Experiments in Obtaining Network Data For Evaluation of Wi-Fi Performance Models

Axel Smeets
dat12asm@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Björn Landfeldt

Examiner: Christian Nyberg

June 2, 2021

Printed in Sweden
Tryckeriet i E-huset, Lund

Acknowledgements

At this moment, looking back, I’m not exactly sure if I want to thank Björn Landfeld–my supervisor–for pitching his grand vision to me, or if I want to curse him for tricking me down the rabbit hole, nevertheless, I really want to thank you for your support, advice, feedback and sense of humour throughout this long journey – I learned a lot digging myself deeper and deeper down.

Next, I want to thank Stefan Höst–my second supervisor–for always keeping your door open, Jaume and co. at Telenor for your enthusiasm and ultimately for letting me do my own thing.

I also want to thank my friends at ON-IQ for your patience and support, my fellow plinkman Axel Mårtensson for listening and my dive buddies Kristoffer and Caroline always interrupting my studies.

And last but not least, a heartfelt thanks to my family.

Abstract

The performance characteristics of Wi-Fi networks have traditionally been studied and analysed using analytical models and simulations. Due to the complexity of wireless communication the existing analytical Wi-Fi network models rely on certain network constraints and simplifications in order to be mathematically tractable.

We set out to evaluate the practicality of using Wi-Fi performance models to estimate network performance by collecting the model necessary parameters directly from an access point. In order to evaulate, we must also collect network metrics, such as packet payload size and number of nodes, for comparison with the model parameters. We explore different venues to collect these parameters and metrics to find out if it is practical to apply the models in Wi-Fi networks.

After performing three attempts, we conclude that this is difficult due to several aspects in the Linux kernel, such as batching optimization patterns, proprietary kernel modules and firmware blobs. We believe that the data probably already exists somewhere in the device firmware or driver software, but a lack of documentation prevents us from effectively finding and using these data fields. We hope to see this data being made available as part of an automatic optimization functionality in future generations of Wi-Fi.

Popular Science Summary

Kan Wi-Fi fungera utan krångel? För att se om det är möjligt att förutsäga och eventuellt åtgärda prestandaproblem så har vi undersökt om det går att tillämpa akademiska Wi-Fi-modeller på mätdata från accesspunkter.

Vi konstaterar att det är mycket krångligt att samla in den data som behövs för att utvärdera modellernas tillförlitlighet. På grund av vissa designval i Linuxkärnan är det mycket svårt att utläsa denna data direkt. Drivrutinerna för nätverkskorten har förmodligen tillgång till den tidsdata vi behöver, men vi tror det krävs stor yttre påverkan innan chiptillverkarna aktivt gör den här typen av data tillgänglig. Förhoppningsvis kan detta bli en del av en automatisk optimeringsfunktionalitet i en framtida generation av Wi-Fi.

Modellerna i sig utgår från den koordineringsprocess som alla noder i ett Wi-Fi nät måste underkasta sig. Processen kallas för "Carrier-sense/Multiple Access", och beskriver nästan sig själv: varje nod lyssnar på nätets radiokanal och kontrollerar att ingen annan sänder, innan den försöker skicka ett meddelande. Modellering av denna process blir snabbt väldigt komplex och för att göra modellerna matematiskt hanterbara har man gjort ett flertal antaganden och förenklingar, vilket såklart kan påverka förmågan att spegla verkligheten.

För att undersöka modellernas tillförlitlighet har vi därför försökt samla in mätdata, från accesspunkter och laptops, som motsvarar de parametrar som modellen själv beskriver: paketstorlek och hur lång tid det tar att få tillgång till radiokanalen. Paketstorlek är trivial att samla in med t.ex. Wireshark, medans den sistnämnda visade sig vara betydligt besvärligare.

Vi har konstruerat och genomfört tre uppställningar med experiment.

Den första, med det välkända programmet Wireshark, visade sig enbart kunna samla in tidsdata om det fanns speciellt hårdvarustöd för tidsstämplar (mycket ovanligt i konsumentprodukter).

I den andra uppställningen modellerade vi Linuxkärnans nätverkssystem som ett kösystem. På grund av optimeringar i hur Linuxkärnarn hanterar avbrott så har mätdata från detta experiment en för stor varians för våra ändamål.

I den sista uppställningen grävde vi djupare, in i nätverkskortets drivrutin där vi hittade några intressanta parametrar. Efter flera tester drar vi slutsatsen att det kanske går att använda dessa parametrar, men att vi inte kunnat bestämma parametrarnas.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

This introductory chapter begins by reviewing Wi-Fi usage in our modern society, and use this to motivate an evaluation of a IEEE 802.11 (Wi-Fi) performance model based on a Markov Chain approximation of the Distributed Coordination Function (DCF).

At the end of this chapter we describe the methodology, organization and problem definition.

1.1 Background

Wi-Fi seems to be the wireless home network protocol of the (forseeable) future. In 2014 a report on Wi-Fi adoption found that 25% of households, all over the world, had set up Wi-Fi networks. In households with fixed-line broadband access, 65% had set up a Wi-Fi network[13]. The report also states that the number of Wi-Fi-enabled devices is projected to increase.

Consumers today have higher expectations regarding network throughput than the original IEEE 802.11 standard was designed for back in the mid 90's. In recent years, the Wi-Fi label has become hugely popular and the number of Wi-Fi-capable devices have skyrocketed, especially in urban areas and neighbourhoods. The protocol that once was aimed at corporate sector is now almost everywhere around us, and in vastly different use-cases than for which it first was designed. Beside home network use, Wi-Fi networks are also deployed for mobile network off-loading [16].

Alongside this explosion of households relying on a Wi-Fi router to connect their everyday electronics (computers, phones, TVs) and "smart devices" (i.e. internet-connected devices), our usage patterns and quality expectations have similarly increased: video streaming in 1080p and even UHD (4K) is now possible on many platforms.

But the increased Wi-Fi usage does not come without problems. It has become widely known among consumers that Wi-Fi can exhibit poor performance (for a multitude of reasons). A user streaming video (a use case where user experience is sensitive to throughput) to their TV will have a significant impact on the quality of service another user on the same network experiences on their video conference (sensitive to latency & jitter). As more people attempt to work remotely this type of network contention can only be expected to become more common.

Ignoring the physical aspects involved, the primary task of a wireless network protocol is to share the underlying medium to all clients in an effective manner. As with all radio technologies, Wi-Fi is primarily constrained by the radio spectrum it can utilize. All available performance is derived from a clever exploitation of this physical medium. As Wi-Fi usage has increased, the corresponding increase in radio spectrum usage, and the resulting issues of media sharing and interference, puts the protocol, and its medium access mechanisms in particular, under even more pressure.

1.2 Motivation

The performance of a household Wi-Fi network is not solely determined by a router or the broadband connection. Factors such as network configuration (channel settings, guard intervals, access modes), environment (noisy neighbours?), and clients (e.g. hardware and Wi-Fi generation) have a major impact on the ultimate network performance perceived by users.

To meet a wide variety of customer expectations in scenarios such as VoIP, low-latency gaming, ultra-high definition streaming and many network nodes, Wi-Fi has evolved dramatically in complexity, resulting in a multitude of configurable parameters. Even though newer routers are able to (somewhat) automatically (re)configure themselves based on analysis of neighbouring networks, they are not guaranteed to be optimal since they have a local view of the network (i.e. one point-of-view). Older devices rely on manual configuration, often using factory defaults.

If possible, could measuring (the right) Wi-Fi performance metrics shed some light on why the perceived network performance is poor in a given situation? Could the performance metrics be used to construct expert-type systems? In addition would these metrics be useful in the development of autonomous (re)configuration algorithms, embedded in a router? How would these algorithms be designed? Our position is that a reasonable–for a reasonable definition of the definition–approach is to build algorithms upon a model.

Accurately modelling the Wi-Fi communication and related performance characteristics is an active field of research and today there are various proposed models which perform well in simulations [4][6]. Some of these models are based on the observation that Wi-Fi implements *Carrier-sense multiple access/Collision avoidance* (CSMA/CA)—"listen before speaking"—in a *Distributed Coordination Function* (DCF) to reduce the likelihood of collisions happening in the first place, and what to do when collisions occur. A branch of these models–of which one we will attempt to evaluate in this report–are built on the approximation of the DCF as a Markov Chain. Furthermore, the models are often constrainted by assumptions, neccesary for mathematical tractablility, that cast doubt on the models ability to reflect and perform in the physical world.

Evaluating if the models, despite their assumptions, are useful for determining network performance could potentially be of enourmous benefit for consumers, business and ISPs alike. Imagine each router embedding and periodically running the model with locally sourced data, automatically alerting the end-user or

ISP of potential performance problems and possible interventions. Who knows, at some point in the future, devices might even attempt to cooperatively (and autonoumously!) resolve identified network problems.

In this report we solely collect metrics from devices running Linux, as it is the foundation of the majority of all Wi-Fi products on the market today. The same software that runs on a Wi-Fi router can be run on a laptop and since the Linux kernel is "open source", we can experiment directly with the software itself.

1.3 Method, Problem Definition and Organization

We aim to evaluate how well the model presented in [6] perform in the physical world. The methodology for our work will be explained in further detail in following chapters, and is based on analysing the model, collecting and comparing empirical data with the model.

Our problem definitions are formed by taking our overall goal of evaluating the Felemban-Ekici model and breaking it into smaller pieces:

- *Problem 1* - primary question: is the Felemban-Ekici model from [6] *useful* for determining Wi-Fi network performance?
- *Problem 2* - definition: what is a reasonable definition of *useful* in this context?

Since we have elected to use an experimental methodology, we must also include definitions related to the collection and evaluation of empirical data.

- *Problem 3* - analysis: what data should be collected?
- *Problem 4* - experiment: how should the necessary data be collected?
- *Problem 5* - evaluate: compare collected data with model and our definition from *Problem 2*

The remainder of this thesis is organized as follows.

Chapter 2 provides a background to, and overview of, related systems, protocols and hardware.

Chapter 3 introduces the research field and prior works.

In Chapter 4 we present the methodology and experiments dervied from *Problem 2*, *Problem 3* and *Problem 4*.

We show collected data for *Problem 3* and *Problem 4* in Chapter 5, and discuss these results with regards to *Problem 5* and *Problem 1* in Chapter 6 along with ideas for future work and our closing thoughts.

Some source material can be found in the Appendix. However, please refer to the repository available online at `https://github.com/smeets/thesis` for more content and details.

Chapter 2
Background

This chapter gives a brief introduction to the technologies and tools that were used in this project. As explained in the introduction, the primary goal of this thesis was to evaluate the usefulness of the model presented in [6] by comparing the channel access time of the model with an estimation, derived in Chapter 4. As will be explained in Chapter 4, obtaining exact measurements turned out to be non-trivial and we therefore present information helpful in understanding the forthcoming chapters of this thesis.

We begin with a brief overview of the IEEE 802.11 protocol and the *distributed coordination function* in particular, to form a basis for understanding the Felemban-Ekici model, as well as our measurement setup.

Since we will use Linux-based devices to perform measurements it will be important to also have some understanding of the Linux kernel's networking system. As we are particularly interested in measuring the time to send a packet, we describe the network stack components which control and process an outbound (egress) UDP packet, limited to 1500 bytes in order to not incur PHY-layer fragmentation.

Subsequently we provide an overview of the hardware used in this thesis, the TG799-vac router, its OpenWrt system and how we interact with the firmware using the micro bus system architecture.

Finally, we describe why Wireshark, a well-known and widely used open source tool for capturing and inspecting network data, had to be replaced by more special-purpose programs. We also introduce the program that was developed to perform our network experiments—Jana—and compare it with existing programs.

2.1 IEEE 802.11

The ubiquitous family of wireless network protocols and ammendments, such as IEEE 802.11g and IEEE 802.11x, is commonly known as Wi-Fi. Specifically, IEEE 802.11 defines the PHY and MAC layers of the network stack. Each ammendment introduces additions, redactions and changes to these layers. It is beyond the scope of this report to provide a detailed overview of the, sometimes quite significant, differences between ammendments. Of special interest for our work is the *Distributed Coordination Function* (DCF) which remains relatively unchanged, primarily for

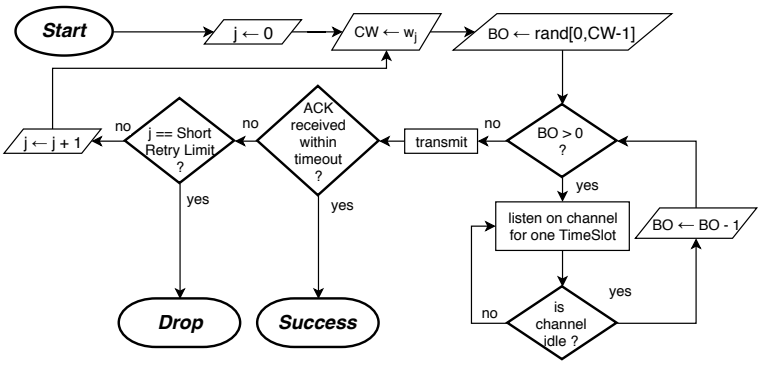


Figure 2.1: Schematic flowgraph of the DCF, where j is the current transmit attempt, CW the current contention window size, W_j the contention window size at attempt j and BO the back-off counter.

backwards compatibility reasons. For a more complete definition of the *Distributed Coordination Function* (DCF) the reader is referred to [1], [2] and [3].

In order to effectively allow multiple clients to access and utilize a shared medium, a medium access control (MAC) protocol is utilized. The MAC protocol governs *when* clients interact with the underlying physical medium (PHY-layer protocol specifies *how*).

IEEE 802.11 implements a *Carrier-Sense Multiple Access/Collision avoidance* (CSMA/CA) medium access control (MAC) scheme with a binary exponential back-off algorithm. The CSMA/CA algorithm is run locally on each network node and is called the *Distributed Coordination Function* (DCF). Figure 2.1 describes the access and back-off mechanisms of the DCF as a flowgraph.

When nodes in a IEEE 802.11 network want to transmit data they must first listen on the channel and wait until no activity has been detected for a duration, the *DCF Interframe Space* (DIFS). Since this effectively synchronizes the nodes waiting to transmit, a random delay is introduced to desynchronise nodes. This delay is called back-off time ($T_{backoff}$) and relates to the *collision avoidance* algorithm. The back-off procedure quantizes time into discrete *time-slots*, each 9 μ to 50 μs long, depending on the IEEE 802.11 "version".

While in back-off, nodes listen on the medium for a full *slot* and, if no activity has been sensed, decrements the back-off counter. If nodes detect activity on the medium during a slot, the counter is not decremented (counter freezing). Upon reaching zero the node may attempt to (re)transmit. If no `ACK` has been received after a certain duration (`ACK-Timeout`) the node waits another DIFS, enters further back-off and restarts its journey back to zero again. The node has a fixed number of attempts to retransmit the frame, `ShortRetryLimit` [1], and drops the frame once exceeded. The exponential increase of the back-off counter is visualised in Figure 2.2.

Time spent in back-off for each (re)transmission attempt j is described in equation 2.1, where *SlotTime* is defined in [1] and $\mathcal{U}(0, W_j - 1)$ is a uniformly

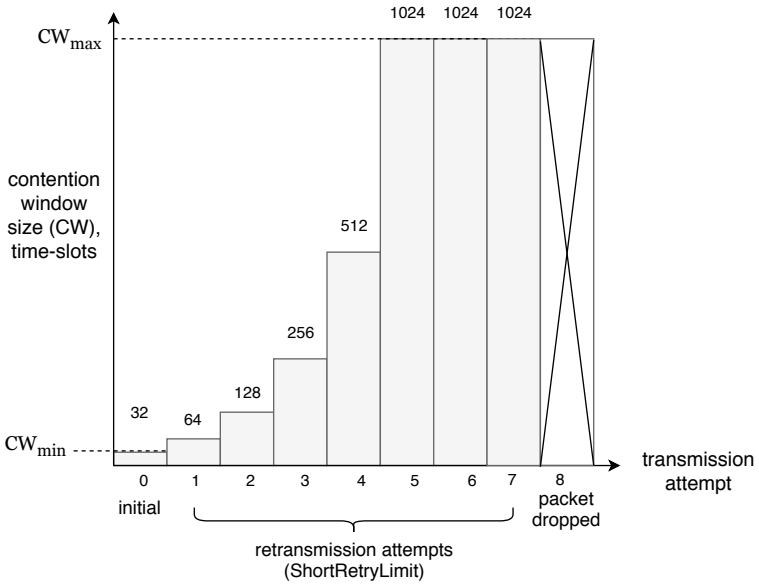


Figure 2.2: Contention window size increases exponentially on each retransmission attempt, from CW_{min} up to CW_{max}

sampled integer value between 0 and the contention window size $W_j - 1$, i.e., the maximum number of time-slots (exclusive, since 0 is a valid back-off value).

$$T^j_{backoff} = SlotTime \times \mathcal{U}(0, W_j - 1) \tag{2.1}$$

The contention window configuration is commonly expressed by two parameters, CW_{min} and CW_{max}. As seen in Figure 2.2, the initial contention window size is CW_{min} and each subsequent transmission attempt will double the previous window size, with a maximum value CW_{max}. The transmission attempt m where the window size becomes CW_{max} is defined in Equation 2.2.

$$m = \log_2 \frac{CW_{max}}{CW_{min}} \tag{2.2}$$

Equation 2.3 defines the contention window size at attempt j, where L is the maximum number of retransmission attempts (`ShortRetryLimit`). The defined values of j are the initial attempt ($j = 0$) and an extra L attempts before termination, which resolves to $0 \leq j \leq L$.

$$W_j = \begin{cases} 2^j CW_{min} & \text{if } 0 \leq j < m, \\ CW_{max} & \text{if } m \leq j \leq L \end{cases} \tag{2.3}$$

The description above details one of two relevant "access modes" (MACs) in IEEE 802.11—"basic mode". During CSMA/CA each node listens for activity locally and can therefore fail to detect nodes whose signal is too weak to be received, causing what's known as the *hidden node* problem.

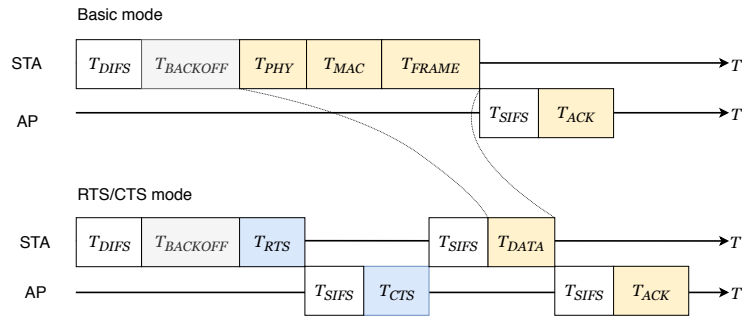


Figure 2.3: Time overview of a successful frame transmission and response in "basic mode" and "RTS/CTS mode"

The second multiple access scheme requires that clients (also known as stations or STAs) first acquire the right to transmit, using a *request-to-send* (RTS) message. If the STA is given permission, the AP will respond with a *clear-to-send* (CTS) message. This access scheme is called RTS/CTS and potentially increases performance in scenarios with multiple "hidden nodes". The RTS/CTS protocol assumes that all STAs have similar tx/rx capabilities. An overview of the required interframe spaces (IFS) and protocol packets for "basic" and "RTS/CTS" modes and how they compare, can be found in Figure 2.3.

2.2 Linux Networking

The Linux kernel has a complex networking subsystem which is beyond the scope of this report to detail. Since we will be using Linux-based machines in our tests and measurements, this section will present a refresh on the Linux networking stack. Due to the measurements needed, only the outbound (egress) path is described.

A high-level schematic view of a packet's path can be found in Figure 2.4. It shows the packet's path from a userspace program, through the kernel and into the network interface responsible for physically transmitting it. There are 5 main components in the figure: userspace, kernel, sockets, queueing discipline (qdisc), driver and Network Interface Card (NIC). Each component is more complex than described and interested readers are referred to [10], [12] and [9, Chapter 17]. The flow through Figure 2.4 and the 5 components is described below:

A userspace program initializes, binds and attempts to send data through a socket using system calls `socket(2)`, `bind(2)` and `send(2)`, `sendto(2)` or `sendmsg(2)`.

The linux kernel allocates a per-socket kernel buffer (`skb`), where related kernel (bookkeeping), socket and packet data is stored. The initial size of the `skb` depends on the `net.core.wmem_default` and `net.core.wmem_max` kernel parameters. The `send(2)`, `sendto(2)` and `sendmsg(2)` syscalls must request memory from the `skb` belonging to the socket in use, before proceeding through the packet processing stack (UDP/TCP, IP, etc) and finally enqueuing the packet in the queueing discipline (qdisc). The syscalls either blocks or fails, with `EWOULDBLOCK`

or `EAGAIN`, when the packet does not fit into the send buffer (`skb`), depending on the I/O mode of the socket (blocking or non-blocking).

The queueing discipline (qdisc) system is the internal queueing system of the linux kernel. A qdisc is a configurable, per-network device queueing system with a default "pfifo_fast", prioritized fifo, queue, see Figure 2.5. As packets arrive, the qdisc signals the driver, using a scheduler, that data is available. The qdisc can be queried and configured using the `tc(8)` (traffic control), `ip(8)` and `ifconfig(8)` programs. There is a reconfigurable CPU core-to-qdisc mapping that specifies which CPUs can deliver packets to a specific qdisc. This mapping can be configured using `ethtool(8)`.

The driver interfaces between the kernel and network device hardware/-firmware and can be considered a program itself (kernel module). Drivers used throughout this thesis all use an internal ring buffer for egress packets, called the "TX-Ring". The driver pulls packets from the qdisc into the "TX-Ring" and signals the NIC that packets are ready to be sent. The processing of freeing memory consumed by a packet is defered by an unknown time after the NIC signals the driver it has processed the packet.

The Linux kernel supports a feature called "packet taps" which enable kernel modules to capture packets as they enter or exit the kernel. The outbound taps are called at the end of the Linux kernel's outbound packet processing path, right before control of the packet is handed over to the driver.

The network interface card (NIC) which primarily sends and receives packets. The internal design documents of many NICs are not publicly available and open source drivers only hint at the design of important subsystems, such as buffers and prefetchers.

The NICs used in this thesis are commonly found in laptops and do not have a reconfigurable internal "TX-Ring". Common sense suggests that NICs use some internal buffering mechanism in addition to the "TX-Ring" controlled by the driver. Since we have no way of differentiating between these two "TX-Rings" the "TX-Ring" depicted in Figure 2.4 should be seen as an abstraction of these two.

2.3 Intel Wireless Wi-Fi Driver

All selected machines used to measure Wi-Fi statistics have Intel wireless network interface cards (NIC). On Linux these NICs are controlled by the "Intel Wireless WiFi driver for Linux"—`iwlwifi`.

The `iwlwifi` driver supports both an older operation mode, dvm (`iwldvm`, as well as a newer, mvm (`iwlmvm`). The `iwlmvm` module, included as part of `iwlwifi`, in the Linux kernel source tree of version 4.13 was adapted for this thesis work to enable logging of packet timing statistics with microsecond accuracy.

Internally, the driver allocates a fixed number of ring buffers used for holding transmission frame descriptors, firmware commands and incoming frames. The memory of these ring buffers is shared between the host and the NIC by using DMA controllers but resides in host DRAM. By using a ring buffer the driver is able to queue up to 254 frame descriptors for transmission, and vice versa for receiving. This design dramatically reduces the per-frame communication overhead between

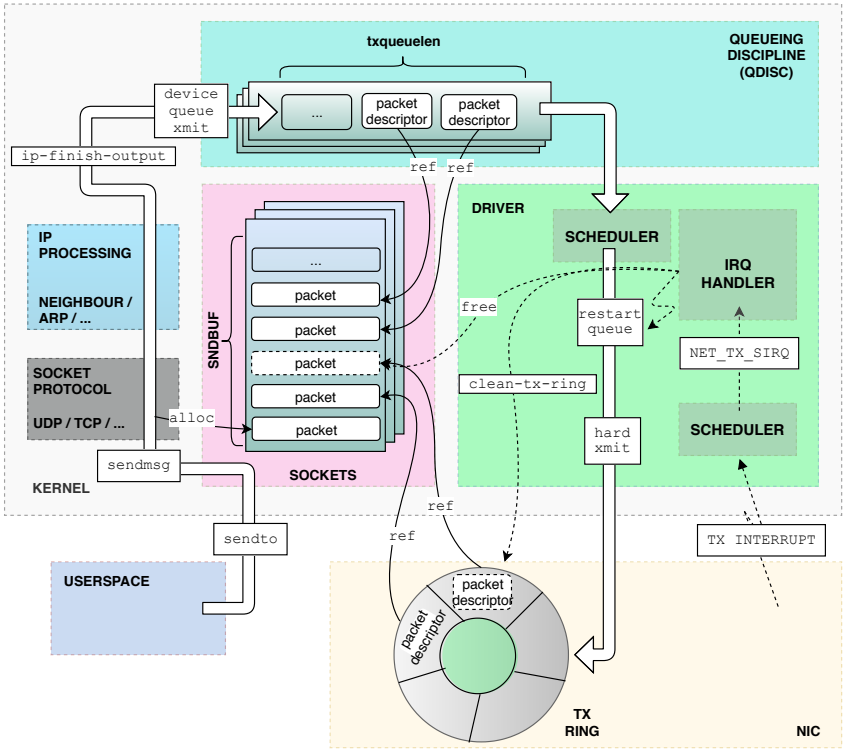


Figure 2.4: Schematic overview of a packet's way from userspace to the physical network interface.

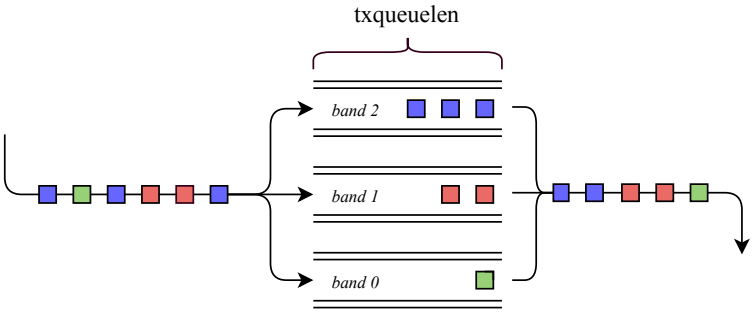


Figure 2.5: An overview of the default queueing discipline, `pfifo_fast`. Packets are enqueued into band 0, 1 or 2 depending on queue configuration and packet TOS bits. Band 0 is highest priority and band 2 lowest. Queue length is counted in number of packets.

Figure 2.6: The TG799-vac router from Technicolor

driver and NIC in certain conditions, at the expense of increased latency due to queueing.

2.4 TG799-vac

The OpenWrt-based router examined in this thesis, as depicted in Figure 2.6 and commonly known as TG799-vac, supports two concurrent IEEE 802.11n and 802.11ac (2.4GHz and 5 GHz) IEEE 802.11 interfaces using 2x2 and 3x3 antenna configurations, respectively (tx-antennas x rx-antennas). The device is manufactured by Technicolor and uses Broadcom and Quantenna modems.

The IEEE 802.11n chip supports SGi (Short Guard interval) and STBC (Space-time block code) over 20/40 MHz. The IEEE 802.11ac chip supports the same technologies and increases the channel bandwidth to 20/40/80 MHz. SGi reduces the guard interval used to eliminate intersymbol interference (as opposed to interframe interference, which IFS eliminates) from 400 ns to 800 ns. STBC is a technique to increase the reliability of transmission.

The router has been the default router provided by many Swedish ISPs and is therefore a good router to use in the experiments.

2.5 ubus - the OpenWrt micro bus architecture

`ubus` is command line interface to the bus daemon (`ubusd`). Modules are organized by namespaces in `ubusd` and can be interacted with using `ubus`. A usage example can be found in Listing 2.1.

This tool was primarily used to read statistics from the TG799-vac device, such as tx rates, packet counters and medium availability, intended for "in the wild" evaluation of the model.

```
1 $ ubus call wireless.accesspoint get
2 {
3     "ap0": {
4         "ssid": "wl0",
5         "station_history": 1,
6         "max_assoc": 0,
```

```
7           ... (omitted)
8       },
9       "ap1": {
10          ... (omitted)
11      },
12      ... (omitted)
13  }
```

Listing 2.1: ubus call listing all access points on this device

A key observation is that output is encoded in JSON format, which does not limit the size of numbers. Special care must be taken when using a JSON decoder to ensure that numbers are deserialized correctly. We observed one such problem during our thesis work.

2.6 Wireshark

Wireshark is a well-known program for capturing and inspecting network data, and has been used extensively during the research and development of this thesis. On Linux, it uses `libpcap` to register a packet tap for live capture. Due to the way packet taps work in the Linux kernel Wireshark cannot timestamp the moment the packet physically was transmitted, unless the network interface card explicitly supports hardware timestamps. None of the NICs used in this thesis have this feature, hence we cannot use Wireshark for any time measurements.

2.7 jana

`jana` is a program, developed during this thesis, for running network tests. Specifically, network tests using expontential, uniform and gamma distributed packet send rates and payload sizes. This is in contrast to more powerful tools such as `pktgen` and `trafgen`, which offer better performance and more per-packet control at the expense of more configuration. Another widely used network benchmark tool is `iperf3`, which we used for some experiments. However, it does not–to our knowledge–support inter-packet latency and payload distributions.

As traffic patterns have a high impact on the network the idea was to see how the model performed under bursty condition, such as web browsing and content streaming, as well as constant-rate traffic such as video conferencing and gaming.

The network test cycle is basically a "ready, set, go!"-type of design: a server waits for clients to say hello, and, after all N clients have been registered, broadcasts a "set" message which all clients echo back to the server, which again waits for N "set" before broadcasting a "go" message which marks the start of the network test. All clients wait approximately 1 second after they have received the "go" message before starting the packet generation, which decreases the likelihood of clients failing to start due to the start packet being lost. This simple approach proved resistant to partial network failures during the test wind-up stage.

The transmitting UDP socket was configured as non-blocking in an attempt to emulate the inter-packet latency distribution as much as possible.

For interested readers, we recommend browsing the various "Bufferbloat" projects (`https://www.bufferbloat.net/projects/`), which links to at least two very promising tools:

`IRTT` (`https://github.com/heistp/irtt`) – "IRTT measures round-trip time, one-way delay and other metrics using UDP packets sent on a fixed period, and produces both user and machine parseable output."

`Flent` – "The FLExible Network Tester" (`https://flent.org/`).

Chapter 3
Previous Work

Modelling of Wi-Fi performance has been an active field of research and this chapter provides an introduction to the modeling approach described by Bianchi [4] and subsequent efforts to improve it made by others. In particular, the studied model presented by Felemban & Ekici [6] will be discussed.

As mentioned previously, the IEEE 802.11 family specifies the PHY and MAC layers of WLANs. Analysing the performance impact of these protocols, and their numerous parameters, is key to improving network performance. There are two possible paradigms available: measuring/sampling or constructing an analytic model of the system/property. Measuring can be done directly on both physical hardware and software simulations whereas analytical models typically provide performance figures as solutions of equations.

The complete behavior of the IEEE 802.11 standards has yet to be captured by any single analytic model, and researchers instead take the standard scientific approach of solving a constrained variant of the problem, which in turn requires rigorous validation of the solution. A model may perform well in certain conditions and pathologically in others. These behavioural pecularities arrise from simplification of system behaviour and properties. In a given model, simplifications can also be inherent in the original construction mechanism—the way the model itself was constructed—which forces the model itself to undergo a sort of ironic self-analysis and validation. Model authors typically present a validation effort to prove their model's credibility, which further requires scrutiny of the validation itself, possibly in absurdum...

In short, model authors try to describe a complex system by modelling key components in a simplified setting, and validate their model using other models which have been more thoroughly reviewed.

Given this context, we first present the original Markov chain model from [4] and describe how it models IEEE 802.11 properties using the DCF, some significant and intentional simplifications made and a narrow selection of subsequent contributions made by others in [8][5][14][15].

Finally, the evaluated model [6] is described as well as the differences to both the original model [4] and some intrinsic model properties useful in forthcoming chapters.

3.1 The Bianchi Model

In [4], Bianchi presented a novel approach to modeling IEEE 802.11 performance by creating a Markov chain model of the *Distributed Coordination Function* (DCF). Bianchi defines three important properties: transmission probability (τ), normalised throughput (S) and channel access delay (T). This section provides a summary of the original model and definitions useful in later chapters.

Before going into detail about the Bianchi model, we start from the beginning.

The underlying assumption of a MAC-layer-based model of IEEE 802.11 is that, in a setting with more than 2 STAs, the MAC protocol should be the system bottleneck. By extension, it is reasonable to model the performance of the network based on the DCF.

In [4], Bianchi starts by limiting the proposed model analysis to fully-connected, single-hop networks in "saturation". The saturation condition requires that all STAs, at any point in time, always want to transmit. This allows Bianchi to omit send queue distributions and simplify collision probabilities, which will become important. Additionally, Bianchi assumes that there is a fixed number of N equivalent, contending STAs.

The back-off counter for a given STA at time t is represented by the stochastic process $b(t)$. Since the back-off counter at any given time t is dependent on the transmission history it follows that $b(t)$ is non-Markovian. To solve this problem Bianchi assumes that each transmission attempt collides with a constant and independent probability p.

In addition to the back-off counter process, $b(t)$, Bianchi also introduces a stochastic process $s(t)$ representing the current back-off stage for a given STA at time t. Recall from Equation 2.2 that m is the number of back-off stages, from which the states of $s(t)$ can be obtained as $(0, \ldots, m)$.

After breaking the historical dependency of the back-off process, Bianchi can model the time-discrete bidimensional process $\{s(t), b(t)\}$ as the Markov chain seen in Figure 3.2. As seen in the figure, the model ignores several important behaviors of the DCF, in particular retransmission limit and backoff counter freezing due to channel state.

1. **Retransmission Limit** - the model does not drop the packet after failing `ShortRetryLimit` retransmissions, instead it continues until the packet has been successfuly acknowledged.
2. **Back-off Counter Freezing** - in each back-off state, the probability of decrementing the back-off counter—equivalent to sensing the channel idle—is always 1.

Bianchi assumes that the conditional collision probability p is constant and independent of the back-off stage. This implies that p, the probability of an STA encountering collision during transmission, is equivalent to the probability that any of the other $N-1$ STAs also attempted to transmit, with transmission probability τ

$$p = 1 - (1 - \tau)^{N-1} \tag{3.1}$$

By modelling the back-off counter itself, combined with the saturation condition and omission of retransmission limit, Bianchi solves the transmission probability τ by essentially finding the steady state probability of the back-off counter being zero, and obtains

$$\tau = \frac{2(1-2p)}{(1-2p)(CW_{min}+1) + pCW_{min}(1-(2p)^m)} \tag{3.2}$$

However, since τ is derived from the back-off counter, and the back-off counter depends on the conditional collision probability p, it follows that τ and p are recursively defined. Bianchi constructs a nonlinear system with equations for τ and p and solves it numerically.

With probabilities for τ and p, Bianchi continues to his core contribution—the normalised throughput. Denoted S, normalised throughput is defined as "the fraction of time the channel is used to transmit payload bits".

$$S = \frac{E[\textit{payload information transmitted in a slot time}]}{E[\textit{length of slot time}]} \tag{3.3}$$

With $E[P]$ denoting mean payload size, probabilities for transmission P_{tr} and transmission success P_S, the numerator in 3.3 becomes $P_S P_{tr} E[P]$. In a similar fashion, the denominator in 3.3 can be expressed as the sum of empty time slots, busy time slots and times slots with collisions. Let σ be the duration of an empty slot, T_S the average time the channel is sensed busy in case of successful transmission, and T_C the average time the channel is sensed busy in case of collision for the non-colliding STAs. Now, Bianchi presents an equation for the normalized througput S

$$S = \frac{P_S P_{tr} E[P]}{(1-P_{tr})\sigma + P_{tr} P_S T_S + P_{tr}(1-P_S) T_C} \tag{3.4}$$

Note that S is expressed independent from the "access modes" found in Figure 2.3, which details the communication flow of a single, successful packet transmission and acknowledgement. Let $H = T_{PHY} + T_{MAC}$ and assume propagation delay δ. The differences between the "access modes" can thus captured by the variables T_S and T_C

$$\textit{basic} \begin{cases} T_S^{bas} = H + E[P] + SIFS + \delta + ACK + DIFS + \delta \\ T_C^{bas} = H + E[P] + DIFS + \delta \end{cases} \tag{3.5}$$

$$RTS/CTS \begin{cases} T_S^{rts} = RTS + SIFS + \delta + CTS + SIFS + \delta \\ \qquad\quad + H + E[P] + SIFS + \delta + ACK + DIFS + \delta \\ T_C^{rts} = RTS + DIFS + \delta \end{cases} \tag{3.6}$$

From these equations Bianchi concludes that for any network of size N there exists a troughput-optimal transmission probability τ, achievable by tuning the congestion window sizes and CW_{min} and CW_{max}.

To make the model tractable Bianchi simplifies many behaviours, of which two have received considerable efforts to implement. Some notable publications

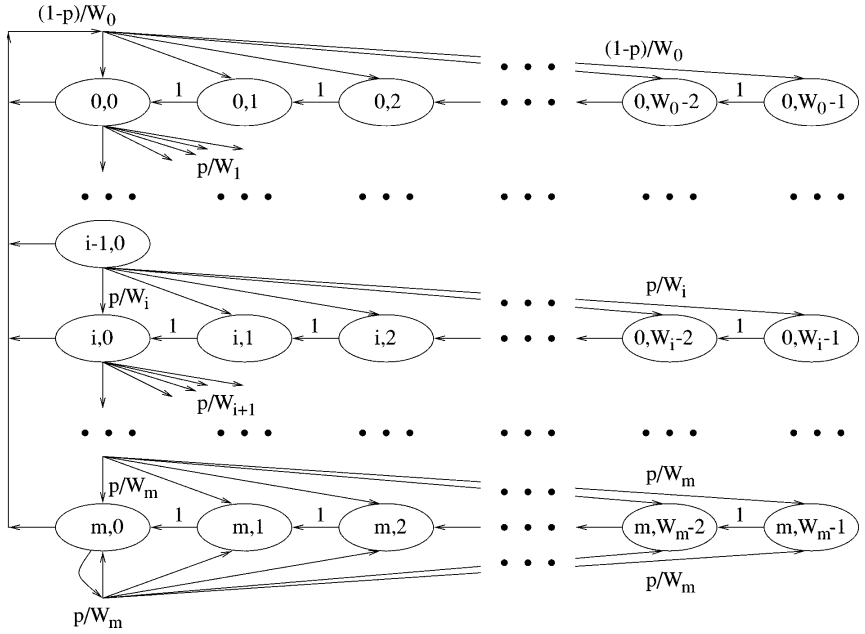


Figure 3.1: Bianchi's *Tagged-Node Markov chain* model of the IEEE 802.11 DCF where p is the collision probability, W_i the contention window size at attempt i $(0 \leq i \leq m)$ and m from Equation 2.2.

are Wu [8] and Chatzimisios [5], who proposed improved models that included the retransmission limit, followed by Zhang [15] and Xiao [14], who proposed models that include back-off counter freezing.

3.2 The Felemban-Ekici Model

In 2011, a decade later, Felemban & Ekici published an extended version of Bianchi's model, where they significantly improved the model's accuracy by introducing a more accurate behaviour of the entry into backoff and the backoff countdown procedures [6].

An overview of the TNMC model from [6] is presented in Figure 3.2. Some differences compared to Bianchi's TNMC are inclusion of retransmission limit (in state $\{0, L\}$, collision results in packet drop) and counter freezing P_f (probability of state a $\{i, j\}$ transitioning to itself).

The inclusion of retransmission limits results in a different expression of τ compared to Equation 3.2. Recall from Equation 2.3 that W_j is the size of the contention window at back-off stage j and that L is the short retry limit from [1]. Felemban & Ekici solves τ similarly to Bianchi [4] by finding the probability of the back-off counter reaching 0 in all back-off stages, expressed as

$$\tau = \frac{1 - P^{L+1}}{(\Sigma_{j=0}^{L}[1 + \frac{1}{1-P_f}\Sigma_{k=1}^{W_j-1}\frac{W_j-k}{W_j}])(1-P)} \tag{3.7}$$

where P is the conditional collision probability (equivalent to p in Bianchi's paper) from Equation 3.1.

Somewhat contradictory, Felemban and Ekici obtains different values for the parameters T_C and T_S, compared to Bianchi. This is probably explained by the different Wi-Fi generation targeted.

$$basic \begin{cases} T_C^{bas} = T_S^{bas} = DIFS + T_h + T_p + SIFS + ACK \end{cases} \tag{3.8}$$

$$RTS/CTS \begin{cases} T_S^{rts} = DIFS + T_{RTS} + SIFS + T_{CTS} \\ \qquad + SIFS + T_h + T_p + SIFS + ACK \\ T_C^{rts} = DIFS + T_{RTS} + SIFS + T_{CTS} \end{cases} \tag{3.9}$$

The equation for normalised throughput is indeed also different, but still similar to Bianchi's.

$$U = \frac{P_S T_p}{P_S T_S + (P_B - P_S)T_C + (1 - P_B) * \sigma} \tag{3.10}$$

Where P_S and P_B correspond to the probabilities that a slot contains a successful transmission and that a slot is sensed busy, respectively.

As shown in Figure 2.1, the DCF back-off process algorithm specifies that a node only decrements the back-off counter if the channel was sensed idle. In [6], the authors obtained an accurate countdown probability (P_d) by introducing an additional Markov chain to model the channel-sensing process and estimating the probability of *not* counting down, i.e. probability of counter freeze (P_f). This chain is called Channel-Sense Markov chain (CSMC). The counter freeze probability, P_f, is computed by finding the steady state probabilities of the CSMC by fixed point iteration.

The addition of the CSMC and P_f increased the model's accuracy significantly compared to other models in various test conditions. In particular, the introduction of P_f increased accuracy of the model when extended to *unsaturated* networks.

While the model proposed by Felemban-Ekici models the DCF more closely and accurately, several assumptions and omissions, inherited from [4], make the model a very interesting candidate for real-world testing, namely:

1. **Steady state** - the counter freeze probability P_f is computed by finding the steady state probabilities of the CSMC. The packet transmission and conditional collision probabilities τ and P, respectively, are also computed similarily by finding the steady states of the TNMC. Transient behaviour is not included.

2. **Colliding nodes** - to simlify the equation for collision, the maximum number of simultaneously colliding nodes is assumed to be 3.

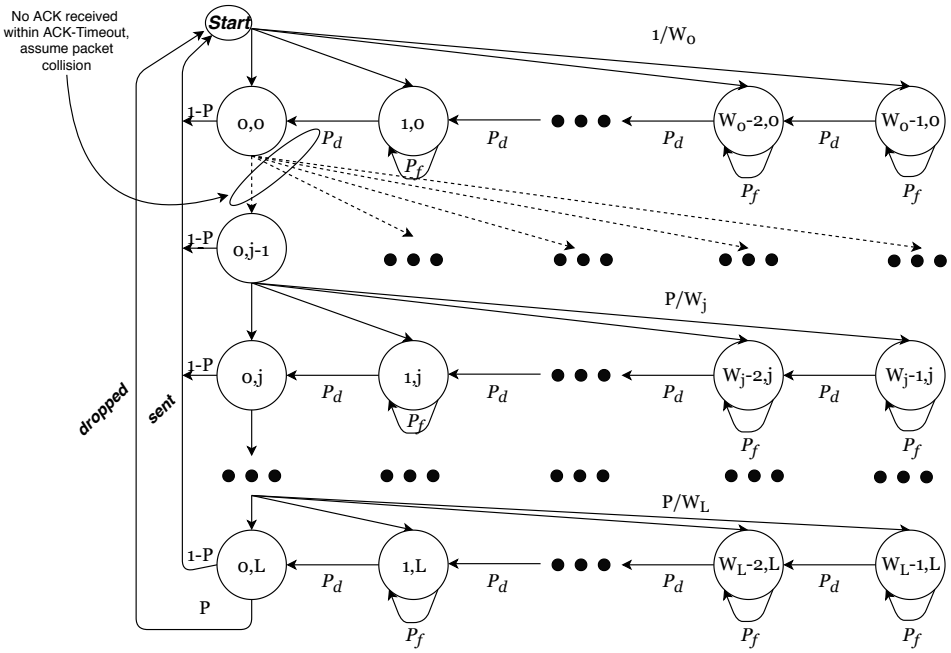


Figure 3.2: Felemban-Ekici's Tagged-Node Markov chain (TNMC) model of the IEEE 802.11 DCF. P is packet collision probability, P_d is probability to decrease backoff counter, $P_f = 1 - P_d$, W_j is contention window size at attempt j and L is the Short Retry Limit

3. **Wi-Fi b/g/n** - the original model from [4] was presented for Wi-Fi b/g networks and the evaluated model from [6] for Wi-Fi n. Today both Wi-Fi ac and Wi-Fi ax exist. These newer generations include support for beamforming and aggregation technologies that increase the performance of the network in ways not considered in [4] or [6].
4. **Independent packets** - to simlify the equation for the conditional collision probability P, packet collisions are assumed to independent.
5. **The capture effect** - a behaviour where a node successfully transmitting a packet is more likely to win the race for the next transmission, due to other nodes freezing their back-off counters, is not considered at all.

Another key contribution made in [6] was the presentation of an *unsaturated model*. Recall that the Bianchi model assumed that the network was in *saturation* conditions, i.e. all STAs always have something to send.

The condition limits the number of networks and scenarios the model can be effectively applied to and thus limits the usefulness of the model. In [6], the authors extended their saturated model work in unsaturated networks, primarily by introducing packet transmission rate distributions.

Chapter 4
Experimental Approaches

This chapter begins by presenting the overall rationale for the experimental approach used to evaluate the theoretical model.

The primary challenge of analysing the usefulness of the Felemban-Ekici model lies in the practical problem of accurately capturing network performance statistics. In total, three different experiments were designed, peformed and evaluated. Each subsequent experiment lead to a deeper understanding of the network system and how various systems interact and their effect on the results.

Any experiment is, by design, constructed and designed in order to observe a property or behaviour. So that is where this chapter starts – reasoning about different parameters of the Felemban-Ekici model. Armed with this set of parameters, experiment design then becomes a game of exploiting known system properties and behaviour to obtain a comparable data set.

After selecting intresting properties of the model we then proceed to present all model evaluation experiments in a chronological order. For each experiment we try to map the underlying idea to information presented in Chapters 2 and 3. The experiments will relate to an informal and incomplete model of the system under test and should therefore be read at least twice to answer questions such as "if we assume the model to be true, is the experiment itself sound?" and "is the model itself a good enough approximation?". In an attempt to spare the reader from inferring incorrect information based on experiment models, each experiment will be also presented with a short analysis regarding its soundness.

After describing all model evaluation experiments we continue to present the experiments designed to test if the stats collected from `ubus` on the TG-799 firmware showed any obvious problems.

Finally, we describe a proof-of-concept experiment to evaluate the Felemban-Ekici model that we ultimately could not perform.

4.1 Overview

As mentioned in the introduction, in order to solve the primary problem as described in Section 1.3 it has to be partitioned into more manageable pieces. The problem discussed in this section is "*Problem 2* - what is a reasonable definition of *useful*?".

The premise is as follows: in order to evaluate a theoretical model there must first exist some set of data provided by the model itself, and an equivalent set of data to compare against. In this report, the data provided by the model are referred to as the *model metrics*. The other dataset, the comparison set, is gathered from a specific device and referred to as the *captured metrics*.

Since both the original model from Bianchi and Felemban-Ekici models have strong performance claims backed by evaluations done with simulators, the stated parameters and their behaviour can be considered fact–the models accurately model IEEE 802.11 performance. What remains unknown is how well these models perform outside simulations – on physical hardware using (imperfect) software drivers. One could argue that since IEEE 802.11 is a specification all hardware and software that claim to follow that spec. should be considered equivalent to the spec. and by extension simulators as well. However, this is in fact an assumption that can serve as a conjecture and needs to be proven or disproven.

By this reasoning an answer to the original question can be formed:

Q *Problem 2* - what is a reasonable definition of *useful*?

A Construct an experiment and collect metrics that have either a direct or indirect equivalent in the theoretical model. If these metrics match numerically the theoretical model is useful. If these metrics don't match numerically the theoretical model can still be considered useful if the trends of these metrics match. If neither of these statments hold, the model is not useful. If this is the case then we cannot verify the model, and the model should be considered useless–after all, what is the point of an unverifiable model?

The experimental methodology essentially boils down to designing experiments, based on knowledge of a device and its system, in such a way that the experiment resulted in a set of *captured metrics* that were comparable with a predefined set of *model metrics*.

4.2 Model Metrics

As stated earlier, in order to know what to capture we must first select relevant metrics from the model and the protocol itself. After reading through [6] some variables and parameters are of particular interest:

- N – number of network nodes
- D – per-packet payload size distribution
- U – normalized throughput (wrt. channel bit rate)
- P – conditional packet collision probability
- T – channel access delay
- RTS/CTS – the model presents models with basic access, RTS/CTS and a hybrid access scheme using a payload threshold

In addition to these, there are also some parameters and variables that are used in [6] but originate from IEEE 802.11 specification itself:

- channel bit rate
- L – the `ShortRetryLimit`
- CW_{min} and CW_{max} – contention window min and max size

There are, of course, other parameters in the model, and certainly many more parameters in the implementations of IEEE 802.11 (e.g. power saving, distance, line-of-sight). These are, for the sake of tractability and real-world usefulness, ignored and may contribute to significant errors (although such errors have yet to be discovered in simulations).

As described earlier, the Felemban-Ekici model provides normalized throughput (U), channel access delay (T) and conditional packet collision probability (P).

Normalized throughput (U) is constructed upon the conditional packet collision probability (P), payload size distribution (D) and a variant of channel access delay, see Equation 3.10.

Thus the most important *model metrics* are *conditional packet collision probability* (P) – the probability that a packet collides during transmission – and *channel access delay* (T), defined as "the time from the packet becoming the head of the queue until the acknowledgment frame is received.".

Conditional packet collision probability (P) is a simple metric, "count the number of transmission attempts and collisions", but not an easy metric to obtain since it occurs at a very low level in the networking stack. A possible alternative is the *packet drop probability* P^{L+1} – the probability of exceeding the short retry limit. The Linux networking system reliably reports transmission success and failure, but not attempts.

The *channel access delay* (T) can be obtained by measuring the total time described in Figure 2.3. Such a solution would require radio equipment not available for this thesis. A key insight is that, in a controlled experiment, most of the parameters in Figure 2.3 are actually constant. The only dynamic parameter is the time spent in backoff, T_{BACKOFF}.

To summarize,

Q *Problem 3* - what data should be collected?

A The number of network nodes (N), conditional packet drop probability P^{L+1}, channel access delay (T), payload size D, channel bit rate, retry limit (L) and contention window sizes CW_{min} and CW_{max}.

With a set of well-defined *model metrics* we need to collect equivalent data ponts for, we now move on to describe the experiments, which provide an answer to "*Problem 4* - how should this be done?".

4.3 Experiment 1: Wireshark

The first experiment was based on the industry-standard Wireshark program to capture outgoing packets. Built on libpcap, Wireshark uses a Linux kernel feature

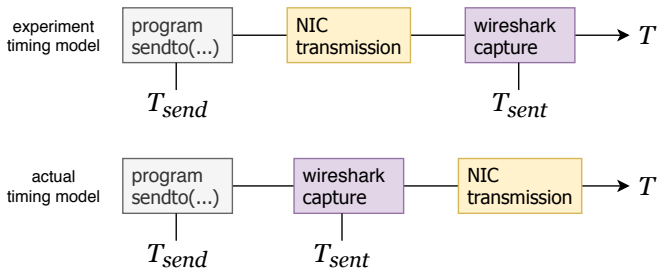


Figure 4.1: Experiment 1: Wireshark, experiment design vs actual timing models

called "packet taps" to "capture" packets. Wireshark records the packet itself and some related timing data in a log file in a "pcap" format.

Naively, the corresponding *channel access delay* was thought to be obtainable to comparing the timestamps between a send program's call to `sendto` and the "capture time" reported by Wireshark, as seen in Figure 4.1 (experiment timing model, i.e. "the model which makes the experiment work"). This would, in fact, be comparable to the complete timeline from Figure 2.3. No *captured metric* comparable to *conditional packet collision probability* could be constructed, simply due to the fact that Wireshark is too high-level to know about IEEE 802.11 frame transmission failures.

The experiment was designed as follows:

1. device `R`, a TG799-vac router
2. device `S`, connected via ethernet to `R`, running `jana` in server mode
3. device `C`, connected via Wi-Fi to `R`, running `jana` in client mode with no specified inter-packet latency distribution (a.k.a saturation mode)
4. wireshark running on `C`, captures outgoing packets to `S`

On `C`, placing the socket in non-blocking mode (`O_NONBLOCK`), `jana` can detect whenever the socket buffer is full and avoid measuring the time it took for the kernel to free some memory for the blocked packet.

With valid values for T_{send} and T_{sent}, we can obtain the time it took to transmit this packet, or something roughly equivalent to the *Channel access delay* by taking the difference $T_{\text{sent}} - T_{\text{send}}$

However, the T_{sent} timestamp obtained from Wireshark/libpcap is not what one would expect and for our purposes this divergence between what we expected and what libpcap delivered was too large to contiune this line of experimentation. In Chapter 6 we will discuss the underlying reasons for this in more detail.

4.4 Experiment 2: Queueing the Network System

As the Wireshark-based approach seemingly crashed into a hard wall, insights gleaned from the numerous attempts made to salvage it hinted at a more theoret-

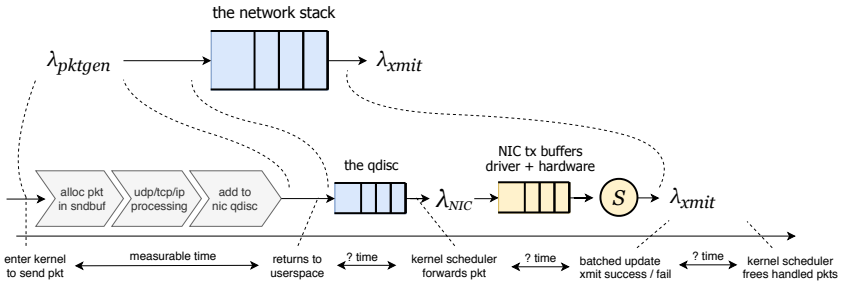


Figure 4.2: Experiment 2: Modelling the egress path of a packet as a queueing system

ical and academic design. *What if we modelled the whole networking stack as a queueing system? Wouldn't the properties we were looking for simply be properties of this system?*

The insight that finally ties these loose ideas together, is that a kernel-owned buffer, known as the `sndbuf`, owns the memory of each packet currently being transmitted or received on a particular socket. The full lifecycle is shown in Figure 2.4 and is summarized here. Some code calls into `sendto` or `sendmsg`, which attempts to allocate memory from the `sndbuf`, before potentially running tcp, udp or ip processing filters and finally enqueueing the packet in the qdisc. The kernel manages pumping packets from the qdisc into the NIC driver. After transmission, the NIC driver/hardware signals the kernel that packets were transmitted or dropped. The kernel puts these packets on a "to free" list and eventually frees the related memory, allowing new packets to be allocated from the `sndbuf`.

Modelling the network stack as a M/M/1/K system with K queue length, where K is

$$K = \frac{\texttt{sndbuf}}{E[D] + \textit{kernel overhead}} \tag{4.1}$$

where $E[D]$ is the average payload size and *kernel overhead* a constant per-packet bookeeping overhead.

The qdisc queue length is known, and the number of packets currently in the qdisc system can be queried. Packets can either be owned by the qdisc or the NIC and so, by assuming that the `sndbuf` is full, the number of packets in the NIC queue system can be computed as

$$L_{\text{NIC}} = K - L_{\text{qdisc}} \tag{4.2}$$

We know have an equation for the number of packets in the NIC queue system of Figure 4.2. Recalling Little's Law, $L = \lambda W$, which, in this context, states that the average number of packets L in the networking system is equal to the generation rate multiplied by the average time W each packet spends in the networking system. This law is applicable to any arrival process and service distribution, and crucially, it's also applicable to any subsystems as well.

With Little's Law we can obtain the service time W_{NIC} of a packet, i.e the average time a packet spends at the server. The exact definition of W_{NIC}, assuming a stable flow of packets and mostly full `sndbuf`, becomes

$$\begin{aligned} L_{\text{NIC}} &= \lambda_{\text{NIC}} W_{\text{NIC}} && \text{Little's Law} \\ W_{\text{NIC}} &= \frac{L_{\text{NIC}}}{\lambda_{\text{NIC}}} && \text{isolate service time } W_{\text{NIC}} \\ W_{\text{NIC}} &= \frac{K - L_{\text{qdisc}}}{\lambda_{\text{NIC}}} && \text{substitute } L_{\text{NIC}} \text{ from Equation 4.2} \end{aligned} \tag{4.3}$$

Furthermore, W_{NIC} can be deconstructed as shown in Figure 2.3 plus a term for time spent waiting in the queue, and provides the final link how W_{NIC} relates to the variable we are looking for, T_{BACKOFF}

$$\begin{aligned} W_{\text{NIC}} = {} & T_{\text{QUEUE}} + T_{\text{DIFS}} + T_{\text{BACKOFF}} \\ & + T_{\text{PHY}} + T_{\text{MAC}} + T_{\text{FRAME}} \\ & + T_{\text{SIFS}} + T_{\text{ACK}} \end{aligned} \tag{4.4}$$

Where T_{QUEUE} is the time spent waiting to be served, T_{DIFS} and T_{SIFS} are the interframe spaces defined in IEEE 802.11 and set by the access point [1], T_{PHY} the preamble, T_{MAC} time to send MAC header and T_{ACK} the time for a special `MAC` packet. Simply, `DIFS` and `SIFS` depend on network configuration whereas `PHY`, `MAC`, `FRAME` and `ACK` primarily depend on channel bit rate, guard interval and time quantization ($T_{\texttt{slot}}$).

Some practical problems had to be resolved:

1. Fast polling of the qdisc queue, solved by hacking the `tc` (traffic control) program, see `https://github.com/smeets/thesis/tree/master/d3fi/tcq`
2. A full qdisc will sliently drop new packets, solved by increasing the max queue length using `ifconfig txqueuelen`
3. The NIC should at all times be maxed out, solved by increasing the `sndbuf` size by increasing kernel parameters `/proc/sys/net/core/wmem_max` (UDP `sndbuf` maximum size) and `/proc/sys/net/core/wmem_default` (UDP `sndbuf` default size) to 10 MB. Combined with a large qdisc size this should allow the kernel to always have packets to pump into the NIC.

The complete experiment:

1. Start a UDP-spam process, see `https://github.com/smeets/thesis/tree/master/d3fi/ethx.py`
2. Get the active inode from `/proc/net/udp`
3. Log data from the specified iface and inode, see `https://github.com/smeets/thesis/tree/master/d3fi/data1.sh`, using data from `tcq`, `ethtool` and `procfs` device `/proc/net/udp`

4. Send log data into a web server, `https://github.com/smeets/thesis/tree/master/d3fi/server.js`, which stores the data and makes it available to a browser

5. A browser on the local host that loads a page from above server and periodically fetches new log data

The primary issue with this approach is related to how the kernel keeps already transmitted packets for an unknown amount of time before freeing the packet data in batches, increasing overall system throughput.

4.5 Experiment 3: Hacking on the driver

Compared to the previous attempt, a more practical approach would be to try and observe whenever packets are handed over to the NIC from qdisc and handed over from the NIC back to the kernel. Sampling this process would then give an estimate for the number of packets enqueued in the NIC.

This is in fact how the third and final experiment started: *"Where in the kernel can we log qdisc → NIC and NIC → kernel handover events?"*. The anwer, of course, is *"in the NIC driver"*.

While exploring in-tree wireless drivers a suspicious field appeared in a driver for intel wireless NICs (`iwlwifi`), `wireless_media_time`, described as

```
* @wireless_media_time:
 *  for non-agg: RTS + CTS + frame tx attempts time + ACK.
 *  for agg: RTS + CTS + aggregation tx time + block-ack time.
 *  in usec.
```

and while this description appears to include some components from Figure 2.3, the comment is too vague to form any definition. Some notes:

- "wireless media time" should, in our opinion, refer to the total duration the wireless medium was used. This implies that backoff is not included and that frame transmission time is included. However, the comment indicates the opposite.

- Consumer Wi-Fi networks often use a hybrid access mode, defaulting to basic and only sending RTS/CTS frames if the payload size exceeds some threshold. In our case this threshold is greater than the maximum amount of payload a IEEE 802.11 frame can contain, which implies that RTS/CTS will never be used in our setup.

- "frame tx attempts time" could include any combination of BACKOFF, PHY, MAC and FRAME components. It is possible to probe some of these components, e.g. varying payload size should give a corresponding impact if FRAME is included, BACKOFF can be proved by changing number of STAs.

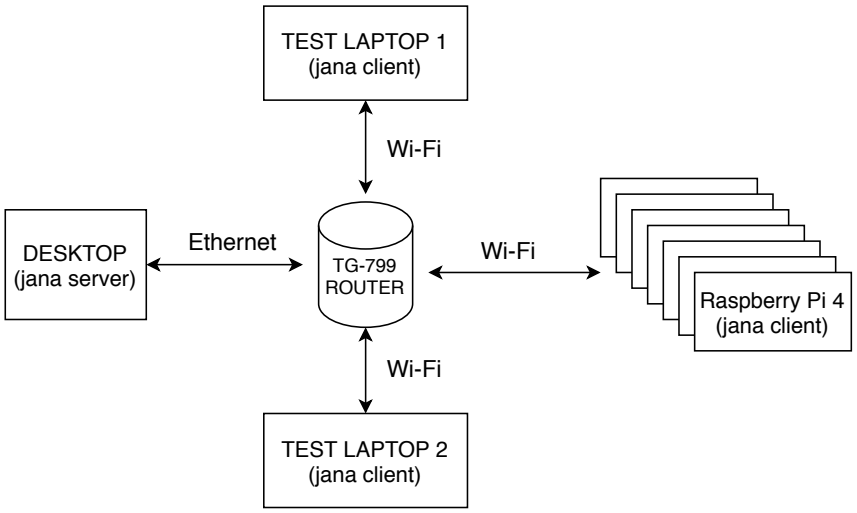


Figure 4.3: Experiment 3: Hacking the Wi-Fi driver setup overview

In any case, the primary objective is to observe if this field is of use. Figuring out exactly what "frame tx attempts time" is tricky. We can, however, run multiple experiments with varying payload sizes and active network clients to see if backoff or time spent transmitting data are included.

Our testbed, as shown in Figure 4.3, consists of

1. a TG-799 router
2. one desktop, running the jana server (connected via ethernet)
3. 20 Raspberry Pi 4, acting as network load generators (connected via Wi-Fi)
4. one laptop with modified Wi-Fi driver (connected via Wi-Fi)
5. one control laptop with modified Wi-Fi driver (connected via Wi-Fi)

The two laptops used the modified Wi-Fi driver to log the `wireless_media_time` throughout the test. Both laptops ran the jana client, sending UDP packets to the server running on the desktop, which was connected via ethernet. An ethernet connection was chosen since Wi-Fi uses a shared medium, which means that communication is simplex. By connecting the server via ethernet, we can assume that the cost of forwarding packets from Wi-Fi to Ethernet is so low that it can be ignored for our experiments. In addition to the two laptops we set up a number Raspberry Pi 4s to also run the jana client and send UDP packets to the jana server, with the intention increase the load on the CSMA algorithm and risk of Wi-Fi frame collisions, which in turn affects the backoff process.

The experiment was parameterized on payload size and number of active jana clients. Packet generation distribution was fixed to "as many as possible".

The hypothesis is that varying payload size would reveal if `wireless_media_time` contained any transmission timings, and varying the number of active jana clients (i.e varying medium contention) would reveal any backoff timings.

The number of clients were $5, 10, 15, 20$ and payload sizes varied from 1 byte to 32 kilobytes (1, 2, 4, 8, ...). Note that the maximum IEEE 802.11 frame size is around 1460 bytes, so any packet jana generates with a payload larger than approximately 1KB will result in multiple IEEE 802.11 frames being transmitted. Intuitively, the per-frame wireless media time should remain constant while the per-packet latency should increase linearly with the number of frames requires to fully transmit the packet.

4.6 Evaluating: In the Wild

Evaluating the model under controlled conditions ("in the lab") is only one part of the analysis. How the model performs "in the wild" is a natural next step and in this section we will describe the methodology behind our model evaluation, "in the wild".

Since we cannot expect end-users to load logging programs we must either run a logger on a network client or log directly from the router. We do not have access to a programmable radio so we will attempt to acquire metrics directly from the router.

The network interface metrics can be accessed through various different interfaces. In this thesis two programs were tried during exploratory testing, `ubus` and `quantenna api`. In the end `ubus` was chosen due to its broader range of statistics (both 2.4 and 5 GHz modems, compared to only 5 GHz for quantenna) and consistency of its output (no corrupt/invalid JSON was detected, compared to some corrupted output for quantenna). Since both interfaces report metrics from the same sources (from kernel, driver and firmware) we should get roughly equivalent data points no matter the interface.

In conclusion, network performance metrics will be collected by periodically calling the `ubus` program on the router.

4.6.1 Selecting ubus metrics

After deciding *how* to collect data, it is time to decide *what* data to collect.

As mentioned earlier, the model parameters form our comparison set. Data collected during testing needs to map directly or indirectly to the comparison set. It makes sense, then, to start with model parameters and try to find equivalent metrics.

- *network nodes (N)* - maps directly to number of connected network clients. We exclude the router from this number as it will never attempt to transmit volountarily.
- *channel bit rate* - the bit rate used to transmit and receive. Expressed in physical tx/rx rates in most drivers and indicated by the Modulation and Coding Scheme (MCS) index. It is important to note that even if IEEE 802.11 n can transmit up to 600 Mbps, the advertised bit rate will depend on signal strength, number of MIMO antennas, guard interval, bandwidth and other factors. Therefore, it is better to log the physical tx/rx rates actually used instead of using the theoretical maximum.

- *normalized throughput (U)* - is estimated in the model as average time spent transmitting payload vs. total system time. This is roughly equal to the average throughput of the system divded by the advertised channel bit rate. Average network throughput can be obtained by computing $8\frac{\text{RX}_{\text{bytes}}}{\text{time}}$ (from router's point of view).
- *packet collision probability (P)* - refers to the IEEE 802.11 frame collision probability and as far as we researched, must be obtained from the NIC. Our NIC does not provide this information. An alternative is to count the number of packets dropped by the NIC due to exceeding the maximum number of retransmission attempts. The model defines this property as P^{L+1}, where P is the conditional collision probability and L the retry limit.

In addition to these metrics, we decided to include the Received Signal Strength Indicator (RSSI), a vendor-specific metric which can be used to gauge signal strength, in case our experiments showed any unexpected behaviour.

4.6.2 Analysing ubus metrics

As good scientists we should not take data at face value. Thus we decided to conduct some experiments to sanity check the reported metrics. For each selected metric a test was performed to try and uncover any unexpected behavior.

- *network nodes (N)* - read directly from `ubus` output
- *channel bit rate* - physical rx/tx rates read directly from `ubus`
- *normalized throughput (U)* - the $\texttt{RX}_{\texttt{bytes}}$ was read directly from `ubus`
- *packet collision probability (P)* - dropped backets read directly from `iwconfig`

The test: log periodically on a router running iperf3 in server mode with one client running iperf3 in client mode during 12 hours.

4.6.3 RSSI experiments

Received Signal Strength Indicator (RSSI) is an important, and tricky, value. It is important because it is often the only indicator of signal strength, and tricky due to high variance between two chips, from same or different vendors, suggesting that variance stems from a combination of hardware and software factors [7].

In order to establish whether the RSSI metric could be used or not, analysis had to be done to verify the behavior, accuracy and variance of the reported values. This was done in a shielded lab with low-to-no external interference and no self interference from reflection.

Figure 4.4 shows how the router, antenna and laptop were set-up. Three experiment sessions were carried out to measure RSSI, about 5 minutes per session, with two different spectrum analysers.

The first test established the baseline rssi values when the system (laptop) was idle, i.e. any network load originated from background services. The second test was conducted using `iperf3` to push as much network load as possible, see

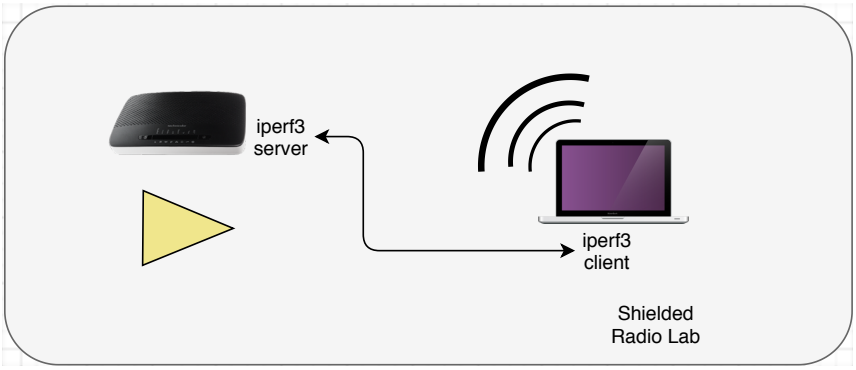


Figure 4.4: TG799 router and measurement antenna side by side, 1 meter from laptop

listings **??**, **??** and **??** for relevant scripts. The third test was a control test with a different spectrum analyser, but otherwise identical to the second test.

Scripts for parsing the spectrum analyser data and related heat map generator are detailed in listings **??** and **??**.

The *spectrum analyser data* was plotted as a heat map; frequency vs. time vs. signal strength. This graph shows both how signal strength varies over time and frequency. In contrast, the *router metrics* only specify one RSSI value and thus the resulting plot compares RSSI vs. time. The graphs can be analysed individually for insight into system behavior and compared to see how they relate.

We did not have time to get the second spectrum analyser to output data as a fuse went during our first attempt with it and we had to wait for it to be fixed. On the other hand, this spectrum analyser was used to calibrate and estimate the attenuation of the antenna cable. In the end this analyser was mainly used to eyeball that the first spectrum analyser was somewhat correct and to estimate cable attenuation.

4.6.4 Evaluation

Armed with a solid set of *router metrics* data collection could begin.

To compare our collected data against the model we first implemented the model as a program, see Listing 1 in the Appendix. As a preliminary test of the implementation we tried to compare our model with the numbers published in [6], and were surprised that only the *conditional collision probability* was correct: neither *normalized throughput* nor *channel access delay* were correct. Since this could be caused by different parametrisation we attempted to find possible parameters using a brute-force approach, but this resulted in skewed and unsound parameters. So we reached out to the authors of the model in order to try and obtain the original source code. After a week we got a reply that it could take a while to dig stuff that old up and waited, but nothing arrived and we received no response to subsequent follow-up emails.

Thus it became impossible for us to actually evaluate the model and so we

shifted focus more heavily onto collection of the neccesary metrics.

Chapter 5
Results

This chapter presents results from experiments 1, 2 and 3 as well as the `ubus` smoke test and RSSI analysis. A discussion of these results will follow in the subsequent chapter.

5.1 Experiment 1

The relation between computed T_{delay}, i.e. $(T_{pcap} - T_{sent})$, and T_{syscall} can be seen in Figure 5.1, where T_{pcap} is the time reported by libpcap, T_{sent} the timestamp right after `sendto` returns and $T_{syscall}$ the time it took for `sendto` to execute. This graph shows how our supposed "packet sent timestamp" was dominated by the time it took to complete the call to `sendto`, which could only block due to a full `sndbuf`.

5.2 Experiment 2

In Table 5.1 we present the recorded NIC serve rate $\lambda_{\texttt{NIC}}$, `qdisc` queue length $L_{\texttt{qdisc}}$, specified `sndbuf` and packet `payload` size. The values were obtained from a session with the system under test + 20 raspberry pi 4s.

Recall from Equations 4.2 and 4.3 how $L_{\texttt{NIC}}$ can be obtained and used with Little's Law to estimate $W_{\texttt{NIC}}$:

$\lambda_{\texttt{NIC}}$	4108
$L_{\texttt{qdisc}}$	427
`sndbuf`	1,5 MB
payload	1042 B

Table 5.1: Average serve rate and estimated queue length of the NIC.

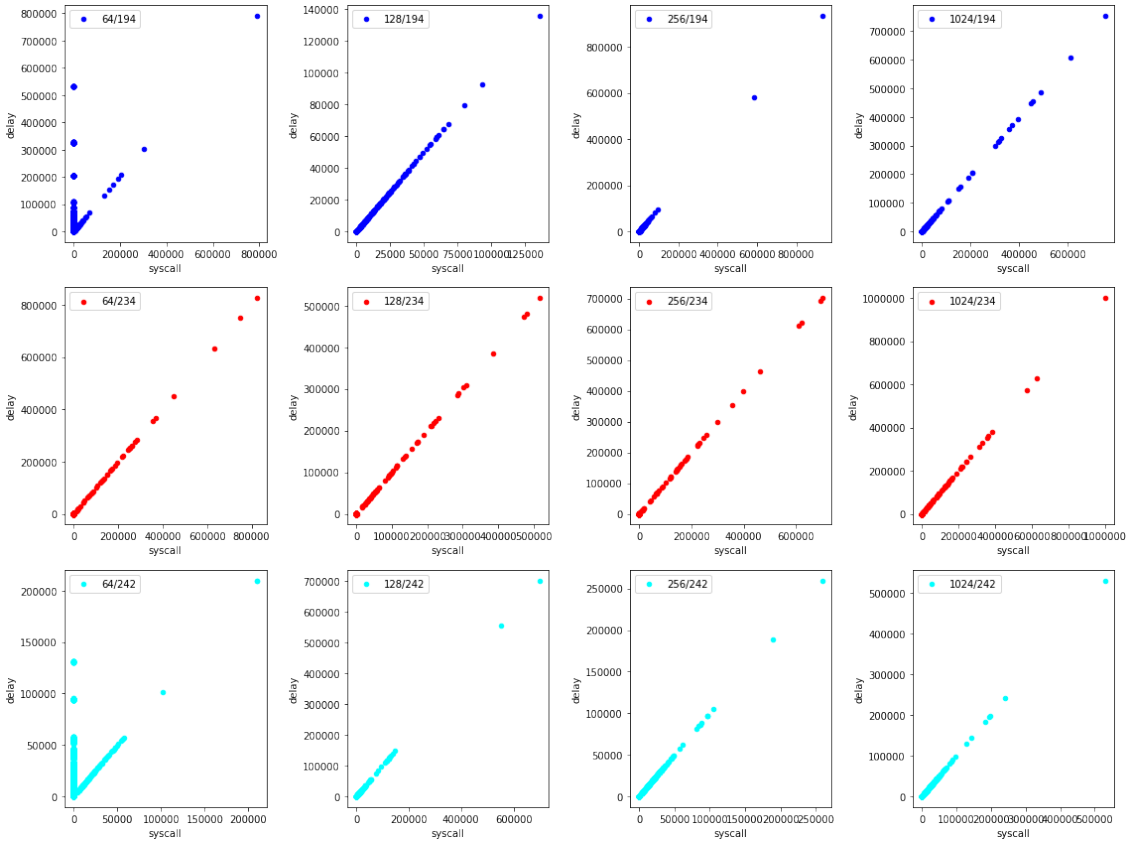


Figure 5.1: Scatter plot of T_{delay} and T_{syscall} (μs) for each packet sent. Tests go from left to right, with each column being one test with a total of 3 different machines (and thus three rows). The first number in each legend is the packet payload size.

$$L_{\texttt{NIC}} = \frac{\texttt{sndbuf}}{\texttt{payload} + \texttt{overhead}} - L_{\texttt{qdisc}} \approx \frac{1.5 \times 10^6}{1442} - 427 = 563$$
$$W_{\texttt{NIC}} = \frac{L_{\texttt{NIC}}}{\lambda_{\text{NIC}}} = \frac{563}{4108} = 137\ ms$$

Notice how $W_{\texttt{NIC}}$ determines that under a fully loaded scenario, it would take more than 100 *milli*seconds to transmit 1 KB using UDP. Our definition of `overhead`: IP + IEEE 802.11 + kernel bookkeeping.

5.3 Experiment 3

The final experiment was carried out on IEEE 802.11 *n* (2.4 GHz) and *ac* (5 GHz). In Figures 5.2 and 5.3 we present the total network throughput and `wireless_media_time` for IEEE 802.11 *n*. Similary, test results for IEEE 802.11 *ac* are shown in Figures 5.4 and 5.5.

Varying between each test is the number of actively participating clients as well as their payload size. No specific packet generation distribution was used, each client sent packets "as fast as possible".

It is important to note that, during this experiment, we could not control the presence of nearby networks. However, the experiment was performed during after-hours when no one was left at the office to run any traffic-heavy applications. Additionally, any signficant traffic caused by unmanaged software updates should show up as a significant outlier (much less throughput, and higher `wireless_media_time`) for during an affected experiment run.

In addition, running 60 tests (15 payload sizes × 4 host sizes) for IEEE 802.11 *n and ac*, takes a couple of hours and should therefore more easily show any temporary network impacts ony a single test run (combination of active nodes and payload size) in the throughput graphs.

5.4 `ubus` smoke test

The `ubus` smoke test did not yield any presentable results, since most of the test focused on verifying that no invalid data was returned.

No malformed output was noticed during our tests.

Inconsistencies on 5 GHz modem were first reported to us by Telenor, the 32-bit byte counter did not overflow correctly and will get "stuck" at $2^{32}-1$. Our findings initially showed similar behaviour for 2.4 GHz as well, but was later attributed to a JSON conversion step (probably represented as signed 32 bit integers, instead of unsigned) in our logging approach.

This has been mitigated in our experiments by using byte counter values from `iw` instead.

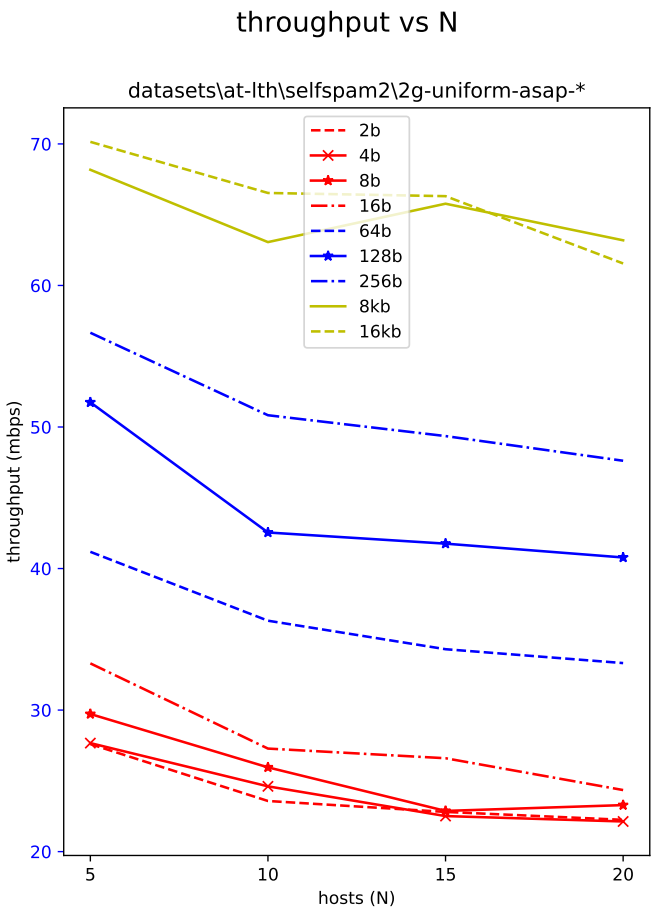


Figure 5.2: Experiment 3 - total network throughput (incl. UDP overhead) using IEEE 802.11 n.

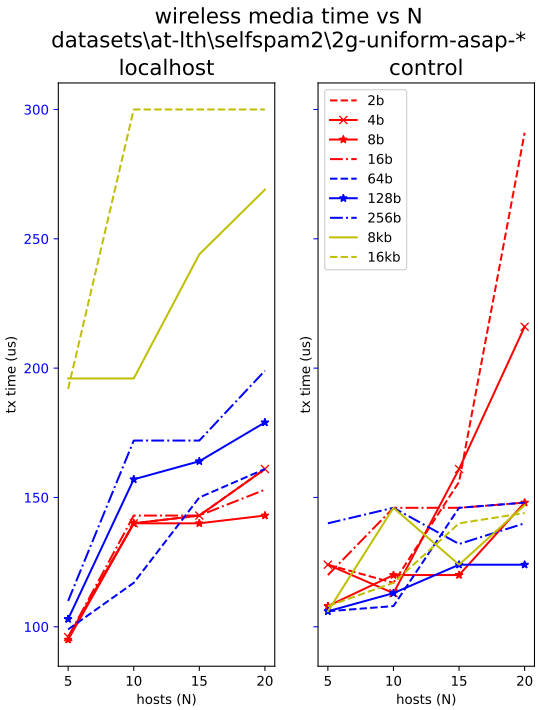


Figure 5.3: Experiment 3 - wireless media tx time captured on two devices using IEEE 802.11 n.

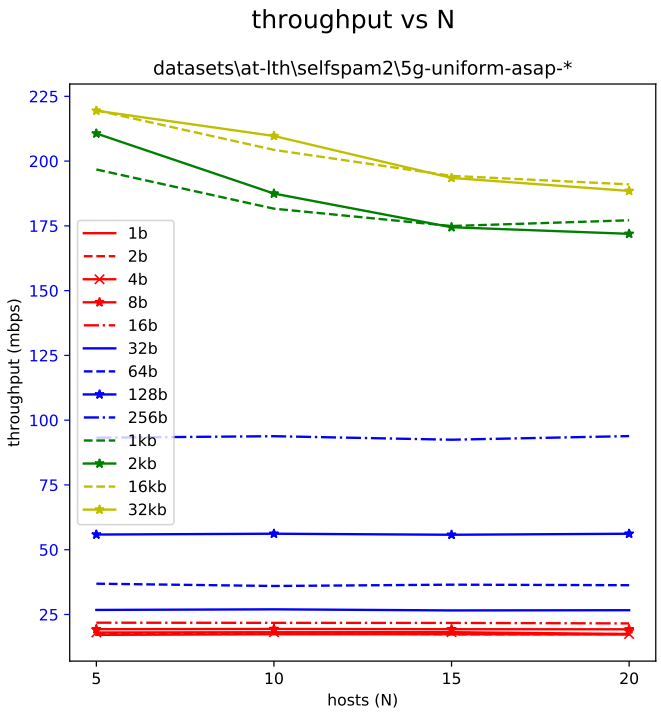


Figure 5.4: Experiment 3 - total network throughput (incl. UDP overhead) using IEEE 802.11 ac.

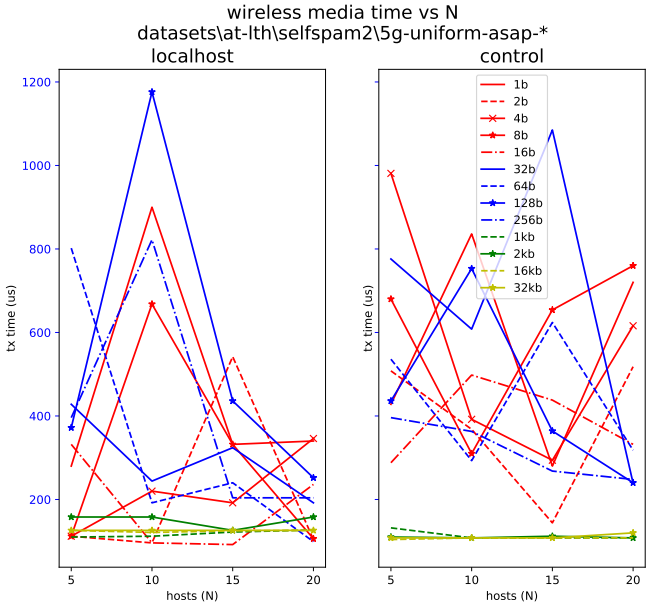


Figure 5.5: Experiment 3 - wireless media tx time captured on two devices using IEEE 802.11 ac.

5.5 RSSI

Our RSSI experiment results are shown in Figure 5.6 and 5.7. Graphs show RSSI values as reported by the HF-6080 spectrum analyser, by sweeping the 802.11 ac spectrum in 1 MHz resolution.

Figure 5.6 shows the frequencies of the 802.11 ac spectrum measurable by the device whereas Figure 5.7 only shows frequencies for the (80 MHz) channel used.

RSSI values *are* corrected for attenuation caused by the cable, average of 6 dB across 802.11 ac frequencies, obtained from a Rohde & Schwarz ZVL.

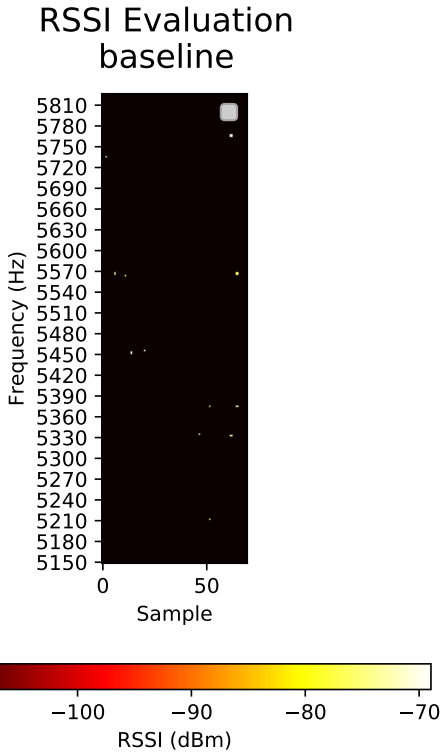


Figure 5.6: RSSI baseline experiment, background activity.

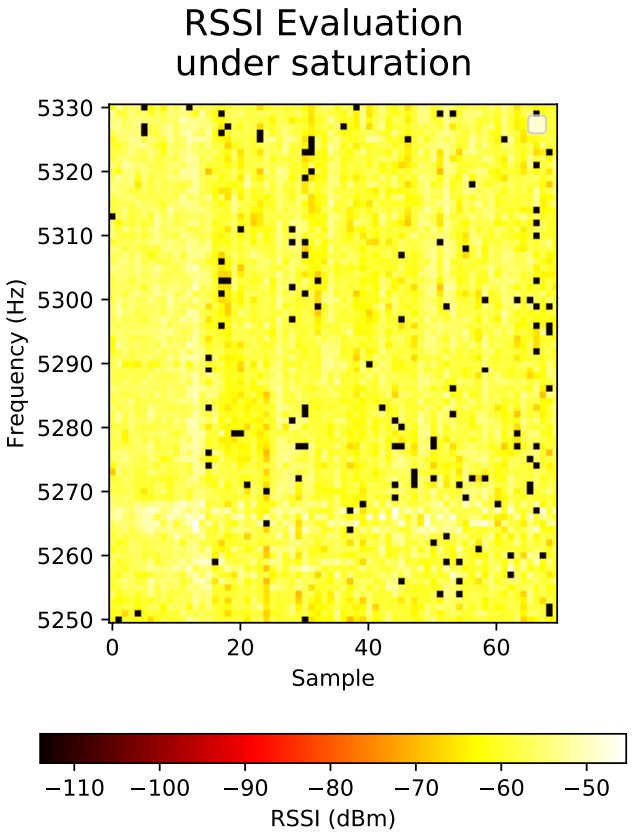


Figure 5.7: RSSI experiment under iperf3 saturation using one 80 MHz channel.

Chapter 6
Discussion and Future Work

In this chapter we first discuss the experiments and their results. Then we proceed to discuss the model based on experiments. Finally, we present some ideas on interesting research opportunities and our closing thoughts.

6.1 Discussion of results

In this section we will comment on the results obtained, primarily from our experiments. More importantly, we will also expand upon the discussion presented for each experiment in the Chapter 4 where we explained our experimental approach.

6.1.1 Experiment 1: Wireshark

As revealed in Figure 4.1 (actual timing model), the design of this experiment has a critical flaw which is not apparent without deeper insight into the network stack. As explained earlier, the "packet tap" feature from which Wireshark obtains packets is triggered when the packet exits the kernel queueing system (qdisc) and handed over to the driver for transmission, rather than once the driver receives a transmission event from the hardware. This completely invalidates the fundamental design model and thus the experiment as such.

Further exploration showed that the experiment design could work if packets could be timestamped in hardware and somehow captured after transmission. There *is* support for this in Linux, and Wireshark, but unfortunately relies on features not available in our hardware.

The presented results are mostly included for completeness and should not be regarded as anything useful, except for measuring system call to pcap latencies.

6.1.2 Experiment 2: Queueing the Network System

As described in Section 4.4, there are some issues with the definition of W_{NIC} from Equation 4.4 due to a deferred memory release mechanism in the kernel. Specifically, the definition of the NIC queuing system from Figure 4.2 includes a note "kernel scheduler frees handled pkts".

In order to increase throughput (at the expense of latency), Linux will free up handled packets in batches. This part greatly impacts the `sndbuf` which is used

in Equation 4.2, implying that sent or dropped packets – but not yet freed – will still count as enqueued in the NIC. In short, the modeled L_{NIC} includes packets in driver/firmware, in hardware and, crucially, in the kernel's to-free list.

By increasing the `sndbuf` we could observe a corresponding increase in L_{NIC} (and W_{NIC}) in our experiments. However, this fact does not imply that there are not any more packets in the NIC. Recall that the NIC queueing system actually consists of a driver/firmware queue and a hardware queue. The hardware queue on most consumer chips is *probably* either fixed and very limited in size, or non-existant (RAM *is* expensive).

After additional analysis of the source code of Intel's open source Wi-Fi driver, the driver/firmware queue length was determined to be 256 packets large. By assuming a full `sndbuf` (i.e. K packets in the network system) and a full NIC, we can model K as $K = L_{\text{qdisc}} + 256 + \text{hardware} + \text{to-be-free}$ where `hardware` and `to-be-freed` are unknowns.

One can argue that for the hardware queue to have any impact on the other queues, it would have to be of similar size. Otherwise, it should be safe to assume an interrupt-based solution: a driver manages known packets and hardware signals which have been sent using (soft) interrupt requests (IRQ), i.e. no dedicated hardware buffer. Thus the total queue size of the NIC is determined by the driver, in our case 256 packets, simplifying $K = L_{\text{qdisc}} + 256 + \texttt{to-be-freed}$. This exposes the core issue with the original queue model: there is no separation between "in queue, waiting to be served" and "served, waiting to be freed".

In the program which logged L_{qdisc}, we could observe how many packets were fed into the driver "at once". Our tests show that multiple, 10+, packets being handed over to the driver between timestamps $< 10\mu s$. This confirms that the driver is able to keep the internal buffer/queue fully saturated, but does not shed any light on the size of `to-be-freed`.

We can estimate the range of `to-be-freed` to $[K - L_{\text{qdisc}}, K - 256 - L_{\text{qdisc}}] = [307, 563]$, depending on the amount of packets in the NIC waiting to be transmitted. Reworking Table 5.1 ($L_{\texttt{NIC}} = [1, 256]$) we now obtain a $W_{\texttt{NIC}} = \frac{256}{\lambda_{\texttt{NIC}}} \approx [240\ \mu s, 62\ ms]$, again depending on the amount of packets in the NIC waiting to be transmitted.

In conclusion, this experiment, under assumption of saturation, no hardware buffer on the NIC, $L_{\texttt{NIC}} = 256$ packets, can estimate the range of $W_{\texttt{NIC}}$. As mentioned earlier, comparing the obtained values from our experiment with the Felemban-Ekici model is prevented by different network speeds and configurations.

6.1.3 Experiment 3: Hacking on the driver

Since we could not find a concise definition of the `wireless_media_time`, and ultimately argued that the value must include $T_{BACKOFF}$ if there is a positive correlation with regards to the number of nodes, and include T_{FRAME} if there is a linear correlation with regards to the payload size.

The sampled `wireless_media_time` presented in Figure 5.3 (2.4 GHz) shows an interesting behaviour: the timing data from the system under test (localhost) clearly increase as the number of active raspberry pi nodes increases. This behaviour is not apparent in the control set (values sampled from another system

running the same kernel but with another NIC), as the timing values do not strictly increase. A statistical analysis of the datasets could determine the level of correlation.

It looks like the payload size is a contributing factor the media time on the localhost dataset, but not on the control dataset. The relative media time factor is around 2X while the payload size factor is between $64X$ to $8000X$, and the absolute difference is around 100 μs, which amounts to about 100 μS$\times$144 Mbps = 14.4 Kb. Thus we can immediately conclude that there is no linear correlation between payload size and media time. Our best guess is that the NIC probably does about 100 μs worth of work between each (s)IRQ (which is when we sample). This guess does not seem to hold for the control set, however.

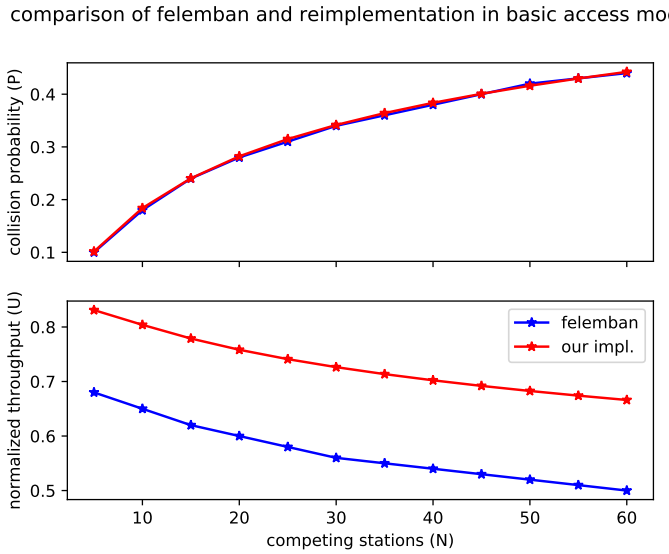
Moving on to 5G, Figure 5.5 presents a completely different behavior and the only similarity with 2G is the shared minimum media time, at around 100 μs. There does not seem to be any immediate correlation between either payload size and media time, nor number of nodes and media time. Even though we label this test as inconclusive, at least both the localhost dataset and control dataset show very similar behaviour. Comparing 2G and 5G modem behaviour is very tricky, not only due to a difference in protocol but often also due to a difference in hardware, firmware and kernel driver. It is worth noting that the difference observed suggest that there is no silver bullet which works for both IEEE 802.11 n (2G) and IEEE 802.11 ac (5G).

Indeed, differences between IEEE 802.11 n and IEEE 802.11 ac can have dramatic effects on observable metrics. Recall that during the CSMA process, time is quantized into time-slots, from 9 to 50 μs. At similar physical transmission rates, with $T_{\texttt{slot}} = 9$ will *attempt* to decrement the $T_{\texttt{backoff}}$ more than 5 times as quickly, compared with $T_{\texttt{slot}} = 50$. However, since transmission take exactly the same amount of time, the only difference becomes the time spent waiting for the `DIFS` and `SIFS`. Now consider that when $T_{\texttt{slot}} = 9$, the physical transmission rate is 200 Mbps and 72.2 Mbps when $T_{\texttt{slot}} = 50$. This is IEEE 802.11 n compared to IEEE 802.11 ac.

The insight here is that a IEEE 802.11 ac network can send frames a) faster compared to n and b) decrements the backoff counter more rapidly, with the implication that, network-wide, there will be more transmission attempts and, consequently, packet collisions, compared to IEEE 802.11 n, assuming network saturation and equal sample durations.

This behavior is implied in graphs, where the media time for small packets in IEEE 802.11 ac is significantly higher compared to IEEE 802.11 n (a large media time should primarily be caused by collisions and thus, by definition, frame drops). We will continue to discuss the frame drop probability in the upcoming section.

In conclusion, we found that the experiment could show expected correlation between $T_{BACKOFF}$ and media time for 802.11 n. This was, however, not observed on our control system and therefore we have to conclude that our findings are too inconclusive, with a saving clause that the experiment shows promise.



although

Figure 6.1: Our model reimplementation compared to values extracted from the original paper under similar network conditions.

6.2 The Felemban-Ekici Model

As it turned out, evaluating the usefulness of the Felemban-Ekici model was an incredibly difficult endeavour. Unfortunately, the first step of the thesis – gathering data to evaluate the model with – turned out to be the real challenge, and challenging problems have their own lure – they hook you in until they are solved, or, as in our case, time runs out.

To our disappointment, we could not discover any combination of parameters in our re-implementation of the Felemban-Ekici model which resulted in similar throughput and access delay results as the original paper. As seen in Figure 6.1, we could, however, accurately implement the conditional packet collision P.

To put this into perspective, at rougly 50 connected clients the conditional packet collision probability reaches 40%, which results in a $0.4^{L+1} \approx 0.06\%$ probability to fail transmission $L+1$ times and drop a packet. At 20 clients, which is what we used during experiment 3, this falls to $0.27^{L+1} \approx 0.003\%$.

Since clients are assumed to be identical, the packet drop probability for one client and full network are equivalent. In Figure 6.2 we show the total number of `tx_fails` (as reported to the kernel) relative to the total number of packets transmitted, for 802.11 *n*. Data from 802.11 *ac* is shown in Figure 6.3. As seen directly, the clients have a problem staying connected to the network throughout the 802.11 *n* test (indicated by lack of plots) and only few or no packet drops were detected. The 802.11 *ac* test, however, hovers around the 0.003, a full two orders of magnitude larger than the model's estimated 0.003%.

Finally, we tried to rescale and normalize the throughput by assuming that

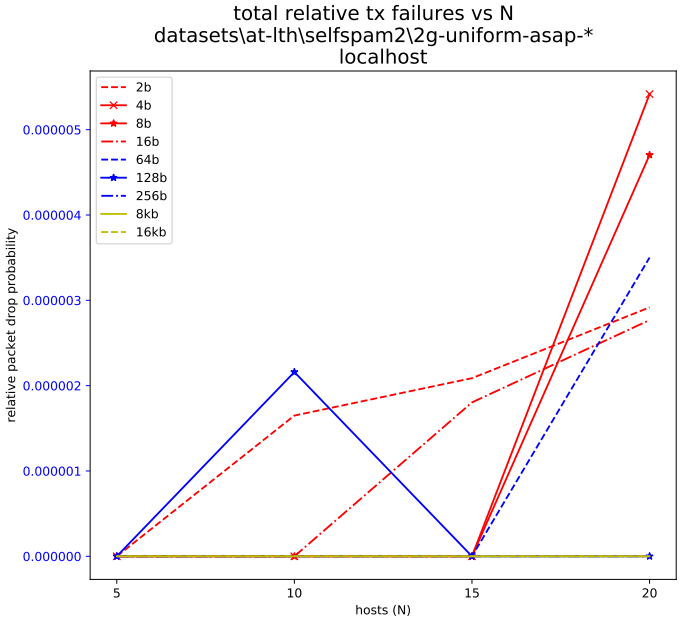


Figure 6.2: Empirically obtained packet drop probability for 802.11 n.

the model was true. The model estimates about that at $N = 5$, normalized throughput is at 68%, which puts channel capacity at $\frac{\text{throughput}}{0.68}$. Figures 6.4 and 6.5 show the (rescaled) normalized network throughput (using the computed channel capacity), for varying N and payload size. In both figures, the cyan curve show the normalized throughput obtained from the Felemban-Ekici paper with 1 KB payload at 1 Mbps (over 802.11 n).

Figure 6.4 (802.11 n) shows an expected behavior. Larger payload sizes have higher relative utilization and roughly follow the Felemban-Ekici curve. Lower payload sizes fall behind very fast. This hints at a weakness in the model regarding different traffic flows.

Seemingly in opposition, Figure 6.4 appears to imply that smaller payload sizes incur no penalty as N increases. Figure 5.4 reveals why, the throughput is already incredibly low. Moving on, the larger payload sizes show similar curves compared to 802.11 n.

In conclusion, we cannot determine whether the predicted conditional packet collision probability is correct or not. Although the model appears to predict network throughput degredation as the number of connected nodes increase, it's certainly not an apples to apples comparison. Our 802.11n network was mostly stable at 72.2 (for raspberri pi 4's) and 200 Mbps on 802.11 ac, quite different than model's 1 Mbps channel. As we could not generate normalized throughput values from our reimplementation we cannot know what the output of the model would have been for our network configurations.

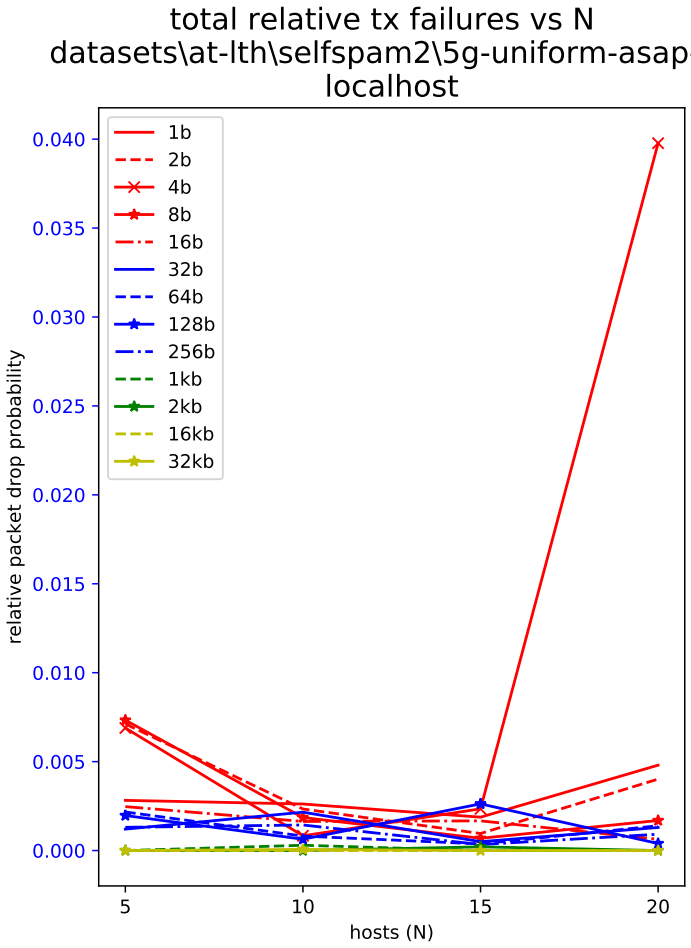


Figure 6.3: Empirically obtained packet drop probability for 802.11 ac.

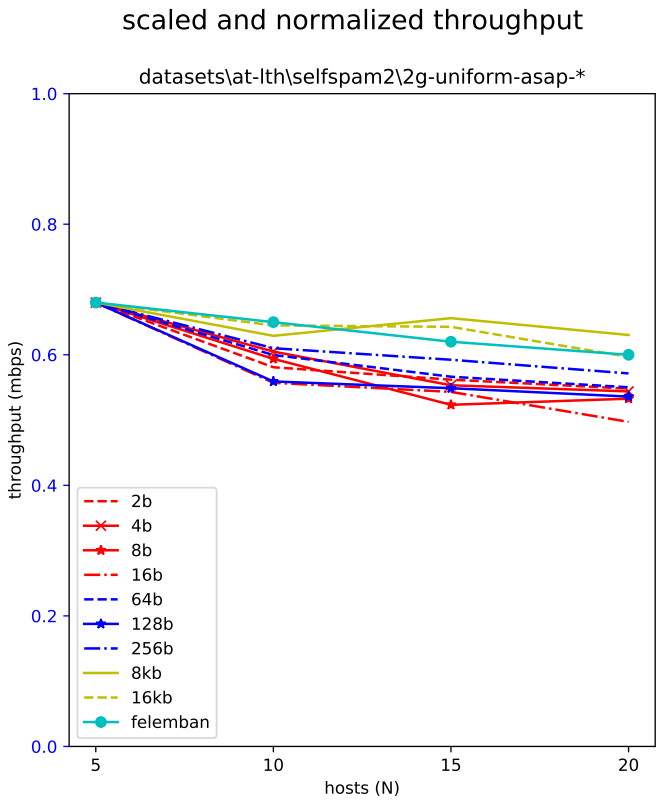


Figure 6.4: Rescaled, (measured) normalized network throughput for 802.11 n compared with Felemban-Ekici.

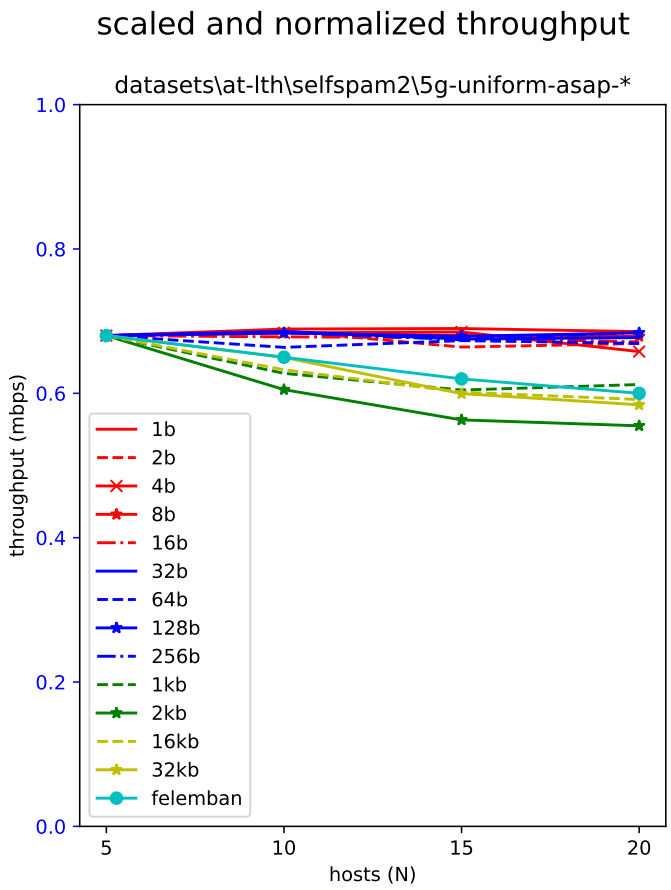


Figure 6.5: Rescaled, (measured) normalized network throughput during 802.11 ac compared with Felemban-Ekici.

6.3 Future work

- As seen in the network overview (Figure 2.4) and noticed during experiment 2, system call overhead and the userland/kernel split have serious implications for latency-sensative measurements. An area worth exploring is known as "userland networking" – a technique common in high-performance packet processing in which a userland application is allowed exclusive access of a network device. Bypassing the kernel removes potential bottlenecks such as context switches, memory allocation and copying, scheduling and multithreading. In theory, this technique can reduce the number of system calls of an application. More importantly, such an application can design the communication between NIC and userland as a ring buffer, eliminating the cost associated with memory allocation and deallocation experienced in Experiment 2 (the unknown number of `to-be-freed` packets).
- An alternative approach to bypassing the kernel, is to bring the userland program *into* the kernel. The Berkley Packet Filter (BPF) is a register-based filter evaluator designed for packet filtering [11]. It has since been extended and redesigned under the "extended BPF" (eBPF) moniker as an in-kernel virtual machine with filters, taps and hooks all over the kernel. In short, the eBPF machine can run arbitrary code inside the kernel, triggered by specific events. In theory, this should allow for more accurate measurement of the packet transmission process.
- Continuing on the started path with a modified Wi-Fi driver, it would be useful to also perform similar tests where the `jana` server also has to reply back to the clients. These tests should more closely mimic the TCP protocol upon which most of today's networking rely. Measuring the round trip time (RTT) of packets and comparing the measured values with a modeled RTT (based on IEEE 802.11 and TCP) could also lead to some interesting results.
- And as is common practice today: when in doubt, throw machine learning at the problem. While machine learning cannot enable more accurate measurements, a machine-learning model could be trained to predict network behaviour based on a large set of collected and labeled (as in, metric X indicate outcome Y) metrics. A model may also be constructed for determining possible user-friendly interventions (move device closer to router, the channel is currently observing heavy interference so switch channel), however an expert system (guided by real-time metrics) would probably be sufficient. The system could be designed to run on each device (distributed) or on a trusted machine (centralised). A centralised machine would have more information to act on when issuing interventions back to users and, potentially, directly to the Wi-Fi routers. Such a system could easily be extended to record the outcome of any interventions (e.g. significantly better or worse), possibly allowing for collection of data for unsupervised machine learning.

  During our thesis work we were able to identify several interesting metrics, available from `ubus`, such as number of connected clients, neighbouring network data, medium availability, TX/RX PHY rates, RSSI, glitches and RTS

settings. See `https://github.com/smeets/thesis/blob/master/paramlist.md` for more parameters and their descriptions.

6.4 Closing remarks & thoughts

This thesis primarily indicates that it currently is and most probably will become increasingly difficult to evaluate and use existing models for practical Wi-Fi equipment. As the IEEE 802.11 specification evolves, new Wi-Fi routers will have new implementations and new features. At least, there are probably fewer chipset makers than router equipment manufacturers, so the expected variance (of the chipset and driver) of these future devices should be similar to today's devices.

Our attempts to obtain highly accurate packet timing data also show that it requires a non-trivial amount of effort. It seems more likely that an overall measurement using ML/AI system is easier to obtain for helping customers with their routers than trying to develop a system that works from the inside out as there are too many gaps between model and physical reality.

However, there is a kind of natural elegance to the way the Bianchi (and subsequent models) approach network performance. By focusing on the (assumed) bottleneck–the Distributed Coordination Function–most of the recent improvements in throughput (e.g. batching, MI/MO, multi-user MI/MO) can simply be abstracted as channel capacity (which can be reliably sampled from the driver). In addition to this versatile way of modelling different performance features, the models have excellent performance in Wi-Fi simulations. We therefore believe that is is important to construct a functional implementation– which we weren't able to do–and run experiments in consumer networks.

References

[1] Ieee standard for wireless lan medium access control (mac) and physical layer (phy) specifications. *IEEE Std 802.11-1997*, pages 1–445, Nov 1997.

[2] 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 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, Oct 2009.

[3] Ieee standard for information technology– telecommunications and information exchange between systemslocal 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-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, Dec 2013.

[4] G. Bianchi. Performance analysis of the ieee 802.11 distributed coordination function. *IEEE Journal on Selected Areas in Communications*, 18(3):535–547, 2000.

[5] P. Chatzimisios, A. C. Boucouvalas, and V. Vitsas. Ieee 802.11 packet delay-a finite retry limit analysis. In *GLOBECOM '03. IEEE Global Telecommunications Conference (IEEE Cat. No.03CH37489)*, volume 2, pages 950–954 Vol.2, Dec 2003.

[6] E. Felemban and E. Ekici. Single hop ieee 802.11 dcf analysis revisited: Accurate modeling of channel access delay and throughput for saturated and unsaturated traffic cases,. *IEEE Transactions on Wireless Communications*, 10(10):3256–3266, 2011.

[7] G. Lui, T. Gallagher, B. Li, A. G. Dempster and C. Rizos. Differences in rssi readings made by different wi-fi chipsets: A limitation of wlan localization. *International Conference on Localization and GNSS (ICL-GNSS)*, pages 53–57, 2011.

[8] Haitao Wu, Yong Peng, Keping Long, Shiduan Cheng, and Jian Ma. Performance of reliable transport protocol over ieee 802.11 wireless lan: analysis and enhancement. In *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 599–607 vol.2, June 2002.

[9] Greg Kroah-Hartman Jonathan Corbet, Alessandro Rubini. *Linux Device Drivers, 3rd edition*. O'Reilly Media, 2005.

[10] The kernel development community. The Linux Kernel/Linux Networking Documentation. `https://www.kernel.org/doc/html/latest/networking/index.html`, 2019. [Online; accessed 15-May-2019].

[11] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, page 2, USA, 1993. USENIX Association.

[12] packagecloud. Monitoring and Tuning the Linux Networking Stack: Sending Data. `https://blog.packagecloud.io/eng/2017/02/06/monitoring-tuning-linux-networking-stack-sending-data/`, 2017. [Online; accessed 15-May-2019].

[13] E. Smith. Global broadband and wlan (wi-fi) networked households forecast 2009-2018. *Strategy Analytics*, 2014.

[14] Yang Xiao. Performance analysis of priority schemes for ieee 802.11 and ieee 802.11e wireless lans. *IEEE Transactions on Wireless Communications*, 4(4):1506–1515, July 2005.

[15] Liang Zhang, Yantai Shu, Oliver W. W. Yang, and Guang-Hong Wang. Study of medium access delay in ieee 802.11 wireless networks. *IEICE Transactions*, 89-B:1284–1293, 04 2006.

[16] Huan Zhou, Hui Wang, and Xiuhua Li. A survey on mobile data offloading technologies. *IEEE Access*, PP:1–1, 01 2018.

```
1  import math
2  import numpy as np
3
4  from scipy.special import binom as binomial
5  from numpy.linalg import norm
6
7  """
8  Solve the steady state via the power method.
9  """
10 def solve_steady_state(pi, epsilon=1e-8, max_iter=1e5):
11     P = pi.T
12     size = P.shape[0]
13     A = np.zeros(size)
14     A[0] = 1
```

```
    i = 0

    while True:
        A1 = P.dot(A)
        A = P.dot(A1)

        n = norm(A - A1, 1)
        i += 1
        if n <= epsilon or i >= max_iter:
            return A

def suml(fn, limits):
    lo, hi = limits
    val = 0
    for i in range(lo, hi+1):
        val = val + fn(i)
    return val

class Felemban():
    """
    Felemban distributed coordination function model.

    from models.felemban import Felemban
    x = Felemban()
    x.setup(*x.compute())
    x.simulate()
    """
    def __init__(self, L=7, N=3):
        self.L = L
        self.N = N
        self.setup(tau=0, P=0, Pf=0)

    def setup(self, tau, P, Pf):
        """
        Set model parameters.
        Parameters can be derived from the compute
            method.

        Keyword arguments:
        tau -- packet transmission probability
        P -- packed dropped probability
        Pf -- packet freeze probability
        """

        self.tau = tau
        self.P = P
        self.Pf = Pf
```

```
61
62      def compute(self, tau0=0.0, epsilon=1e-6,
63                  CW_min=32, CW_max=1024):
64          """
65          Calculate model parameters iteratively.
66
67          Keyword arguments:
68          tau0 -- initial guess for tau (0 < tau0 < 1,
                default 0.0)
69          epsilon -- desired precision (default 1e-6)
70          CW_min -- ieee specified congestion window (
                default=32)
71          CW_max -- ieee specified congestion window (
                default=1024)
72          """
73
74          # Pe is defined to be 1 in the paper
75          pe = 1
76
77          # IEEE specification
78          m = int(math.log(CW_max/CW_min, 2))
79
80          L = self.L
81          N = self.N
82
83          # guess initial value of tau
84          tau = tau0
85
86          # alpha is specified in paper to 0.5
87          alpha = 0.5
88
89          def W(j):
90              assert(j >= 0)
91              assert(j <= L)
92              return 2**j*CW_min if j < m else CW_max
93
94          # Equation (6)
95          def Q(n):
96              return binomial(N-1, n) * (tau**n) * (1 -
                    tau)**(N - n - 1)
97
98          while True:
99              # Equation (2) - aka. p_tau(tau, )
100             P = 1 - (1 - tau) ** (N - 1)
101             Pdrop = P**(L+1)
102             if Pdrop == 1:
103                 print("warn␣Pdrop=1,␣tau={},␣P={}\n".
```

```
                format(tau, P))
104
105             pei = (1 - tau)**(N-1)
106
107             # Equation (3)
108             pes = binomial(N-1, 1) * tau * (1 - tau)**(N
                    -2)
109             pec = 1 - pei - pes
110
111             pss = 1/W(0)
112             psi = 1 - pss
113
114             # Equation (8)
115             CW_avg = suml(
116                 lambda i: (1-P) * (P**i) * W(i)/(1-Pdrop
                        ),
117                 (0, L))
118
119             # Equation (7)
120             pci = suml(lambda n: Q(n) * (1 - 1/CW_avg)**
                    n, (2, N-1))
121
122             # Equation (9)
123             pcs = suml(
124                 lambda n: Q(n) * n * (1/CW_avg) * (1 -
                        1/CW_avg)**(n-1),
125                 (2, N-1))
126
127             pcc = 1 - pci - pcs
128
129             # Equation (10)
130             pi = np.array([
131             [pei, pes, pec],
132             [psi, pss, 0],
133             [pci, pcs, pcc]
134             ])
135
136             # A = [ Pi Ps Pc ]
137             A = solve_steady_state(pi, epsilon)
138             PI = A[0]
139
140             # Equation (4)
141             Pd = PI * pe
142
143             # Equation (5)
144             Pf = 1 - Pd
145
```

```
146             # Equation (1)
147             #tau_newp =(1 - P**(L+1)) / ((1 - P) * sum
                    ([1 + (1/(1-Pf)) * sum([(W(j) - k)/W(j)
                    for k in range(1,W(j))]) * P**j for j in
                    range(0,L+1)]))
148             tau_new = (1 - P**(L+1)) / ((1 - P) * suml(
149                 lambda j: (1 + (1/(1-Pf)) * suml(
150                     lambda k: (W(j) - k)/W(j),
151                     (1, W(j)-1))
152                 ) * P**j,
153                 (0, L)))
154
155             # tau_i = alpha*tau_{i-1} + (1-alpha) *
                    tau_new
156             tau_old = tau
157             tau = alpha * tau_old + (1 - alpha) *
                    tau_new
158             if abs(tau - tau_old) <= epsilon:
159                 break
160
161         return tau, P, Pf
162
163     def U(self, bps=1e6, access_mode="basic", payload
            =8192,
164           slot_idle=50):
165         """
166         Normalized channel throughput.
167
168         Keyword arguments:
169         bps -- channel bit rate in bits per second (
                default=1Mbps)
170         access_mode -- either "basic" or "rts" (default
                ="basic")
171         payload -- packet payload, in bits (default
                =8192)
172         slot_idle -- idle slot time, in microseconds (
                default=50)
173         """
174         channel_bit_rate = bps
175
176         # IEEE frame sizes
177         MAC = 272 * 8        # bits
178         PHY = 128 * 8        # bits
179         ACK = 112 * 8 + PHY # bits
180         RTS = 160 * 8 + PHY # bits
181         CTS = 112 * 8 + PHY # bits
182
```

```
183         # IEEE guard times
184         DIFS = 128*1e-6 # s
185         SIFS = 28*1e-6 # s
186
187         # Transmission duration of RTS, CTS and ACK
                packets
188         TRTS = RTS / channel_bit_rate # s
189         TCTS = CTS / channel_bit_rate # s
190         TACK = ACK / channel_bit_rate # s
191
192         # P(channel is busy)
193         Pb = 1 - (1 - self.tau)**self.N
194
195         # P(begin successful transmission)
196         Ps = self.N*self.tau*(1 - self.tau)**(self.N-1)
197
198         # Transmission duration of headers
199         Th = (MAC + PHY) / channel_bit_rate # s
200
201         # Transmission duration of payload
202         Tp = payload / channel_bit_rate # s
203
204         # Duration of idle time slot
205         Ti = slot_idle*1e-6 # s
206
207         if access_mode == "rts":
208             # use rts/cts access mechanism
209             Ts = DIFS + TRTS + SIFS + TCTS + SIFS + Th +
                    Tp + SIFS + TACK
210             Tc = DIFS + TRTS + SIFS + TCTS
211         elif access_mode == "basic":
212             # basic access mechanism
213             Ts = Tc = DIFS + Th + Tp + SIFS + TACK
214         else:
215             # wat
216             pass
217         print("Pb={}\nPs={}\nTh={}\nTp={}\nTi={}\n".
                format(Pb, Ps, Th, Tp, Ti))
218         return (Ps*Tp)/((Ps*Ts) + (Pb-Ps)*Tc + (1-Pb)*Ti
                )
219
220     def print_stats(self):
221         P = 1 - (1 - self.tau) ** (self.N - 1)
222         print("N={}, tau={}, P={}\n".format(self.N, self
                .tau, P))
223
224 for N in range(5,70,5):
```

```
    x = Felemban(N=N, L=7)
    x.setup(*x.compute())
    x.print_stats()
```

Listing 1: Felemban-Ekici model implementation