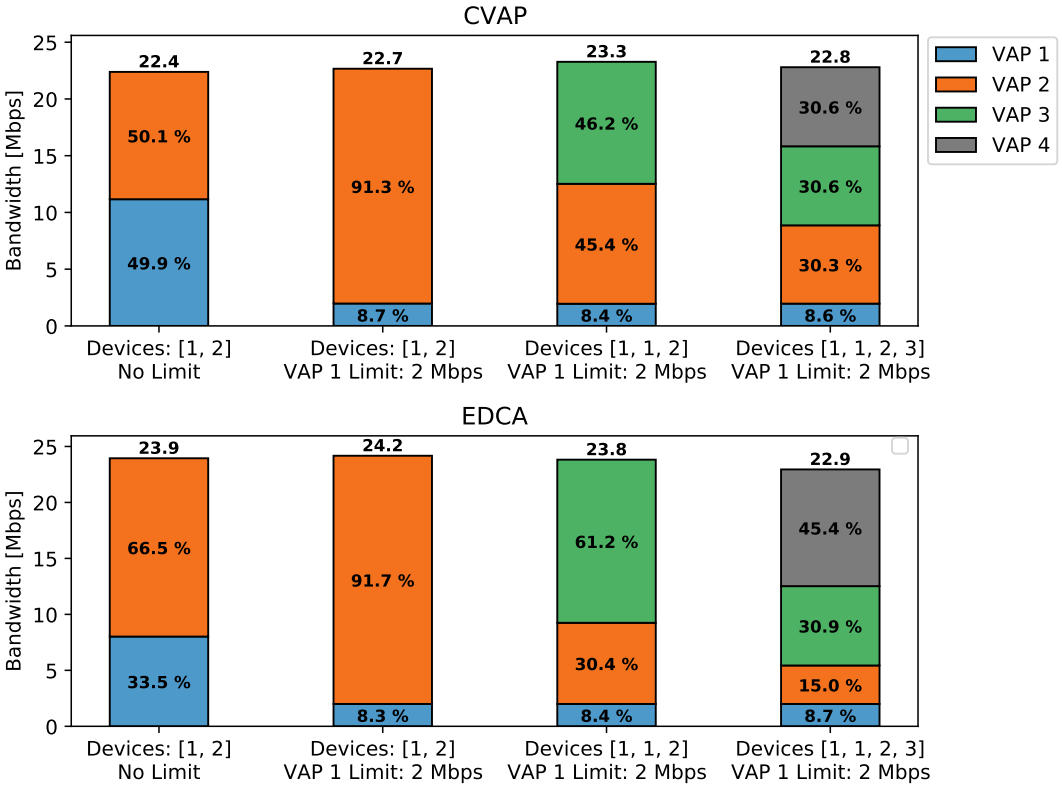

Figure 6.10: EDCA and C-VAP throughput in experiment with unsaturated stations

modifying their RTS threshold using the Linux *iw* command and repeat an experiment similar to the one carried out in Section 6.2.

A closer examination of the measured data in this case demonstrated a problem with the implemented empty slot counting in the case of RTS/CTS. Additional experiments using two devices and a low, fixed contention window value show that around 10 times more empty slots are counted in the case with RTS/CTS compared to the case without. This unexpected effect only occurs when the number of collisions is high and it appears that the Raspberry Pi devices are not starting the backoff process as fast as the router expects them to after a collision of two RTS frames.

This is a problem for the algorithm because it makes it impossible to obtain an accurate measurement of P_e and, despite a very low contention window value, the measured P_e ends up being far too high. This in turn causes the algorithm to try and further lower the contention window and prevents it from operating correctly.

The algorithm starts to work when around 9-15 empty slots are subtracted from the total count of empty slots per collision, but it remains unclear why this intervention is necessary. The results shown in Figure 6.11 are collected using a correction -12 empty slots per collision. This demonstrates that the algorithm works in principal also for activated RTS/CTS, but it is more difficult to draw further conclusions since we are less confident that the measured values for P_e^* are fully accurate. Further investigation needs to be done to identify the cause for the large additional number of empty slots

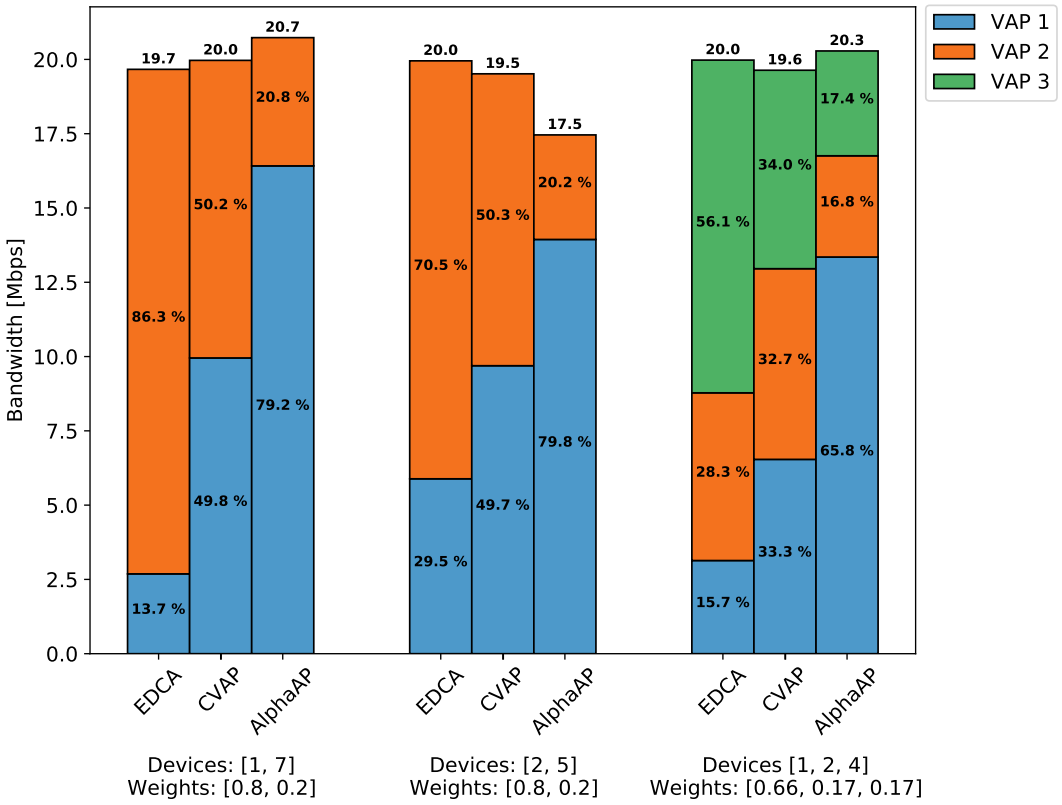

Figure 6.11: Throughput AlphaAP, C-VAP and EDCA in comparison using RTS/CTS

counted. Note that the maximum total performance is lower for EDCA, C-VAP and AlphaAP because RTS/CTS frames add additional overhead to the transmission.

To reduce the RTS/CTS overhead, every RTS/CTS handshake can be followed by transmission of more than a single data frame. Devices will not use frame aggregation when transmitting using the 54Mbps OFDM data rate. Therefore, for this experiment, we allow the use of the MCS data rate (MCS index 2). The changed data rate is the reason why the total performance for this experiment differs significantly from previous experiments for EDCA, C-VAP and AlphaAP.

Results are shown in Figure 6.12. The same problem regarding overcounting of empty slots applies here and we subtract 12 empty slots per collision, as done previously. C-VAP and AlphaAP appear to perform worse than the corresponding EDCA mechanism.

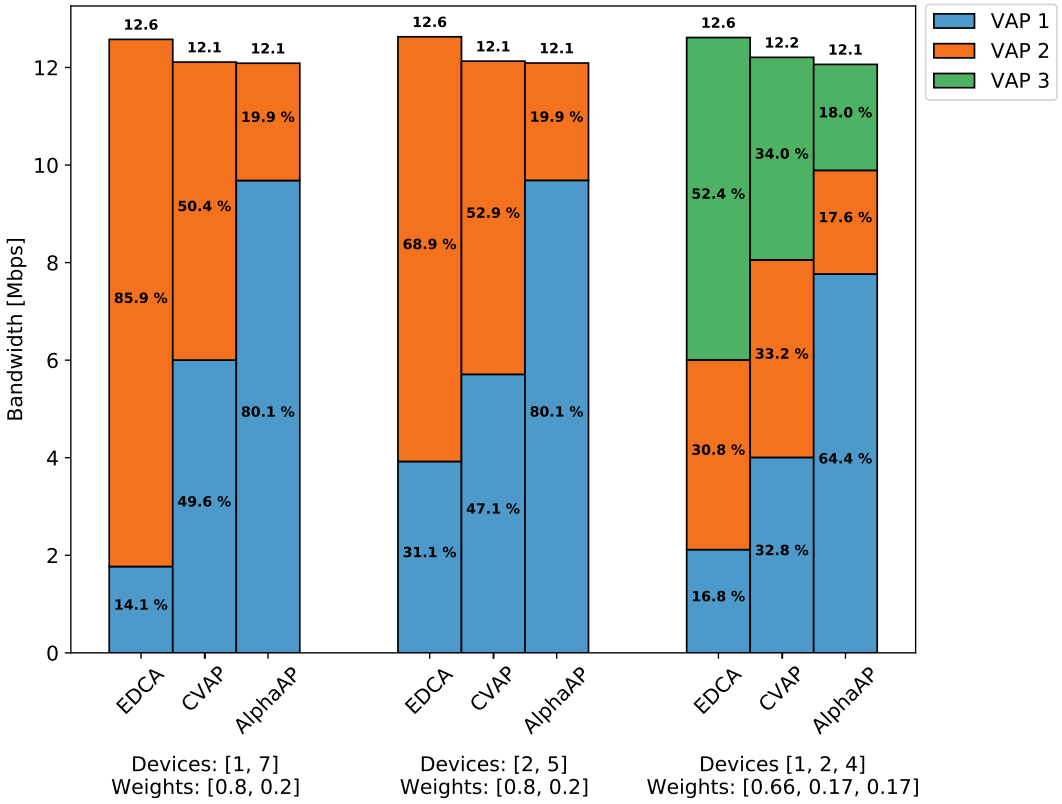

Figure 6.12: Throughput for AlphaAP, C-VAP and EDCA (using Frame Aggregation)

Chapter 7

Conclusion

For this project, we were able to make use of a largely undocumented, low-level firmware system to provide time-critical measurements, in the range of a few microseconds, in order to implement and demonstrate the feasibility of a control-theoretic approach to providing fairness guarantee in virtualised wireless networks.

We were able to achieve the desired fairness guarantee up to a deviation of less than 2 percentage points. This was demonstrated using a range of different experiments with a varying number of devices, both saturated and unsaturated, and making use of different optimisations to the WLAN standard, like frame aggregation and handshake mechanisms. Furthermore, we showed the effect of varying controller parameters on the operations. A submission for publication of the findings presented in this project is in preparation.

Major challenges in the implementation include the lack of documentation of the Broadcom firmware, especially given the complexity and volume of the code involved. Due to the missing documentation, available reverse-engineered information was used and carefully verified using a range of experiments designed to reveal and correct disparities in the measurements used. Additionally, the complexity of the WLAN standard, especially given the several amendments and adjustments throughout time, made it challenging to verify whether a certain type of behaviour is desired, or in fact a violation of the standard. We believe that one such violation was found during the project due to a bug in the firmware of the used Raspberry Pi wireless clients which causes them to send out bursts of a number of frames without going through the full contention process in between.

Future explorations, following up from the contributions provided in this project, include additional optimization of the algorithm controller parameters when used with handshake and frame aggregation mechanism. To date, the algorithm is less stable in operation with RTS/CTS and frame aggregation compared to scenarios without the use of these mechanisms. Additional evaluations can be carried out to test the performance in scenarios likely to be encountered in authentic real-world operation, such as, for example, allowing differing bitrates, testing a mix of UDP and TCP traffic and examining controller behaviour on more fluctuating traffic.

Bibliography

[1] *ASUS RT-AC86U Dual Band AC2900 Wireless Router Reviewed*. URL: https://www.smallnetbuilder.com/wireless/wireless-reviews/33158-asus-rt-ac86u-dual-band-ac2900-wireless-router-reviewed (visited on 12/04/2021).

[2] Albert Banchs and Luca Vollero. "Throughput analysis and optimal configuration of 802.11e EDCA". In: *Computer Networks* 50.11 (Aug. 2006), pp. 1749–1768.

[3] Albert Banchs et al. "Providing Throughput and Fairness Guarantees in Virtualized WLANs Through Control Theory". In: *Mobile Networks and Applications* 17 (Aug. 2012), pp. 435–446.

[4] *BCM43xx Specification*. URL: https://bcm-v4.sipsolutions.net/ (visited on 12/04/2021).

[5] Gautam Bhanage et al. "SplitAP: Leveraging Wireless Network Virtualization for Flexible Sharing of WLANs". In: *IEEE Global Telecommunications Conference GLOBECOM* (2010), pp. 1–6.

[6] Giuseppe Bianchi. "Performance analysis of the IEEE 802.11 distributed coordination function". In: *IEEE journal on selected areas in communications* 18.3 (2000), pp. 535–547.

[7] McKinsey Company. *The road to 5G: The inevitable growth of infrastructure cost*. URL: https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/the-road-to-5g-the-inevitable-growth-of-infrastructure-cost (visited on 12/04/2021).

[8] Rosario G Garroppo et al. "Providing air-time usage fairness in IEEE 802.11 networks with the deficit transmission time (DTT) scheduler". In: *Wireless networks* 13.4 (2007), pp. 481–495.

[9] Francesco Gringoli and Lorenzo Nava. *OpenFWWF website*. URL: http://netweb.ing.unibs.it/~openfwwf/ (visited on 12/04/2021).

[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". In: *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)* (2016), pp. 1–3534.

[11] AsusTek Computer Inc. *AC2900 Dual Band Gigabit WiFi Gaming Router with MU-MIMO, AiMesh for mesh wifi system, AiProtection network security by Trend Micro, WTFast game accelerator and Adaptive QoS*. URL: https://www.asus.com/uk/Networking/RT-AC86U/ (visited on 12/04/2021).

[12] Cisco Systems Inc. *Cisco Visual Networking Index (VNI) Global and Americas/EMEAR Mobile Data Traffic Forecast*. 2019. URL: https://www.cisco.com/c/dam/m/en_us/network-intelligence/service-provider/digital-transformation/knowledge-network-webinars/pdfs/190320-mobility-ckn.pdf (visited on 12/04/2021).

[13] Jong-Seo Lee and Il-Young Moon. "Research on Virtual Network for Virtual Mobile Network". In: *International Conference on Computer and Network Technology* (2010), pp. 98–101.

[14] Aptilo Networks. *Aptilo Mobile Data Offloading Solution*. May 2017. URL: https://www.aptilo.com/wp-content/uploads/2017/05/Aptilo_Mobile_Data_Offloading_v17-02-17.pdf (visited on 12/04/2021).

[15] Paul Patras, Albert Banchs and Pablo Serrano. "A control theoretic approach for throughput optimization in IEEE 802.11e EDCA WLANs". In: *Mobile networks and applications* 14.6 (2009), pp. 697–708.

[16] Paul Patras et al. "A Control-Theoretic Approach to Distributed Optimal Configuration of 802.11 WLANs". In: *IEEE Transactions on Mobile Computing* 10.6 (2011), pp. 897–910. ISSN: 1536-1233.

[17] Matthias Schulz and Jakob Link. *Nexmon Channel State Information Extractor*. URL: https://github.com/seemoo-lab/nexmon_csi (visited on 12/04/2021).

[18] Matthias Schulz, Daniel Wegemer and Matthias Hollick. "The Nexmon Firmware Analysis and Modification Framework: Empowering Researchers to Enhance Wi-Fi Devices". In: *Computer Communications* 129 (May 2018).

[19] Pablo Serrano et al. "Control theoretic optimization of 802.11 WLANs: Implementation and experimental evaluation". In: *Computer networks (Amsterdam, Netherlands : 1999)* 57.1 (2013), pp. 258–272. ISSN: 1389-1286.

[20] Pablo Serrano et al. "Control theoretic optimization of 802.11 WLANs: Implementation and experimental evaluation". In: *Computer Networks* 57.1 (Jan. 2013), pp. 258–272.

[21] Ronan Turner. "AlphaAP: Achieving Proportional Fairness in Virtualised WLANs". 4th Year Project Report. B.S. Thesis. University of Edinburgh, 2016.