Classifying Rate Adaptation Algorithms in IEEE 802.11b/g/n Wireless Networks

Master Thesis

Tor Martin Slåen

August 15, 2012
Classifying Rate Adaptation Algorithms in IEEE 802.11b/g/n Wireless Networks

Tor Martin Slåen

August 15, 2012
Abstract

This thesis provides a tool for helping wireless network professionals to recognize different Rate Adaptation Algorithms (RAA) implemented in IEEE 802.11 wireless network device drivers. The RAA used in the wireless network adapters are responsible for selecting the bit-rate used by the hardware when transmitting frames over the wireless channel. This algorithm heavily affects the performance of wireless devices.

We present the Rate Adaptation Classifier (RAC) which passively listens to data traffic between wireless stations. Based on the observed traffic, RAC performs logging and statistics. The final output of the application can be used to classify the rate adaptation algorithm used by the observed wireless device. RAC can be used on any platform which exports the correct headers to user-space through the PCAP framework. RAC has the ability to listen to and analyse any IEEE 802.11b/g wireless network and contains code to perform basic statistics on IEEE 802.11n.

RAC captures and logs the important pieces of observed data traffic and is not affected by the encryption used by the wireless network. RAC is only interested in the Physical (PHY) and some Link-Layer information exported by the monitor interface.

We present a series of validation tests, analyse the results obtained from RAC and compare these results against the theoretical expected behaviour of each RAA. We show that the results produced are directly comparable to the RAAs expected behaviour. We will also present the results from a series of experiments where we test Rate Adaptation Classifier (RAC) and the proposed method to match its output to known rate adaptation algorithms.

Thesis Supervisors: Prof. Michael Welzl and Naeem Khademi.
Preface

This work is the result of a 60 point master thesis project at the University of Oslo, Institute of Informatics. The project was performed by Tor Martin Slåen in the time period 2011 – 2012.

Special thanks to Naeem Khademi for the support and guidance throughout the thesis. Both Naeem and my main supervisor Prof. Michael Welzl deserves special thanks for reviewing the work and providing feedback before final delivery.

Many thanks to all my fellow students at the Network and Distributed systems lab. You have all been a great help and a fantastic source of encouragement throughout the project.

Thanks to my family for their support.
Contents

I Introduction 1

II Background and literature review 5

1 Related work 7

2 IEEE 802.11 standard 9

2.1 Physical layer 9

2.1.1 Modulation Techniques 9

2.1.2 Extended modulation techniques 11

2.1.3 Long and short preamble 12

2.2 Media Access Control 13

2.2.1 RTS/CTS 13

2.2.2 Frames Types 14

2.3 Bit-rates in 802.11a/b/g 16

2.4 Bit-rates in 802.11n 16

2.4.1 MCS index 18

2.4.2 Rate Selection 18

3 Rate Adaptation 21

3.1 PHY-based algorithms 21

3.1.1 Receiver-Based AutoRate (RBAR) 22

3.2 Link-Layer-Based algorithms 22

3.2.1 Automatic Rate Fallback (ARF) 23

3.2.2 Adaptive ARF (AARF) 23

3.2.3 Adaptive Multi-Rate Retry (AMRR) 24

3.2.4 Onoe 24

3.2.5 SampleRate 25

3.2.6 Minstrel 25

3.2.7 PID 26

3.2.8 MiSer 26

3.3 MAC 802.11 framework 26

III Rate Adaptation Classifier (RAC) 27

4 Design 29

4.1 Fingerprinting Rate Adaptation Algorithms 29
V Evaluation Results

8 Classifying Known Algorithms

8.1 Madwifi validation tests

8.1.1 Analysing AMRR

8.1.2 Analysing Onoe

8.1.3 Analysing SampleRate

8.1.4 Analysing Minstrel

8.2 Measured RAA characteristics

8.2.1 Madwifi

8.3 Comparing Rate Adaptation Algorithm Behaviour

VI Conclusion and future work

Glossary
List of Figures

2.1 802.11 Frame Components ........................................ 14
2.2 General MAC Frame Format ........................................ 15
2.3 A-MSDU ................................................................. 15
2.4 A-MPDU ................................................................. 16

5.1 Sampling event ......................................................... 54
5.2 Sampling event with immediate following best rate change 54

7.1 Validation tests: Fixed AP ........................................... 67
7.2 Validation tests: Moving AP. 10 and 50 is the time (in seconds) where the AP is located during the test. .......... 67

8.1 AMRR Rate Changes .................................................. 74
8.2 Onoe Rate Changes ................................................... 77
8.3 RAC rate changes per second deficiency ......................... 77
8.4 SampleRate Rate Changes ......................................... 83
8.5 Minstrel Rate Changes ............................................... 88
List of Tables

2.1 MAC Frame Address Field Contents .......................... 15
2.2 Bit-rates in IEEE 802.11 ........................................ 17
2.3 Radiotap defined-fields/Rate ............................... 17
2.4 Radiotap defined-fields/MCS .................................. 17
2.5 Radiotap defined-fields/MCS/known ......................... 18
2.6 Radiotap defined-fields/MCS/flags ......................... 19
2.7 Radiotap MCS index table ................................... 19

4.1 Madwifi Retry Chain ............................................ 30
4.2 AMRR Retry Chain with best rate of 36Mbps. ............... 31
4.3 AMRR Retry Chain ............................................. 34
4.4 Onoe Retry Chain .............................................. 35
4.5 SampleRate Retry Chain ...................................... 36
4.6 madwifi SampleRate packet size separation ............... 38
4.7 Minstrel Retry Chain during normal operation ........... 40
4.8 Minstrel Retry Chain during lookaround operation ...... 40
4.9 Fingerprinting Summary: Madwifi r4180 ............... 44

5.1 Radiotap defined-fields ........................................ 47
5.2 Comparison between the RAC capture engine and Wire-
shark using the Buffalo WLI-UC-G300N USB Wireless
Adapter ............................................................... 53
5.3 Rate match between the RAC capture engine and Wire-
shark using the Atheros AR9285 Wireless Network Adapter 53
5.4 Packet capture sequentiality .................................. 54

6.1 Testbed Technical Specifications ................................. 63
6.2 Testbed Wireless Hardware ....................................... 64

8.1 AMRR retry chain with best rate of 18Mbps .......... 73
8.2 AMRR retry chain with best rate of 5.5Mbps .......... 73
8.3 AMRR Validation Statistics .................................. 75
8.4 Onoe validation test statistics ........................... 77
8.5 SampleRate Retry Chain as implemented in madwifi .. 80
8.6 SampleRate validation test statistics ................. 82
8.7 Rate Statistics, experiment 1, SampleRate (rates80211bg.dat
aggregated, 3 runs) ............................................. 84
8.8 Minstrel validation test statistics ....................... 86
8.9 RAA Comparison Table (Madwifi) ......................... 89
Part I

Introduction
Wireless Local Area Networks (WLAN) first entered the market in the late 1990s after the IEEE 802.11 Working Group was established to work on the first standard. They finished their work and published the standard in 1997. Since the original standard was published, there have been several extended versions of the same base technology. The legacy 802.11 standard was designed using two bit-rates at the PHY layer, 1Mbps and 2Mbps. This was the basic design principle of the first Rate Adaptation Algorithm Auto Rate Fallback (ARF). This algorithm was designed to select the best performing rate out of the two under a changing wireless environment.

Shortly after the first IEEE 802.11 standard was published, two additions were released, 802.11a [8] and 802.11b [9]. These standards used more physical modulation schemes and improvements to the standard in order to provide higher data rates. The 802.11a standard was designed to work in the 5GHz band providing data rates up to 54Mbps while the IEEE 802.11b standard were designed to work in the 2.4GHz band, extending the legacy IEEE 802.11 rates, adding 5.5Mbps and 11Mbps.

In 2009, the IEEE 802.11n amendment [11] was published. This addition enables rates up to 600Mbps using several spatial streams and includes other improvements to the physical transmission protocol.

In this thesis, a tool for network professionals trying to get a better understanding of their wireless network devices is presented. The tool is aimed towards Rate Adaptation Algorithm (RAA) in 802.11 wireless Networks. The tool is called Rate Adaptation Classifier (RAC); it is able to listen and capture wireless network traffic and help the user to get a better understanding of the RAA used in a specific device. To remain flexible in the face of constantly changing device drivers and algorithm updates, this tool is not intended to give the user a definite answer of which RAA any particular device is using, but to provide useful data to the user which must be interpreted and analysed in order to determine which RAA is used in any specific case. A method is presented to systematically derive the RAA.

RAC passively listens, captures and analyses traffic of any IEEE 802.11 capable wireless device. The tool requires the host computer to have an IEEE 802.11 capable wireless network adapter which is capable of running in monitor mode [22, 29]. The 802.11 standard defines a set of physical rates for transmitting frames between wireless stations, but it does not, however, define how the wireless hardware selects one of these rates when transmitting a frame.

Selecting the correct rate is vital in order to maximize the performance of the wireless device [5, 26, 12]. The performance can be defined to be the raw throughput, power saving, low loss or via other performance metrics. As the standard does not define the rate selection mechanism, several have been proposed and implemented, the first one being ARF [16] which was designed for WaveLAN-II devices. ARF is called a Rate Adaptation Algorithm (RAA) or Rate Selection Algorithms (RSA), but this thesis will use the term Rate Adaptation Algorithms.
(RAA) exclusively.

The IEEE 802.11a/b/g standard defines a set of available rates up to a physical rate of 54Mbps in 802.11a and 802.11g. An improvement to ARF was presented in [18]; Adaptive Auto Rate Fallback (AARF), which improves both long-term and short-term adaptation of the algorithm. Several other Rate Adaptation Algorithms have been published, e.g. Adaptive MultiRate Retry (AMRR), Onoe, SampleRate [23], Minstrel and CARA [17].

**Problem Statement**

RAAs implemented in drivers accompanying IEEE 802.11 wireless devices are vendor-specific and are highly likely to be proprietary. Information regarding RAAs may or may not be released to the public. Network administrators and researchers want to know which algorithm is implemented in these drivers to better understand the observed traffic and performance of a certain device under different scenarios.
Part II

Background and literature review
Chapter 1

Related work

Little work has been done trying to fingerprint, or to classify Rate Adaptation Algorithms used in IEEE 802.11 wireless devices. The work done by [21] tries to determine the RAA by using a relatively new machine learning technique called Support Vector Machine (SVM). The idea of this work is that by collecting wireless traffic over a sufficient period of time and then use the collected information as an input to the SVM, one can generate a set of rules which can be used to recognize RAAs by listening to station communications. The method, which is described by [21] to be relatively simple, shows a classification accuracy of 95% to 100%.

In this thesis, a part of the presented tool is designed to efficiently capture wireless network traffic. In [20], the authors present a comprehensive study of the MAC level behaviour of wireless networks. Their approach to measurements of 802.11 wireless networks was to use a number of wireless monitors which captured all relevant parts of wireless data. They developed a tool call Wit, which had three parts. halfWit, the merging component, was used to merge captured traffic from all monitor devices into one capture stream. nitWit takes output from halfWit and creates annotated copies of the captured and interfered packets, and dimWit, the analysis component, takes the output from nitWit and uses it for different analysis purposes. While the work of [20] did not consider RAAs in their analysis, they had an interesting approach to the wireless capture process.
Chapter 2

IEEE 802.11 standard

Local wireless computer networks today usually implement the IEEE 802.11 protocol. The IEEE 802.11 wireless network standard was developed in order to provide wireless local area networks over the non-licensed ISM band [15].

802.11 defines and specifies the protocol over two layers, the Physical Layer (PHY) and the Link-Layer. Section 2.1 will introduce the important and relevant parts of the 802.11 PHY while Section 2.2 will introduce the Link-Layer.

2.1 Physical layer

The Physical layer of the 802.11 protocol stack is associated with the first layer of the OSI model with the same name. 802.11 uses several modulation and coding schemes at the physical layer namely, DSSS, FHSS, OFDM and HR-DSSS.

In Section 2.1.1, we will describe the different modulation techniques used. This section also introduces the main amendments for IEEE 802.11 including IEEE 802.11n. In Section 2.1.3 we will introduce the different 802.11 preamble mechanics (long and short).

2.1.1 Modulation Techniques

The IEEE 802.11 standard implements three main modulation techniques, namely Direct Sequence Spread Spectrum (DSSS), Frequency Hopping Spread Spectrum (FHSS) and Infrared (IR).

Direct Sequence Spread Spectrum (DSSS)

DSSS is one of the modulation technique used in the 2.4GHz frequency band and is the primary technique used in 802.11b. The DSSS modulation technique is somewhat tolerant to noise, and even if the signal is distorted during transfer, the original sequence can still be extracted from the transmission due to its redundancy in the carrying signal.
The use of Direct Sequence Spread Spectrum (DSSS) in mobile networks, e.g., GPRS severely outperforms the direct radio communication used in GSM, where the stations are time divided and each station gets a time-window for when they can communicate with the base-station. DSSS (as used in GPRS) has a built-in multi-signal simultaneous transmission capability as data from two sources can be sent at the same time. This is made possible by the modulation of the signal and the fact that different stations use their own pseudo-random numbers (chipping code) which is known by only the station and the base-station. This also enables DSSS’s built-in privacy mechanism where the station and the base-station are the only ones with knowledge about the random number used. This gives DSSS a rudimentary encryption scheme since other stations would not be able to decode the DSSS modulated signal without knowing the chipping code. However, this method is not used in the 802.11b DSSS modulation, as DSSS was chosen because of its other properties.

In 802.11 radio networks, all stations are required to be able to receive and understand broadcast transmissions. If DSSS was used with one chipping code for each station, the Point Controller (PC) would have to encode broadcast messages with every station’s key for them to be able to pick up on broadcast messages. This would have created a whole lot of unnecessary traffic over an already saturated wireless medium. Rather, DSSS was chosen because of its natural resistance towards noise.

**Frequency Hopping Spread Spectrum (FHSS)**

FHSS is another modulation technique used in 802.11 wireless networks. This technique uses the notion of hopping from one channel to another at evenly spaced frequencies and time slots. Whenever the transmission has occupied a certain frequency for a set period of time, the communication is moved to another frequency for the next time window. The PC uses broadcast management messages to announce which channels it uses for its hopping scheme, and this is decided by the PC in a pseudo-random fashion. A station who wishes to join the PC usually listens for these broadcast messages and when one arrives, it can start hopping between channels with the PC in order to communicate with other members of the same Basic Service Set (BSS).

The advantage of using the Frequency Hopping Spread Spectrum (FHSS) technique is mainly the ability to circumvent noise appearing on certain frequencies. If the FHSS scheme hops to a channel which is heavily affected by noise, it will only affect the communication between the stations during the time spent in the current frequency. Once the PC hops to another channel, the noise from the previous channel will most likely be negligible, and the communication can resume its normal operation utilizing higher transmission rates.

The downside of using this technique is the need for accurate timing on all stations. To be able to follow all communications in the channel,
every station needs to switch to the new frequency whenever the other stations and the PC do. If the internal clock of a certain station is running slower or faster than the PCs clock, the station may switch to the new frequency at an earlier time than the rest of the members. This could result in data loss and retransmissions which could lower the throughput of the entire channel.

2.1.2 Extended modulation techniques

Shortly after the original 802.11 standard was published, the IEEE 802 Executive Committee approved two extensions to the original 802.11 standard. These were named 802.11a and 802.11b and provided mechanisms and methods to achieve higher physical rates in the wireless medium.

802.11a - OFDM in the 5GHz Band

IEEE 802.11a [8] was the first approved extension to the 802.11 standard. It defines improvements and an additional modulation technique for transmitting data between stations. This extension adopts the Orthogonal Frequency Division Multiplexing (OFDM) scheme to the physical layer, which supports rates up to 54Mbps operating in the Unlicensed National Information Infrastructure [28] 5.0 GHz band.

The OFDM modulation scheme in 802.11a is based on the principle of sub-carriers which are orthogonal to a base sub-carrier. Each sub-carrier is modulated from a high-speed binary signal divided into several lower speed signals, in conjunction with one of the channels in the same band. The U-NII 5GHz band is in some countries custom to local laws and regulations, which may limit the allowed transmission power and some channels may be excluded.

802.11b - High Rate DSSS in the 2.4GHz band

IEEE 802.11b [9] was the second approved extension to the 802.11 standard. The physical layer extension is commonly referred to as the High Rate Direct Sequence Spread Spectrum (HR/DSSS), and it extends the 802.11 legacy data rates of 1Mbps and 2Mbps with data rates of 5.5Mbps and 11Mbps. This is made possible by the extension defining two new Physical Layer Convergence Protocol (PLCP) preambles; short– and long preamble. The long preamble uses the same PLCP preamble and header as the legacy 802.11 DSSS Physical Layer (PHY). It operates in the 1Mbps and 2 Mbps rates and is backwards compatible with 802.11 wireless networks using 1Mbps and 2Mbps rates. The short– and long preamble are further explained in Section 2.1.3.

802.11g - Higher Rate Extensions in the 2.4GHz Band

IEEE 802.11g [10] extends the physical layer of 802.11 wireless local area networks with rates up to 54Mbps using the same frequency
band as 802.11b. This extension is backwards compatible with the 802.11b extension and the two are commonly used together when deploying 802.11 wireless networks. Although 802.11g is backwards compatible with the previously approved 802.11b extension, it may reduce the overall throughput of the network to deploy combined 802.11b/g networks. This is due to the legacy overhead of the backwards compatibility for the 802.11b PHY layer.

The physical modulation scheme used in 802.11g networks is the same OFDM scheme as used in 802.11a. Data rates supported in 802.11g are 6, 9, 12, 18, 24, 36, 48 and 54Mbps. The 802.11g standard falls back to CCK (used in 802.11b) for the 5.5 and 11Mbps rates, and DBPSK/DQPSK+DSSS as used in the legacy 802.11 standard for 1 and 2Mbps.

802.11n - Enhancements for Higher Throughput

IEEE 802.11n [11] is the fifth amendment for the 802.11 standard. This amendment provides, among several other things, the ability to use wider channels, frame aggregation and delayed acknowledgements. The physical layer of IEEE 802.11n operates in three modes:

Non-HT (Legacy) Mode is for compatibility with legacy devices which do not support the new MAC layer format. The AP operates in the old 802.11a/b/g format, thus all new features are disabled. This mode can only use 20MHz channel-width.

HT Mixed Mode is a mode for mixing legacy IEEE 802.11a/b/g with the new enhanced modes of IEEE 802.11n. This mode permits stations which only support legacy communication to communicate with the AP, but opens up the enhanced modes to stations able to communicate over the new 802.11n frame format, in addition to also having support for older devices not capable of the enhanced operating modes.

High Throughput (Greenfield) Mode is used with APs that want to transmit exclusively over the new IEEE 802.11n MAC layer frame format. This format is called *Greenfield* and supports all the new features of IEEE 802.11n. Stations which only support IEEE 802.11a/b/g cannot communicate with the AP in this mode.

### 2.1.3 Long and short preamble

There are two defined preambles in the 802.11 physical layer, the Long Preamble (802.11 legacy) and the Short Preamble. The preamble is used by any station when it wants to send data onto the wireless medium. The format of the preamble and its position in the packet is defined by the PLCP, and the PLCP Protocol Data Unit (PPDU) is the data unit during PLCP transmission. The Physical layer Service Data Unit
(PSDU) must have a PLCP Preamble and Header inserted before it to create a PPDU.

The Long Preamble and header is used in the original 1 and 2Mbps DSSS IEEE 802.11 standard. The Long Preamble consists of a 128bit sync field in the beginning of the frame, while the Short Preamble uses a 56bit sync field. The Short Preamble was introduced in order to maximize the throughput of a wireless link to support live video streaming, voice communications and other high demand applications and protocols. All devices in a BSS running in the original 802.11 2.4GHz frequency band are required to be able to transmit and receive Long Preamble frames. All devices in an 802.11g BSS are required to be able to both receive and transmit both Long and Short Preamble frames, while support for Short Preamble frames in 802.11b are optional.

## 2.2 Media Access Control

The Media Access Control (MAC) layer used by the 802.11 family protocols is the foundation of the transmission protocol. It uses the Carrier Sense Multiple Access/Collision Avoidance (CSMA/CA) technique to organize transmissions between nodes operating on the same BSS. In 802.11, a single Access Point (AP) connected to one or more stations is called a BSS. Unlike the Ethernet standard, which uses Carrier Sense Multiple Access/Collision Detection (CSMA/CD), the 802.11 protocol family aims to avoid collisions rather than to detecting them after the transmission is completed. The MAC layer is responsible for the frame exchange protocol which consists of a two-frame sequence, a forward data-frame and a return ack-frame. If the source of the transmission does not receive an ACK frame within a predetermined amount of time, the frame is considered lost and retransmitted based on instructions given by the wireless hardware or the driver. The MAC layer is responsible for

- Reliable Data Delivery.
- Media Access.
- Security.

### 2.2.1 RTS/CTS

The Request To Send (RTS)/Clear To Send (CTS) protocol was introduced in an attempt to better the hidden node problem. A hidden node is a node only visible to a subset of the nodes in the same BSS. The mechanism is based on an initial communication between the sender and the receiver where the sender sends an RTS frame. The receiver replies to this RTS frame with a CTS frame. The receiver contains information about the upcoming transmission for all nodes in the BSS to receive. The RTS/CTS mechanism introduces communication overhead to the wireless channel [7] and the authors of
show that the RTS/CTS mechanism produces little advantages to a IEEE 802.11 network when no hidden nodes are present.

Wireless drivers may use a light version of the RTS/CTS mechanism. Instead of transmitting the RTS frame and waiting for the receiver to reply with a CTS, the station may transmit the CTS frame itself. This will not work as intended with the hidden node problem, but it will tell stations nearby that the wireless channel will be occupied for a set amount of time.

2.2.2 Frames Types

The IEEE 802.11 standard defines three main types of frames

Data: Frames used for data transmission

Control: Frames used to control access to the medium (e.g. RTS, CTS, and ACK).

Management: Frames transmitted like data frames for exchanging management information, but not forwarded to the upper layers.

All of these frame types are further divided into subtypes in order to fulfil specific tasks. All 802.11 frames are composed of the components seen in Figure 2.1. The components are described below.

Preamble: An 80-bit alternating ones and zeros and a 16-bit start pattern of 0000110010111101.

PLCP Header: A collection of information used by the PHY layer to decode the frame.

MAC Data: The actual MAC data.

CRC: A 32-bit field containing a Cyclic Redundancy Check (CRC).

The MAC Data frame consist of multiple fields seen in Figure 2.2. There are four address fields in the MAC Data frame. The addresses contained in these are dependant on the ToDS and FromDS bits in the Frame Control Field. Table 2.1 summarizes. To extract the Destination Address (DA), Sender Address (SA), Basic Service Set ID (BSSID), Receiver Address (RA) or the Transmitter Address (TA), one need to interpret the address fields differently based on the value of the ToDS and FromDS fields.

<table>
<thead>
<tr>
<th>Preamble</th>
<th>PLCP Header</th>
<th>MAC Data</th>
<th>CRC</th>
</tr>
</thead>
</table>

**Figure 2.1:** 802.11 Frame Components
Figure 2.2: General MAC Frame Format

<table>
<thead>
<tr>
<th>To DS</th>
<th>From DS</th>
<th>Address 1</th>
<th>Address 2</th>
<th>Address 3</th>
<th>Address 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>DA</td>
<td>SA</td>
<td>BSSID</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>DA</td>
<td>BSSID</td>
<td>SA</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>BSSID</td>
<td>SA</td>
<td>DA</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RA</td>
<td>TA</td>
<td>DA</td>
<td>SA</td>
</tr>
</tbody>
</table>

Table 2.1: MAC Frame Address Field Contents

Frame aggregation in IEEE 802.11n

802.11n can in addition combine several frames into one larger frame and send this larger frame during one single access to the medium. Frame aggregation is done in one of two ways, Aggregated Mac Service Data Unit (A-MSDU) or Aggregated Mac Protocol Data Unit (A-MPDU). A-MSDU increases the maximum transmission size from the normal 802.11a/b/g size of 2304 bytes to 7935 bytes, while A-MPDU can aggregate frames up to a maximum of 64kbytes.

A-MSDU is based on the technique of aggregating several frames with the same source– and destination endpoints, i.e. the MAC addresses, into one large frame with a common MAC header, see Figure 2.3. Frames can also be aggregated with additional QoS traffic class parameters.

While potentially providing a better throughput, sending large frames is less resilient against bit errors during transfer. Aggregating frames could result in a lower throughput if the transmission fails repeatedly because of an unstable wireless environment.

Figure 2.3: A-MSDU
A-MPDUs aggregation of frames is slightly different from the A-MSDU. While A-MSDU aggregates frames and gathers them all under a single MAC header with a single CRC for the entire frame, A-MPDU basically gathers frames with the same physical destination address and QoS class and sends them back-to-back within one access to the wireless medium, see Figure 2.4. This means that all the individual frames sent back-to-back will still have their own header and CRC, and thus, an error in one of the frames will not affect the others in the aggregated stream.

2.3 Bit-rates in 802.11a/b/g

The IEEE 802.11 standard defines a set of physical bit-rates which can be used when transmitting data over the wireless channel. The available bit-rates depend on the version of the IEEE 802.11 wireless network being implemented. Table 2.2 show the rates available for wireless networks implementing IEEE 802.11a/b/g.

2.4 Bit-rates in 802.11n

The IEEE 802.11n amendment is very different from IEEE 802.11a/b/g. The rates are different and deciphering rates is not as straightforward as in a pure 802.11a/b/g network. Rates used in IEEE 802.11n are dependant on the Guard Interval (GI), the bandwidth and the MCS index.

As the High Throughput enhancement in 802.11n is better for application-level throughput, capturing frames sent over 802.11n is a little different. For 802.11a/b/g, the rates are easily extracted from either the Radiotap or the Prism header, dependant on the driver the wireless card uses. For radiotap [1], one can extract the rate as a raw value (in 500Kbps units), see Table 2.3, the same goes for PRISM and AVS headers.

The field used for extracting the PHY rate is not applicable when the monitor device captures a High Throughput (HT) frame. A HT frame is a frame transmitted using the 802.11n enhancements. To get the rates of 802.11 HT frames, the radiotap header must be parsed for the MCS field. Table 2.4 defines the MCS fields for the Radiotap header.
<table>
<thead>
<tr>
<th>Bit-rate (Mbps)</th>
<th>802.11</th>
<th>802.11a</th>
<th>802.11b</th>
<th>802.11g</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

**Table 2.2:** Bit-rates in IEEE 802.11

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure</td>
<td>u8</td>
</tr>
<tr>
<td>Unit</td>
<td>500Kbps</td>
</tr>
</tbody>
</table>

**Table 2.3:** Radiotap defined-fields/Rate

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure</td>
<td>u8 known, u8 flags, u8 mcs</td>
</tr>
</tbody>
</table>

**Table 2.4:** Radiotap defined-fields/MCS
can see that the MCS field contains three bytes (u8) of data, whereas the first byte is a known bitfield which defines which of the flags are known. If the 0x01 bit in the known byte (bandwidth) is set, we know that the bandwidth values in the flags field are to be trusted. See Table 2.5 and Table 2.6 for the Known and Flags fields respectively.

2.4.1 MCS index

The MCS index table shows the connection between the number of spatial streams, the modulation and the coding rate against different characteristics of the transmitted frame. The rate used is one of these characteristics and the rates are as shown in Table 2.7. This table contains some of the sections found in the complete table from http://mcsindex.com. By using the information from the MCS field, we can decide the rate for a frame by cross-referencing this field with the value of the row in the table. Each of the rows in Table 2.7 contains four rate columns. This means that getting the MCS index alone is not enough to be able to set the rate for the frame. To get the exact rate, we also need to extract the Bandwidth and Guard Interval value from the MCS fields in addition to the MCS index.

Capture example

Say we receive a frame where the MCS index is 26, the GI is 800ns and the bandwidth is 40MHz. We can now find from Table 2.7 that the PHY rate used for the frame is 162Mbps.

2.4.2 Rate Selection

Selecting the correct rate when transmitting a frame is important. If the driver selects a rate which is performing bad, the throughput of the transmission is severely affected. The next chapter will focus on rate adaptation and how this is implemented and used in IEEE 802.11 wireless network devices.

<table>
<thead>
<tr>
<th>flag</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>Bandwidth</td>
</tr>
<tr>
<td>0x02</td>
<td>MCS index known (in MCS part of the field)</td>
</tr>
<tr>
<td>0x04</td>
<td>Guard interval</td>
</tr>
<tr>
<td>0x08</td>
<td>HT format</td>
</tr>
<tr>
<td>0x10</td>
<td>FEC type</td>
</tr>
<tr>
<td>0xe0</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Table 2.5: Radiotap defined-fields/MCS/known

18
<table>
<thead>
<tr>
<th>flag</th>
<th>definition</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x03</td>
<td>bandwidth</td>
<td>0: 20, 1: 40, 2: 20L, 3: 20U</td>
</tr>
<tr>
<td>0x04</td>
<td>guard interval</td>
<td>0: long GI, 1: short GI</td>
</tr>
<tr>
<td>0x08</td>
<td>HT format</td>
<td>0: mixed, 1: greenfield</td>
</tr>
<tr>
<td>0x10</td>
<td>FEC type</td>
<td>0: BCC, 1: LDPC</td>
</tr>
<tr>
<td>0xe0</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

**Table 2.6: Radiotap defined-fields/MCS/flags**

<table>
<thead>
<tr>
<th>MCS index</th>
<th>Spatial Streams</th>
<th>Modulation</th>
<th>Coding Rate</th>
<th>Data Rate (in Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>GI = 800ns</td>
<td>GI = 400ns</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20MHz</td>
<td>40MHz</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>BPSK</td>
<td>1/2</td>
<td>6.5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>QPSK</td>
<td>1/2</td>
<td>13.0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>QPSK</td>
<td>3/4</td>
<td>19.5</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>16-QAM</td>
<td>1/2</td>
<td>26.0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>16-QAM</td>
<td>3/4</td>
<td>39.0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>64-QAM</td>
<td>2/3</td>
<td>52.0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>64-QAM</td>
<td>3/4</td>
<td>58.5</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>64-QAM</td>
<td>5/6</td>
<td>65.0</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>BPSK</td>
<td>1/2</td>
<td>13.0</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>QPSK</td>
<td>1/2</td>
<td>26.0</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>QPSK</td>
<td>3/4</td>
<td>39.0</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>16-QAM</td>
<td>1/2</td>
<td>52.0</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>16-QAM</td>
<td>3/4</td>
<td>78.0</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>64-QAM</td>
<td>2/3</td>
<td>104.0</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>64-QAM</td>
<td>3/4</td>
<td>117.0</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>64-QAM</td>
<td>5/6</td>
<td>130.0</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>BPSK</td>
<td>1/2</td>
<td>19.5</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>QPSK</td>
<td>1/2</td>
<td>39.0</td>
</tr>
<tr>
<td>18</td>
<td>3</td>
<td>QPSK</td>
<td>3/4</td>
<td>58.5</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td>16-QAM</td>
<td>1/2</td>
<td>78.0</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>16-QAM</td>
<td>3/4</td>
<td>117.0</td>
</tr>
<tr>
<td>21</td>
<td>3</td>
<td>64-QAM</td>
<td>2/3</td>
<td>156.0</td>
</tr>
<tr>
<td>22</td>
<td>3</td>
<td>64-QAM</td>
<td>3/4</td>
<td>175.5</td>
</tr>
<tr>
<td>23</td>
<td>3</td>
<td>64-QAM</td>
<td>5/6</td>
<td>195.0</td>
</tr>
<tr>
<td>24</td>
<td>4</td>
<td>BPSK</td>
<td>1/2</td>
<td>26.0</td>
</tr>
<tr>
<td>25</td>
<td>4</td>
<td>QPSK</td>
<td>1/2</td>
<td>52.0</td>
</tr>
<tr>
<td>26</td>
<td>4</td>
<td>QPSK</td>
<td>3/4</td>
<td>78.0</td>
</tr>
<tr>
<td>27</td>
<td>4</td>
<td>16-QAM</td>
<td>1/2</td>
<td>104.0</td>
</tr>
<tr>
<td>28</td>
<td>4</td>
<td>16-QAM</td>
<td>3/4</td>
<td>156.0</td>
</tr>
<tr>
<td>29</td>
<td>4</td>
<td>64-QAM</td>
<td>2/3</td>
<td>208.0</td>
</tr>
<tr>
<td>30</td>
<td>4</td>
<td>64-QAM</td>
<td>3/4</td>
<td>234.0</td>
</tr>
<tr>
<td>31</td>
<td>4</td>
<td>64-QAM</td>
<td>5/6</td>
<td>260.0</td>
</tr>
</tbody>
</table>

**Table 2.7: Radiotap MCS index table**
Chapter 3

Rate Adaptation

Rate Adaptation Algorithm (RAA), Rate Control Algorithm (RCA) or Rate Selection Algorithm (RSA) (hereafter RAA), are algorithms and control mechanisms located in either the driver of a 802.11 wireless network card, or implemented partly or fully in its hardware. The purpose of the RAA is to try to optimize the data throughput, or in some cases optimize the power consumption or other metrics of a wireless device. The RAA consists of a set of rules on how to select the preferred bit-rate for outgoing transmissions in a wireless network taking the condition of the channel into account. The condition of a wireless channel can be determined by the level of the Signal-To-Noise Ratio (SNR), when classifying the condition based on the physical characteristics of the channel, or it can be determined using Link-Layer information such as packet-loss and retransmission of outgoing or incoming frames. Rate Adaptation Algorithms can use the physical layer, the link layer, or even both to determine the state of the channel before scheduling an outgoing packet. This section will describe the different Rate Adaptation Algorithms used in this thesis, and the difference between them to be able to differentiate one from another. We will start by describing algorithms using the physical layer in Section 2.1, then describe link-layer algorithms in Section 2.2. Thereafter we will describe algorithms using both the physical layer and the link-layer to determine the state of the wireless channel.

3.1 PHY-based algorithms

PHY-based rate adaptation algorithms, are algorithms using information about the physical condition of a wireless channel to determine the best suited transmission rate for an outgoing frame. The hardware of an 802.11 enabled wireless network device usually exports the current signal-to-noise ratio of the associated wireless channel. This can be used in wireless drivers to determine the state of the channel by comparing the SNR to a table of pre-calculated values between the SNR and transmission rates used by the channel.
3.1.1 Receiver-Based AutoRate (RBAR)

Receiver-Based AutoRate (RBAR) is a receiver based algorithm which has the goal of optimizing the application level throughput. While this algorithm is an important tool for theoretical comparison between other RAAs, it cannot be implemented in current 802.11 networks because it requires incompatible changes to the protocol. RBAR interprets some MAC control frames differently, and each data frame must include a new header field.

RBAR is an algorithm which works in the receiver end of two data-exchanging parties over the 802.11 Wireless Local Area Network (WLAN) protocol. It uses RTS/CTS and the sender states at which data rate it wants to send the data packet. The only thing the receiver are allowed to suggest to the sender in the returning CTS packet is an alternate power level of the transmission. The data rate of the sender can be adjusted by Acknowledgement (ACK) or Negative Acknowledgement (NACK) received from the receiver at the end of a data transmission.

To summarize, the protocol works like this:

1. The sender sends a RTS packet to the receiver
2. The receiver, if available, responds with a CTS packet, which may or may not contain a suggestion to alter the transmit power of the data transmission.
   - If the CTS packet contains a message about changing the transmit power, the sender may adjust the power accordingly
3. The sender starts sending the packet once it has received and interpreted the CTS packet,
4. The receiver can adjust the rate of the next data transmission by sending ACK or NACK after the data transmission is completed.
   - If changes to the data rate are suggested by the receiver, these changes are then used for the subsequent packet transmission.

The RBAR algorithm suffers from several issues [18], the main issue being that it is incompatible with the IEEE 802.11 standard, and as such, cannot be deployed in any existing 802.11 wireless environments.

3.2 Link-Layer-Based algorithms

The link-layer in 802.11 WLAN protocol is used by some RAAs to estimate the quality of the channel. In the Link-Layer, one may collect information from data– or signalling frames, or both.

When using the data frame, there are two methods of finding the best rate estimation for a channel. One may probe the channel, by
occasionally sending data frames at a rate different than the current rate, or use the method of non-probing which never sends out probe frames.

### 3.2.1 Automatic Rate Fallback (ARF)

Automatic Rate Fallback [16], also called ARF, was the first RAA to be published. The algorithm is a transmitter based rate adaptation algorithm whose goal it is to increase the application level throughput of the wireless transmission. The idea behind ARF is that each sender tries to use a higher transmission rate after a set number of successful transmissions at the current rate. If it experiences one or two consecutive losses, it falls back to the next lower rate. When ARF increases the sending rate, the subsequent transmission decides if ARF will continue to use the higher rate or fall back to the previous lower one. The first packet is often referred to as a probing packet.

The algorithm works as follows:

1. If ACKs for two consecutive data packets are not received by the sender
   - Then, the sender drops the transmission rate to the next lower rate and starts a timer
2. If ten (10) consecutive ACKs are received
   - Then, the transmission rate is raised to the next higher data rate and the timer is cancelled
3. If the timer expires, the transmission rate is raised as before, but with a condition that if an ACK is not received with the first transmission (the probing transmission), the data rate is lowered again and the timer is restarted.

There are two problems with the ARF algorithm as described in [18], and quickly summarized. The algorithm has problems when the conditions of the channel changes very quickly, as it cannot adapt rapidly enough to the changing environment, and the algorithm always tries to increase the transmission rate even when the channel conditions are very stable, which in terms lowers the application throughput. In defence of the algorithm, it was mainly designed to work in a 2-rate environment.

### 3.2.2 Adaptive ARF (AARF)

AARF [18] is an algorithm based on ARF with the goal of performing better in stable channel conditions. ARF tries to increase the transmission rate after 10 consecutive successful transmissions. This will lower application-level throughput in stable channel conditions.
where the next higher rate always fails. AARF tries to optimize this by using history of the channel, and increase the number of consecutive successful transmissions if the channel is stable and works best with a fixed transmissions rate.

AARF behaves more or less like ARF, but unlike ARF it increases the number of consecutive successful transmissions it needs before it tries to send a transmission at a higher rate than the current one. It does this by remembering the number of failed attempts to probe the channel at a higher rate, and each time the probe transmission fails, the algorithm multiplies the number of consecutive successful transmissions by two, up to a maximum of 50. If a packet fails twice while in the current transmission, it lowers the transmission rate one step and resets the consecutive successful transmission counter to 10 just like ARF.

3.2.3 Adaptive Multi-Rate Retry (AMRR)

AMRR is an algorithm based on ARF with an additional Binary Exponential Backoff (BEB) much like AARF. The Atheros AR 5212 chipset exports a mechanism to drivers called Multi-Rate Retry (MRR). The MRR is a descriptor table sent to the hardware along with the data to be transmitted. The descriptor table is populated with four rates used when transmitting the frame. Each rate is accompanied with a count value stating the number of times each rate is attempted sent. To get the BEB in AMRR, the values of the count fields in the rate/transmission count are set to one, $c_0=1$, $c_1=1$, $c_2=1$ and $c_3=1$. For more information on the MRR, see Chapter 4. The transmission rates are chosen based on the current transmission rate and the minimum rate of the wireless medium used. $R_3$ is always set to the minimum transmission rate, while $R_1$ and $R_2$ are set to the two rates just below $R_0$. $R_0$ is determined by the previous value of $R_0$ and the transmission results for the elapsed period.

3.2.4 Onoe

Onoe is a credit based algorithm where the credit is determined by the frequency of successful deliveries, erroneous deliveries and retransmissions accumulated during a fixed invocation period of 1000ms (1sec). As the credit is determined using a relative long period of time, it is less sensitive to individual packet loss than ARF and AMRR.

Onoe keeps track of the number of successfully transmitted frames and the number of error frames. Onoe increases its credit when less than 10% of packets require retransmission at a particular rate. The credit is increased until it reaches a value of 10, at which point the algorithm starts sending transmissions at the next available higher rate, and the credit is reset to 0. In the case where retransmissions occurred for more than 10% of the packets sent in the last period, Onoe reduces the rate to the next lower rate and the process is restarted with
a credit value of 0. If a rate has been abandoned because of transmission failures, Onoe marks that rate as a failure rate, and will not attempt to use it again until 10 seconds have elapsed.

3.2.5 SampleRate

SampleRate is an algorithm presented in “Bit-rate Selection in Wireless Networks” by John C. Bicket [23]. This algorithm aims to provide the highest possible application throughput by using statistics and sampling over rates which could provide a better throughput than the current rate. The algorithm keeps track of previously transmitted frames in a table for each rate for each station. Each rate in the table contains the number of attempted transmissions, the number of successive failed transmissions, the number of ACK'ed frames, the total transmission time, average transmission time and lossless transmission time. This helps the algorithm select the rate which gives the maximum throughput even if this is not the highest rate available.

The algorithm uses sampling over rates which are not the current best rate to build statistics about other rates in order to determine the best suited rate for a given “health” of the wireless channel. This is done by sending a frame at a selected rate other than the current one at every ten transmissions. The selection chooses its rate based on the estimated frame transmission time for a given rate, which is based on previous transmissions. Rates which have four successive failed transmissions are not sampled, and if no rate has an estimated transmission time lower than the current rate, SampleRate sends the frame using the rate which has the lowest average transmission time. To calculate the estimated transmission time for a selected rate, the algorithm uses transmission results from the last ten seconds.

3.2.6 Minstrel

Minstrel [24] is a bit-rate selection algorithm designed by the team behind the madwifi driver. Minstrel was designed to maximize the throughput of wireless communications implementing the IEEE 802.11 standard. Minstrel has many similarities to SampleRate, but the algorithm has two different primary metrics. While SampleRate selects its best rate based on the transmission time of each frame, Minstrel selects its bit-rate based on which rate can reach maximum throughput taking into account the expected number of retransmissions, based on statistical history of the wireless channel.

Minstrel, like SampleRate, probes the wireless channel in order to get an overview of its “health”. Minstrel defines these probing packets as “Lookaround” packets and spends a set amount of time probing bit-rates other than the current best. The bit-rates selected for the probing frames are selected more intelligently than in SampleRate. Minstrel, for example, does not sample rates that cannot possibly provide a better throughput than the current best.
The rate statistics of the channel are evaluated periodically. Minstrel uses EWMA in order to update historical statistics, and new results (results from the just completed time period) are added to the statistics of every evaluation event.

3.2.7 PID

PID Rate Control [25] is a RAA created for use in the mac80211 framework [19]. The PID Rate Control builds on a general control theoretic feedback technique which consists of the Proportional, Integral and Derivative elements. The basic function of the PID controller is a feedback loop that controls the output based on the input of the previous output state. The algorithm is controlled through a set of variables, each set to tune the output of the PID controller in order to achieve the best possible results.

3.2.8 MiSer

MiSer [27] is an algorithm constructed with the goal of optimizing local power consumption. This algorithm does an offline calculation of the optimal power/rate pairs using a wireless channel model. At runtime, the algorithm uses the offline constructed table to find the optimal transmission rate and power level for each transmission.

The work presented in [18] mentions problems with this algorithm. A quick summary is that the algorithm requires a priori knowledge about the channel and the environment of the wireless network.

3.3 MAC802.11 framework

The mac80211 framework [19] is a framework written to help developers write drivers for SoftMAC wireless devices. SoftMAC wireless devices require a software part in order to function properly. The software is responsible for controlling the MAC Sublayer Management Entity, and the mac80211 framework is there to provide a common API for driver developers.

The mac80211 framework comes with three built-in RAAs, two implementing the Minstrel algorithm, Minstrel and Minstrel HT and the other being a PID Rate Control implementation. These RAAs can be used by drivers using the mac80211 framework in order to not have to implement their own. Minstrel HT is an implementation of Minstrel supporting High Throughput (802.11n) rates.
Part III

Rate Adaptation Classifier (RAC)
Chapter 4
Design

The goal of this thesis is to design and implement an application capable of helping to classify Rate Adaptation Algorithms used by 802.11 wireless devices. We will analyse and research different rate adaptation algorithms in open-source drivers to better understand their behaviour and to be able to recognize traffic governed by these algorithms. This will enable us to reliably distinguish one RAA from another.

Up to this point, we have mentioned many different RAAs. We are going to focus on RAAs which are used in current popular open-source WLAN drivers such as madwifi. The ath5k and the ath9k driver use the mac80211 framework, which currently deploys three RAAs, Minstrel, Minstrel HT and PID. The madwifi driver has four, AMRR, Onoe, SampleRate and Minstrel.

We chose the approach of manually analysing and classifying RAAs instead of fingerprinting algorithms based on a machine-learning technique, as used by the authors of [21]. This because we believe the SVM technique does not deliver enough details when trying to classify RAAs. We want information about traffic patterns, sampling- and rate change frequency, RTS/CTS usage, retry rates and other metrics making us able to reliably differentiate between algorithms. This approach, while being very precise, requires the user to have deep knowledge of different RAAs. We will provide the user with a method of analysing the output of the tool and draw the correct conclusion based on this method.

This chapter contains the initial research and characteristics of the previously mentioned RAAs. The RAAs are explained in detail and the focus will be on locating characteristics of the RAA that make it stand out. This will be used by the method in order to map traffic patterns and behaviour to algorithms.

4.1 Fingerprinting Rate Adaptation Algorithms

In order to distinguish RAAs, we will analyse and find specific characteristics for each of the selected RAAs. We will determine how we can use this in order to decide which RAA is producing which traffic
pattern. Following this will be a discussion of what to use and how to use it.

Different RAAs have different characteristics. Some change their rate selection at a set time interval, while others change their rate selection based on the number of transmitted packets. RAAs handle frame retransmission differently and the ones using Multi-Retry Chain (MRR) populate the retry chain differently. RAAs select their best rate based on the rate they decide to be the best performing rate. The metrics used to decide the best performing rate may differ.

IEEE 802.11 WLANs retransmit failed frames. Rates used during retransmission can differ from the rate used in the original frame, but the sequence number is kept the same. There are different ways of handling this, but wireless devices using Atheros chipsets can pass a descriptor to the hardware containing the selected rates. This descriptor is called the MRR. Different RAAs populate the retry chain differently, and capturing retransmissions enables us to analyse this rate selection.

The madwifi retry chain, see Table 4.1, is a set of ordered pairs of rate and retry count used when a transmission is scheduled to be transmitted by a station. The RAAs job is to fill this table with the preferred rates and their retry counts. For example, the AMRR rate adaptation algorithm can populate this table as in Table 4.2. In this arbitrary example, the best current rate selected by AMRR is 36Mbit/s. AMRR fills the table with the best current rate as $r_0$; $r_1$ and $r_2$ are filled with the next immediate lower rates, and $r_3$ is always filled with the lowest rate. The retry counts are all set to 1, which means that when a loss happens, the Hardware Abstraction Layer (HAL) will immediately switch to the next lower rate and try this once. If this also fails, it will switch to the next, $r_2$ and try this once as well. If the three first transmissions fail, it will fall back to the basic rate and try this once as a last attempt.

As we can see from this behaviour, it is possible to recognize wireless traffic where AMRR is selected as the Rate Adaptation Algorithm based on the retry chain. It has to be mentioned that other RAAs use the same or a very similar pattern when populating the retry chain. One cannot decide the RAA by analysing the retransmission behaviour alone.

When deciding the best rate, different RAAs implement different strategies for analysing the wireless environment. There are algorithms which use statistics by analysing previously transmitted frames and

<table>
<thead>
<tr>
<th>Segment</th>
<th>Rate</th>
<th>Retry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$R_0$</td>
<td>$C_0$</td>
</tr>
<tr>
<td>2</td>
<td>$R_1$</td>
<td>$C_1$</td>
</tr>
<tr>
<td>3</td>
<td>$R_2$</td>
<td>$C_2$</td>
</tr>
<tr>
<td>4</td>
<td>$R_3$</td>
<td>$C_3$</td>
</tr>
</tbody>
</table>

Table 4.1: Madwifi Retry Chain
<table>
<thead>
<tr>
<th>Segment</th>
<th>Rate</th>
<th>Retry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>36</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.2: AMRR Retry Chain with best rate of 36Mbps.

select the best rate based on calculations and/or a-priori knowledge of the wireless environment. SampleRate periodically survey the wireless environment by deliberately transmitting frames using rates other than the current best. It does this in order to maintain statistics of the wireless environment. 10% of the transmission time is used to sample other rates, and the sampling rate is selected based on which rate may provide a better throughput than the current one.

4.1.1 ARF and AARF

As the first published RAA, Automatic Rate Fallback (ARF), by Ad Kamerman and Leo Moneban in [16], it is one of the simplest RAAs, and one that is very often referenced and considered the first step into improving and utilizing multi-rate capable wireless devices. AARF is the descendant of ARF, especially since it is very much based on ARF with some improvements. These two RAAs are very simple, and since ARF was first implemented and designed to work in a two-rate environment, it did not consider or take into account the underlying modulation techniques existing in today’s more sophisticated versions of the originally defined 802.11 standard. ARF only considers the results of the $n$ last transmissions, and only cares about statistics for the currently selected rate. ARF behaviour is very simple, and can be summarized as follows:

1. 2Mbit/s is considered the default rate.
2. If two consecutive ACKs are lost
   - Immediately fall back to the next lower rate
   - Start a timer and keep track of successively received ACKs
   - If the timer expires or ten (10) successive ACKs are received, increment the current rate and resume normal operation
3. If one ACK is lost immediately after incrementing the rate, fall back to the next lower rate and continue from 2

It should be mentioned that there is some discrepancy between the originally proposed ARF algorithm in [16] and the one implemented for testing in [13]. The originally proposed algorithm mentions a timer which is started when the algorithm enters fallback mode, but it does not differentiate between exiting fallback-mode as of the timer
expiring or when 10 successive ACKs are received. Either way, ARF exits fallback-mode and tries to send the next packet at the next higher rate. If no ACK is received for this first transmission, the algorithm immediately falls back into fallback-mode and restarts the timer. Reference [13] interprets the timer expiration and the 10 successive ACKs as two distinctly different steps exiting fallback-mode; if 10 successive ACKs are received, the algorithm increases the rate to the next higher rate and resumes normal operation. This requires the loss of two consecutive ACKs in order to fall back into fallback-mode. However, if the timer expires, ARF exits fallback-mode on the condition that the first transmission must be successfully ACK’d, if not it will immediately fall back into fallback-mode.

AARF is very similar to ARF, although with some improvements, it still follows the same rules. The main difference between ARF and AARF is the way AARF handles stable network conditions. While ARF always requires 10 successive ACKs to increase the rate, AARF has the ability to tweak this to some extent. AARF keeps incrementing the number of successful ACKs required to increase the rate when the channel is in poor condition and the probing packet fails. The probing packet is the first packet sent after incrementing the current rate. If this packet fails, AARF doubles the amount of successfully acknowledged packets required in order to attempt another probe packet up to a maximum of 50 packets. This counter is reset to 10 when the algorithm steps down to the next lower rate as a result of two consecutive lost ACKs.

**Fingerprinting ARF/AARF**

When fingerprinting ARF and AARF, a few characteristics are worth keeping an eye on. One very important characteristic is that they can only use the next higher rate after having successfully received 10 consecutive ACKs for ARF, or 10, 20, 40 or 50 for AARF. Another is that neither ARF nor AARF can change the current best rate by more than one step at a time. For example, if the current best rate is 36Mbit/s, then ARF/AARF cannot switch to 12Mbit/s directly, it has to step down one rate at the time. This means that ARF/AARF will have to try 24– and 18– before it can try 12Mbit/s. To summarize, ARF/AARF have characteristics as shown below:

- **Reducing the sendrate**
  - ARF/AARF reduce the current rate if two consecutive packets are lost.
  - They fall back to the next lower rate if probing packet fails.

- **Frequency of best rate changes**
  - Current rate may change upwards as quickly as every 10 packets.
– Current rate may change downwards as quickly as every 2 packets.
– For AARF, the rate of change upwards may decrease as the required consecutive successfully acknowledged packets may double up to a maximum of 50 packets before probing the next higher rate.

• Stepping
– Cannot skip rates while changing best rate.

4.1.2 Adaptive Multi-Rate Retry (AMRR)

The team behind AARF [18] realised that the Binary Exponential Backoff mechanism used in AARF could still be useful when working with the Multi-Rate Retry capability exported by the hardware layer in Atheros based chipsets. The introduction of the BEB into the madwifi driver became AMRR. The madwifi driver was already using the MRR chain. The original madwifi driver periodically changed the MRR population scheme (interval between 0.5 and 1 seconds), but the AMRR implementation changed this in order to improve the algorithm reaction time to short-term changes in the wireless environment.

Fingerprinting AMRR

When observing traffic governed by AMRR, we should easily be able to detect its characteristics. One is its inability to change the current best by rate more than one step at the time. Unlike the more sophisticated RAAs, which can change the current best rate from the highest to the lowest rate in one decision step, AMRR is only able to change the current best rate to the immediate next lower or higher rate. AMRR, like its predecessors ARF and AARF, is fairly simple when it comes to how it decides whether or not to change the current rate. AMRR collects statistics over a period of time to see if it needs to change the current best rate. In the madwifi driver, this time period is set to 1000ms (500ms if operating in station mode), which can be observed when looking at AMRR governed traffic.

Multi-Rate Retry

AMRRs Multi-Rate Retry Chain population is very simple. AMRR keeps track of which rate is the current best rate, and puts this rate in the MRRs R0. As AMRR does not keep any other statistics of rates, it cannot determine if rates other than the immediate lower rates below the current best are better choices for the MRRs R1,2. R3 is always populated with the lowest base rate and all counters in MRR is set to 1. The reason for setting the counters to 1 is to make AMRR more responsive to sudden changes in the wireless environment, although having the rate calculation run as seldom as it does, may defeat this
goal. A representation of AMRRs MRR population is shown in Table 4.3.

This behaviour is very advantageous for our fingerprinting. By observing AMRR governed traffic, we should be able to see the contents of the retry chain as packets are lost and retransmitted, and it should be easy to recognize the pattern of AMRR.

- **Multi-Rate Retry population**
  - Lowest baserate at \( R_3 \).
  - Retry fields are all always 1.
  - \( R_{1,2} \) are always the two rates just below the current.

- **Frequency of best rate changes**
  - Current rate changes only after rate update event (1000ms/500ms for madwifi).
  - Can only change rates one step at a time.

### 4.1.3 Onoe

Onoe is the least known Rate Adaptation Algorithm in the madwifi driver. This algorithm is based on using credits for determining the best rate. Rates collect credits by performing well during a sampling period, which is set to 1000ms in the madwifi driver. Every 1000ms, Onoe evaluates the result from the elapsed period and adjust credits based on a few simple rules as shown below:

- increase by one if less than 10% of the packets in the last interval required a retransmission.
- decrease by one if more than 10% of the packets required a retransmission.

If a rate credit exceeds 10, Onoe change the current best rate to the next higher rate, and it decreases the rate to the next lower rate if the average number of retransmissions per packet exceeds one.

<table>
<thead>
<tr>
<th>Segment</th>
<th>Rate</th>
<th>Retry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Current</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Next Lower</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Next Next Lower</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Lowest Baserate</td>
<td>1</td>
</tr>
</tbody>
</table>

*Table 4.3: AMRR Retry Chain*
**Multi-Rate Retry**

Onoe populates the Multi-Rate Retry chain according to Table 4.4. The MRR population for Onoe is very similar to ARF/AARF and AMRR. $R_0$ is the current best rate and $R_{1,2}$ contains the immediate lower rates. $R_3$ is always the lowest base rate. The count fields for Onoe are populated with $C_0 = 4$ and $C_{1-3} = 2$.

**Fingerprinting Onoe**

When trying to fingerprint Onoe, there are certain characteristics we can look for. Mainly, we look at the MRR and how it is populated. We also know that Onoe has the ability to change the current best rate only every 1000ms.

- Multi-Rate Retry population
  
  - Population as seen in Table 4.4.

- Best rate changes
  
  - Frequency of rate changes.
  
  - Can only change rates one step at a time.

**4.1.4 SampleRate**

SampleRate, first published by J. Bicket in [23] was the first RAA which did not take for granted that lower transmission rates would lead to a lower chance of packet loss.

SampleRate aims to maximize the throughput on wireless links by choosing the bit-rate which it predicts to have the smallest per-packet transmission time based on probing alternative bit-rates than the current rate and estimating the transmission time based on the result of probe packets.

**Fingerprinting SampleRate**

In order to decide whether observed traffic from a wireless station is using the SampleRate RAA, we need to analyse the behaviour of SampleRate, and try to find characteristics which are typical to SampleRate. This section will find and describe typical behaviour

<table>
<thead>
<tr>
<th>Segment</th>
<th>Rate</th>
<th>Retry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>best</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>next lower</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>next next lower</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>lowest possible</td>
<td>2</td>
</tr>
</tbody>
</table>

*Table 4.4: Onoe Retry Chain*
for SampleRate, and create a scheme of how we can determine that observed behaviour has originated from a station using SampleRate as its RAA.

**New associations**

The initial transmission when using SampleRate is set to the highest rate possible. For 802.11.b/g, this is 54Mbit/s. SampleRate will transmit each packet four times before it considers a rate as not working. If we were able to pick up the traffic at the initial stages of a connection between a station and an access-point, we would see that the first packet is transmitted at 54Mbit/s.

**Multi-Rate Retry Chain**

The MRR will be populated by SampleRate according to the calculated estimated transmission time for each rate. SampleRate estimates the expected time it takes to transmit one packet with a certain rate, and the rate which is estimated to have the shortest transmission time is selected as the primary rate. SampleRate then fills MRR $R_1$ with the base rate, and $R_2$ and $R_3$ are not used. SampleRate tries to send the packet three times with the preferred rate, and three times with the basic rate. See also Table 4.5.

By using this knowledge, we can try to determine if observed transmissions originate from a station utilizing SampleRate as its RAA. By observing transmissions which need retries, we should be able to see that a packet is being sent with the same rate three times before dropping down to the base rate. The base rate is then retried three times. If we see this behaviour, we can be fairly certain that SampleRate is responsible for populating the MRR.

**Sampling**

SampleRate, as the name suggests, uses sampling in order to gather information about the wireless environment. SampleRate periodically transmits packets at a different rate than the current best, and by doing this, SampleRate is able to calculate the estimated transmission time of alternative rates. If it finds a rate which could perform better than the current one, it changes to this rate and continues its normal operation. SampleRate, as designed by J. Bickett [23], should use

<table>
<thead>
<tr>
<th>Segment</th>
<th>Rate</th>
<th>Retry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>best</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>base</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

*Table 4.5: SampleRate Retry Chain*
10% of the transmission time for sampling other rates. The rates are chosen intelligently, as SampleRate only samples rates which may perform better than the current best and have not recently seen packets unacknowledged four successive times. A rate is considered unusable if transmissions using the rate fails four times in a row, and is then not considered for transmission or sampling for the next 10 seconds. By selecting the sampling rates intelligently, SampleRate will not sample a rate which has a theoretical transmission time lower than the current rate. For example, if the current rate is 11Mbit/s on a 802.11b network, and the average transmission time of 11Mbit/s is less than the lossless transmission time of 5.5Mbit/s which is 2976µs [23], SampleRate will not select 5.5Mbit/s as a sampling rate. However, as SampleRate continues to transmit packets on 11Mbit/s, the wireless environment may degrade, and the average transmission time of 11Mbit/s may increase above 2976µs. At this point, 5.5Mbit/s is eligible for sampling. SampleRate will now chose 5.5Mbit/s as the sampling rate in order to determine if it performs better than the current 11Mbit/s rate.

As SampleRate periodically samples rates other than the current one (if there exist a better alternative), we could possibly use this to our advantage. By keeping track of the number of packets sent by a station, we could, e.g. observe every 10th packet, and look for changes in the rate selection. If we see that the rate selected is different from the current (the last n packets), we could consider this as a sampling packet and continue to observe traffic with this in mind. If we see that this trend continues, we could conclude that the observed traffic may be controlled by SampleRate.

**Quarantine**

As mentioned, sample rate has the ability to “quarantine” a rate when it suffers from four successive failures. When a rate has been quarantined, it will not be eligible for transmission or sampling during the next ten seconds. This can be used by keeping track of each rate which has been observed, and setting a timestamp when the rate was last used. If we observe a rate failing four successive times, and do not see this rate being used for the next ten seconds, it might be SampleRate which dictates the rate selection for the station in question.

**Size buckets**

The madwifi implementation of SampleRate additionally uses buckets to separate three packet sizes. The driver separates transmissions into packets with a size from 0– up to 250bytes, packets with size from 250– to 1600bytes and the packets from 1600– to 3000bytes, see Table 4.6. This means that SampleRate handles transmissions differently for different size packets. Packets of size smaller than 250bytes may have a different best rate than packets with a size of 1500bytes. This is an advantage for us. By implementing tests which looks at transmissions
of mixed sized packets, we may be able to observe that SampleRate treats different packet sizes differently.

**SampleRate characteristics**

To summarize, we can detect SampleRate by observing the following characteristics:

- **Multi-Rate Retry population**
  - MRR is populated with $C_{0,1} = 3$ and $C_{2,3} = 0$.
  - $R_0$ is the current best rate and $R_1$ is the lowest base rate. The last two are never used.

- **Sampling**
  - SampleRate uses 10% of the transmission time to sample rates other than the current best.

- **Initial Rate**
  - When first associated with another station or AP, SampleRate starts off with the highest possible rate.

- **Quarantine**
  - A rate is considered unusable when it fails for four consecutive times. It is then quarantined for the next 10 seconds.

- **Rate Jumping**
  - SampleRate can change its current best rate to any of the rates available. It is not restricted to the immediate next lower or higher rate like ARF/AARF and AMRR.

### 4.1.5 Minstrel

Minstrel [24] is the most advanced RAA implemented in the madwifi project driver. Minstrel is an EWMA based algorithm with the goal to maximize the throughput by adjusting the retry chain in order for it to complete within a set time period. Minstrel uses EWMA to calculate the success of each rate by looking at previous transmission results. The calculation runs every 100ms, and the results of the transmissions of

<table>
<thead>
<tr>
<th>Bucket</th>
<th>Size From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>250</td>
</tr>
<tr>
<td>2</td>
<td>250</td>
<td>1600</td>
</tr>
<tr>
<td>3</td>
<td>1600</td>
<td>3000</td>
</tr>
</tbody>
</table>

*Table 4.6: madwifi SampleRate packet size separation*
the last 100ms are given a 25% weight in the EWMA calculation. This parameter of 25% can be set in the driver by a programmer.

**Fingerprinting Minstrel**

Minstrels behaviour is somewhat similar to SampleRates. There are a few differences, and these are what we want to find, as well as determine how to use these differences to be able to fingerprint Minstrel.

**Multi-Rate Retry Chain**

Minstrels MRR is populated based on timing parameters and throughput calculations. The developers of Minstrel found that they wanted to adjust the retry chain such that the entire chain can be completed within a time span of 26ms. They have taken this from TCP theory: according to [24], TCP starts to back off if a packet sent is not delivered in less than 26ms. This can be used to fingerprint the behaviour of Minstrel. Additionally, Minstrel tries to complete each segment in the retry chain within a set time, which defaults to 6ms. If every segment in the retry chain is given a time limit of 6ms, the entire chain would complete and fail in 24ms if every transmission and retry attempt fails.

Minstrel populates the rate part of the retry chain a little differently than SampleRate. SampleRate populates the chain by figuring out which rate will have the lowest transmission time, while Minstrel populates the chain by calculating which rate will have the best throughput, and then retries this rate for a maximum of 6ms before switching to the rate with the next best throughput. Table 4.7 shows Minstrels MRR population scheme during normal operation. By normal operation, we refer to transmissions which are not “Lookaround” transmissions.

The retry count for minstrel is somewhat different than with other RAAs. This is shown in Tables 4.7 and 4.8 as $\lambda$. $\lambda$ is calculated based on the transmission time of a packet. In the madwifi implementation of Minstrel, the driver uses a function in ath_hal called `ath_hal_computetxtime` for calculating the transmission time. This function calculates the time it takes to send a frame while taking the frame length, the modulation, the rate and the short preamble setting. By using this (and some other values), Minstrel calculates how many times it should retry every rate in the MRR Chain. This is advantageous for us as we can observe the retry chain, and especially focus on the retry count field. If this changes, we can be fairly sure it is Minstrel.

**Lookaround**

Minstrel defines a set number of percent of transmissions as “lookaround” transmissions. These are transmissions which probe the
wireless environment to build statistics which can be used when selecting the best rate. Minstrel uses the same scheme as SampleRate as it periodically sends out packets with a rate other than the current rate. It does this in order to probe other rates in the hope of finding a rate which may perform better than the current rate. Minstrel selects the lookaround rate according to a few simple rules, and does this for 10% of the transmissions. Additionally, Minstrel will randomize the lookaround percentage slightly in order to try to prevent similar behaviour of several stations operating in the same wireless environment. The MRR is slightly different for the lookaround transmissions than for the normal transmissions, which are shown in Table 4.8. We see that Minstrel has two different ways of populating the table. The difference is in segments 1 and 2. When Minstrel selects a lookaround rate, it compares the lookaround rate throughput with the current best rate’s throughput. If the current best rate is expected to provide a better throughput than the lookaround rate, Minstrel will place the lookaround rate in the second segment of the Multi-Rate Retry table. The developers of Minstrel did this after experimenting with the original implementation, and found that the lookaround transmissions lowered the average throughput of the connection when the wireless environment was very good and was operating at the highest rate possible with little loss. The result of this is that Minstrel will never transmit a packet with a rate which has a lower throughput than the current rate, while still being able to sample other rates if all transmissions in the first segment of the MRR fail.

If the selected lookaround rate is expected to provide a better throughput than the current best rate, Minstrel will put this rate in the first segment of the MRR, and as a result get statistics for this rate, and still be able to transfer the packet with the current best rate if this fails. The current best rate is placed in the second segment of the MRR.

<table>
<thead>
<tr>
<th>Segment</th>
<th>Rate</th>
<th>Retry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Best Throughput</td>
<td>λ</td>
</tr>
<tr>
<td>2</td>
<td>Next Best Throughput</td>
<td>λ</td>
</tr>
<tr>
<td>3</td>
<td>Best Probability</td>
<td>λ</td>
</tr>
<tr>
<td>4</td>
<td>Lowest Baserate</td>
<td>λ</td>
</tr>
</tbody>
</table>

**Table 4.7:** Minstrel Retry Chain during normal operation

<table>
<thead>
<tr>
<th>Segment</th>
<th>Lookaround Rate</th>
<th>Retry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Best Throughput random &lt; best</td>
<td>Random Rate</td>
</tr>
<tr>
<td>2</td>
<td>Random Rate random &gt; best</td>
<td>Best Throughput</td>
</tr>
<tr>
<td>3</td>
<td>Best Probability</td>
<td>Best Probability</td>
</tr>
<tr>
<td>4</td>
<td>Lowest Baserate</td>
<td>Lowest Baserate</td>
</tr>
</tbody>
</table>

**Table 4.8:** Minstrel Retry Chain during lookaround operation
**Exponential Weighted Moving Average (EWMA)**

Minstrel calculates the EWMA periodically, and defaults to every 100ms. This means that the current best rate cannot be changed more often than every 100ms. This can be used in our fingerprinting when listening to transmissions from a station. We can find the current best by looking at normal transmissions and observe how often this changes. If this time is about 100ms, it could be that we are observing traffic governed by Minstrel.

**Minstrel characteristics**

To summarize, we can detect Minstrel by observing the following characteristics:

- **Multi-Rate Retry population**
  - Lowest baserate at R$_3$.
  - Changing count for the retry fields (C$_{0-3}$).
  - Not necessarily decreasing rates from R$_{0-3}$. This means that retransmissions may be transmitted at a higher rate than the original rate.

- **Lookaround**
  - Every 10th packet (with some randomness) is a lookaround packet.
  - Rarely see lookaround packets when operating at the highest rate with minimal loss.

- **Frequency of best rate changes**
  - Best rate changes at most every 100ms.

- **Rate jumping**
  - Best rate may change by more than one step, just like SampleRate.

**4.1.6 PID**

The PID rate control [25] is one of the RAAs implemented in the mac80211 framework. The PID RAA is based on the Proportional–Integral–Derivative controller, next to Minstrel$^1$. The purpose of the PID controller is to provide a mechanism for calculating and adjusting the output based on three input parameters. The PID Controllers Proportional control decides the initial behaviour of the output. This may start slow or fast dependant on the desired outcome. The PID Controllers Integral control is responsible for adjustments over time.

---

and the Derivative control is in charge of trying to even out the adjust-
ment in order for the output to converge to a desired state. The values
of these three controls, the Proportional Coefficient (CP), the Integral
Coefficient (CI) and Derivative Coefficient (CD), are adjusted carefully
in order to achieve the desired output.

The PID RAA utilizes the Proportional–Integral–Derivative (PID)
Controllers mechanism by implementing the following equation which
calculates an adjustment value which is used to decide the transmission
rate for the next packet (Equation 4.1):

$$adj = CP \cdot err + CI \cdot err\_avg + CD \cdot (err - last\_err) \cdot (1 + sharpening)$$ (4.1)

adj
adjustment value that is used to switch TX rate (see below)

err
current error: target vs. current failed frames percentage

last_err
last error

err_avg
average (i.e. “poor man’s integral”) of recent errors

sharpening
non-zero when fast response is needed (i.e. right after association
or no frames sent for a long time), approaching zero over time

CP
Proportional coefficient

CI
Integral coefficient

CD
Derivative coefficient

CP, CI and CD are determined during compile time of the framework
and are set to 15, 9 and 15 respectively. After the calculation, PID tries
to find a better rate using the adjustment value looking for a rate which
has not shown a worse error rate than the current one.

The PID RAA does not continuously update its current best
rate. Like the many other RAA mechanisms, it adjusts its best
rate periodically. The mac80211 framework’s implementation of the
algorithm has this period set to 125ms. Although it is not certain that
it runs the update every 125ms (less updates may happen if no packets
has been sent during a period longer than the 125ms period), with a
reasonable amount of traffic, the adjustment may occur as often as every
125ms.
Fingerprinting PID

To be able to recognize traffic governed by PID, there are a few characteristics we could try to observe. Like SampleRate and Minstrel, PID is not restricted to changing to the immediate next higher or lower rate. PID may change the current best rate to any rate amongst the available rates as it sees fit. We could also try to track the frequency of best-rate changes. If the current best rate does not change more often than every 125ms, this could be an indication that the traffic is PID.

PID characteristics

To summarize, PID has the following characteristics:

- Frequency of best rate changes
  - Best rate changes at most every 125ms, but the upper limit may vary.
- Rate jumping
  - Best rate may change to any available rate.

4.2 Retry Strategies

Different drivers and RAAs implement different strategies for retransmissions. A retransmission happens when the receiver either does not acknowledge, or sends a negative acknowledgement, for a transmitted frame. The sender can retransmit the frame at the same rate as the original frame or select another. This section will list three retransmission strategies.

4.2.1 Strategy A: Simple Fallback

This is the simplest of strategies. It basically tries all rates once until it hits its predetermined limit or reaches the base rate of 1Mbps. It then gives up and declares the packet as lost. Algorithms known to use this strategy are the madwifi implementations of AMRR and Onoe.

The madwifi implementations of AMRR and Onoe are described in Sections 4.1.2 and 4.1.3.

4.2.2 Strategy B: Best-rate or Base-rate

This is a very simple strategy. It is implemented in the madwifi version of the SampleRate RAA and uses two or three different rates when retrying a transmission. The base rate is always the last attempted rate. This strategy is recognizable by seeing the best-rate attempted a set number of times followed by a number of base-rate transmission attempts. Variations of this strategy are seen in the implementation of SampleRate in the madwifi driver where there can be three different rates when retransmissions occur while sampling the channel.
4.2.3 Strategy C: Intelligent Selection

A more advanced type of retry strategy is the Intelligent Selection strategy. This strategy is implemented by the Minstrel RAA which selects the retry rates based on a set of different metrics. As implemented in madwifi, the rate selection are based on the performance and the probability of transmission success. The base-rate is always selected as the last resort if all else fail.

4.3 Summary

This section contains a summary of the fingerprinting methodology. We have analysed RAAs used in the madwifi driver and have extracted metrics that can be used to map traffic patterns against them. The results from our research are summarized in Table 4.9.

<table>
<thead>
<tr>
<th></th>
<th>AMRR</th>
<th>Onoe</th>
<th>SampleRate</th>
<th>Minstrel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use RTS/CTS</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Excludes rates</td>
<td>-</td>
<td>-</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Use sampling</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Sampling ratio</td>
<td>-</td>
<td>-</td>
<td>10%</td>
<td>&lt;= 10%</td>
</tr>
<tr>
<td>Rate changes pr. sec.</td>
<td>-</td>
<td>2</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Rate jumping</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Min. time betw. rate ch.</td>
<td>-</td>
<td>500ms</td>
<td>100ms</td>
<td>100ms</td>
</tr>
<tr>
<td>Starting rate (Mbps)</td>
<td>36, 24</td>
<td>36, 24</td>
<td>54, 36, 11</td>
<td>&lt;= 36</td>
</tr>
<tr>
<td>Sampling down</td>
<td>-</td>
<td>-</td>
<td>yes</td>
<td>when appropriate</td>
</tr>
<tr>
<td>Rate quarantine</td>
<td>no</td>
<td>no</td>
<td>10sec.</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 4.9: Fingerprinting Summary: Madwifi r4180
Chapter 5

Implementation

This chapter contains information about the Rate Adaptation Classifier (RAC). RAC is a small piece of software which aims to present the user with useful information regarding wireless stations in the near vicinity. RAC uses a standard WLAN interface in monitor mode, and supports the usual 802.11a/b/g standards as well as the newer 802.11n.

This small piece of software is built using inspiration from another tool aimed at analysing the behaviour of wireless networks, namely horst [14].

The RAC software is built on a number of different modules. These modules are logical pieces handling different parts of the process of capturing and analysing the wireless traffic.

5.1 Packet Capture Engine

To be able to analyse packets that are transmitted over the air between an Access Point and a station, we need to configure the kernel to report every frame received by the wireless interface. This mode is called promiscuous mode and is enabled by either using exported functions in one of several frameworks for packet capturing or by setting the interface manually to promiscuous mode using a kernel syscall or an application such as ifconfig [2].

When capturing packets, there are several libraries one can use in order to ease the process, or one can capture raw packets by manually setting the interface into promiscuous mode and reading packets from it.

When capturing packets from a wireless interface, it is most likely possible to capture packets prepended with either the Radiotap– [1], the Prism– or the AVS header format. These are headers which allows the driver to add information to the packet in order for user-space applications to read different statistics and/or driver-related information before reading the actual packet content.
5.1.1 Radiotap Header

The Radiotap header is one of the mechanisms to supply driver information from kernel to userspace. The Radiotap header is prepended to the packet when the capturing interface is put into monitor mode. The Radiotap header is shown in Listing 5.1. The header is built up of four variables and its entire size is 8 bytes. The \textit{it\_version} field is always 0 and contains the major version of the Radiotap header. The \textit{it\_pad} field is there to align the next 16bit field to a 2byte boundary. The \textit{it\_len} field is the length of the Radiotap header plus the data contained in it. The \textit{it\_present} field contains a bitmask of the information stored in the header. It defines which data is stored in the data section after the header. If bit 0 is set in the bitmask, the TSFT field is present in the data section. The data section is the data immediately following the Radiotap header bitmask.

The defined fields in the Radiotap Header are as shown in Table 5.1, and as we can see, there can be quite a lot of information stored in the header. The Radiotap header is also extensible in order to support vendor information and future extensions to the protocol. By setting bit 31, one can signal that another 4-byte bitmask will follow, and this bit can be set several times effectively chaining as many bitmask as desired.

An important note about the Radiotap header is that it stores all its information in Little Endian byte-order and that the fields are strictly ordered. The data following the bitmask(s) have to appear in the order in which they are defined in the bitmask. If bit 0,1,2 and 3 are set, the data following the header have to be the TFST–, Flags–, Rate and Channel fields – in that order.

Another important characteristic of Radiotap is that all fields in the data part of the header must be aligned to natural boundaries. In the example from the Radiotap website\(^1\), a structure is defined as in Listing 5.2. This is wrong since the \texttt{tx\_attenuation} variable is not defined on a 16-bit boundary. To correct this, the developer needs to align the variable by padding the previous variable up to the correct boundary as seen in Listing 5.3.

\(^1\)http://radiotap.org

\begin{verbatim}
struct ieee80211_radiotap_header {
    u_int8_t it_version; /* set to 0 */
    u_int8_t it_pad;
    u_int16_t it_len; /* entire length */
    u_int32_t it_present; /* fields present */
} __attribute__((packed));
\end{verbatim}

\textbf{Listing 5.1: Radiotap Header Structure}
<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TSFT</td>
</tr>
<tr>
<td>1</td>
<td>Flags</td>
</tr>
<tr>
<td>2</td>
<td>Rate</td>
</tr>
<tr>
<td>3</td>
<td>Channel</td>
</tr>
<tr>
<td>4</td>
<td>FHSS</td>
</tr>
<tr>
<td>5</td>
<td>Antenna Signal</td>
</tr>
<tr>
<td>6</td>
<td>Antenna Noise</td>
</tr>
<tr>
<td>7</td>
<td>Lock Quality</td>
</tr>
<tr>
<td>8</td>
<td>TX Attenuation</td>
</tr>
<tr>
<td>9</td>
<td>dB TX Attenuation</td>
</tr>
<tr>
<td>10</td>
<td>dBm TX Power</td>
</tr>
<tr>
<td>11</td>
<td>Antenna</td>
</tr>
<tr>
<td>12</td>
<td>dB antenna signal</td>
</tr>
<tr>
<td>13</td>
<td>dB Antenna Noise</td>
</tr>
<tr>
<td>14</td>
<td>RX Flags</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>19</td>
<td>MCS</td>
</tr>
<tr>
<td>29</td>
<td>Radiotap Namespace</td>
</tr>
<tr>
<td>30</td>
<td>Vendor Namespace</td>
</tr>
<tr>
<td>31</td>
<td>Reserved, another bitmask follows</td>
</tr>
</tbody>
</table>

Table 5.1: Radiotap defined-fields

```c
struct rtapdata {
    uint8_t antsignal;
    uint16_t tx_attenuation;
    uint8_t flags;
    uint16_t rx_flags;
} __attribute__((packed));
```

Listing 5.2: Radiotap Header Alignment: wrong

```c
struct rtapdata {
    uint8_t antsignal;
    uint8_t pad_for_tx_attenuation; // <-- added
    uint16_t tx_attenuation;
    uint8_t flags;
    uint8_t pad_for_rx_flags;       // <-- added
    uint16_t rx_flags;
} __attribute__((packed));
```

Listing 5.3: Radiotap Header Alignment: correct
5.1.2 PRISM Header

The PRISM header is a consistent 144 bytes long header which is prepended to the 802.11 WLAN header when a WLAN interface who supports it is set to monitor mode. The PRISM header, as seen in Listing 5.4, contains information about the physical characteristics of a received frame. The information contained within the header is amongst others the physical rate, which is very important for this work.

5.1.3 AVS Header

The AVS header was designed to replace the PRISM header format. The AVS header format extends the PRISM header by providing more information about the capture when capturing 802.11a and 802.11g packets.

5.1.4 Capturing RAW Packets

Capturing RAW packets from an interface is quite easy when working with most Linux WLAN drivers. As long as the driver supports Monitor mode, one can capture all packets the interface is able to see.

By capturing RAW packets, the application needs to implement all the logic surrounding the packet capture process. This means opening the device, setting the device into promiscuous mode and controlling the read process.

5.1.5 Capture using PCAP

Linux libpcap [3] implements the Packet Capture (PCAP) API. The libpcap library hides much of the configuration and set-up process from the developer and provides a powerful api for controlling the network interface. Many popular applications use libpcap for capturing packets, e.g. Wireshark [4]. RAC is implemented using the PCAP library as it supports the Radiotap extensions.

5.1.6 Parsing Radiotap Headers

Parsing radiotap headers is done using the Linux kernel radiotap parser. This parser provides a callback-like interface for extracting information from the radiotap header. RAC use this parser in order to extract information like the PHY bit-rate from received frames.

5.1.7 Parsing 802.11 WLAN Frames

Parsing 802.11 WLAN frames is manually performed by RAC. RAC contains a rudimentary 802.11 WLAN header parser which only extracts the important bits. The extracted values are the packet length, frame type, the sender – and the destination address.
**Listing 5.4:** PRISM Header Structure

```c
struct prism_value
{
    uint32 did;
    uint16 status;
    uint16 len;
    uint32 data;
};

struct prism_header
{
    uint32     msgcode;
    uint32     msglen;
    u8char     devname[16];
    struct prism_value hosttime;
    struct prism_value mactime;
    struct prism_value channel;
    struct prism_value rssi;
    struct prism_value sq;
    struct prism_value signal;
    struct prism_value noise;
    struct prism_value rate;
    struct prism_value istx;
    struct prism_value frmlen;
}
```

**Listing 5.5:** AVS Header Structure

```c
struct AVS_header
{
    uint32 version;    /* Header Version */
    uint32 length;     /* Header Length */
    uint64 mactime;    /* MAC timestamp */
    uint64 hosttime;   /* Host timestamp */
    uint32 phytype;    /* PHY type */
    uint32 channel;    /* WLAN Channel */
    uint32 datarate;   /* PHY Rate */
    uint32 antenna;    /* Antenna */
    uint32 priority;   /* Priority */
    uint32 ssi_type;   /* SSI Type */
    int32 ssi_signal;  /* SSI Signal */
    int32 ssi_noise;   /* SSI Noise */
    uint32 preamble;   /* Preamble */
    uint32 encoding;   /* Encoding Type */
};
```
5.1.8 Filtering Frames

When capturing frames, there are many frames that we are not interested in. For example, frames which contain no data, such as NULL frames. Most control frames are not interesting for this work, but control frames such as RTS and CTS are used by some RAA algorithms, which may help us in the classification process.

In order to filter out the correct frames, and to do this the most efficient way, RAC filter frames as soon as they have been parsed. The parser does some filtering, for example filtering out frames which do not contain all the information we need for the application to be able to classify them. The filter process is very simple, and can be summarized as follows:

1. try to match the sender to a predefined stations MAC address
2. see if the packet type is DATA

5.2 Frame Parse Validation

In order to be sure that the RAC parses frames correctly, we need to validate its results. Most importantly, we need to check that the captured and analysed frames rate and sequence number are correctly interpreted.

We are going to do this by simultaneously capturing traffic using two monitor devices, one running the RAC capture engine, the other one running Wireshark [4]. These two monitor devices are placed close to one another, and both are running the same type of hardware. We are going to do this experiment using different 802.11 modes, the 802.11b/g, and the 802.11n mode. In 802.11n mode, we will test the capture engine using various 802.11n modes.

All configurations use the same access-point and the same monitor devices. The access-point is running on one of the ND-Lab testbed nodes, see Section 6.1, running Fedora Linux with a custom Kernel 3.4.0-rc4-wl+. The wireless adapter is a D-Link DWA-547 using the ath9k driver. The wireless interface is configured as an access-point using hostapd v0.8.x.

The monitor devices, which are running the RAC capture engine and Wireshark, are running the same wireless hardware as the access-point. These two machines are placed close to each other in an attempt to synchronise which frames the monitors are able to capture. We can not hope to capture the exact same packets on both monitors, but we can try as good as we can.

The stations used in these experiments are for the single spatial stream case a laptop\(^2\), and a USB Buffalo WLI-UC-G300N\(^3\) wireless adapter running Ubuntu Linux 12.04 with custom kernel 3.4.0-rc4-wl+.

\(^2\)Dell Vostro V130 v/Atheros Communications Inc. AR9285 Wireless Network Adapter running Ubuntu Linux 12.04 with custom kernel 3.4.0-rc4-wl+.
\(^3\)http://www.buffalotech.com
adapter which has support for two spatial streams and rates up to and including 300Mbps.

The RAC capture engine contains a mac-address filter which by default filters out all captured frames not sent from a specified station. Wireshark is a little different, but when used with a filter (shown in Listing 5.6), it only captures frames we are interested in. The filter has a few different components, the wlan.sa is the WLAN Source Address, the wlan.fc.type_subtype filters out NULL and RTS frames and the wlan.fc.retry filters out retransmissions. We do not care about retransmissions at this point, and only capture the first transmission of every frame. The RAC capture engine does not differentiate between the original transmission and a retransmission of the same frame other than setting the retransmission flag in the structure containing frame information.

After the capture, data from both RAC and Wireshark are exported to text-files and analysed. The analysis compares sequence-numbers, time of capture and rates from both capture sources. If there are any inconsistencies found in the rates reported by the two capture sources, it is reported.

Validation Results
To ensure correctness of the RAC capture engine, we are conducting a set of experiments.

The first test, using the Buffalo WLI-UC-G300N USB wireless adapter, tests the RAC capture engine over the most parts of the one- and two-stream cases. It jumps from short to long guard interval and also skips between the used bandwidths. This is the result of the dynamic environment presented in Section 2.1.2. Results from this experiment are seen in Table 5.2.

The second test, using the Vostro V130s built-in Atheros wireless adapter only covers test cases using a single spatial stream. The wireless adapter does not support more than one spatial stream. Table 5.2 shows the number of captured and matched frames between the two capture sources. No inconsistencies were found other than the case where one capture source could not capture the first frame when this frame required one or more retransmissions. It was also discovered that the Buffalo wireless interface did not set the retransmit flag in the MAC frame FCS field when retransmitting a frame. The only way to determine the frame as a retransmission was to look at the sequence number and the time of transmission. Table 5.3 shows

```
wlan.sa == 88:9f:fa:0d:28:e1
and wlan.fc.type_subtype != 0x1b
and wlan.fc.type_subtype != 0x24
and wlan.fc.retry == 0
```

Listing 5.6: Wireshark filter for station capture
the results from the rate-validation tests performed on the Dell Vostro V130. Again, both capture sources reported the same bit-rate and sequence number.

5.3 Packet Classification Engine

Captured packets have to be classified in the current context of the capturing process. There are several types of behaviours we are looking for when trying to classify the different rate adoption algorithms.

- sampling events
- best rate change
- retransmission
- RTS/CTS

5.3.1 Sampling Events

Sampling happens when the RAA decides to send a frame with a different rate than the current best rate in order to probe for changes in the wireless channel. To be able to determine whether a packet is a sampling packet, we need to analyse at least three consecutive packets.

We start by defining a sampling event as an event where, amongst three consecutive frames, the middle frame is sent with a different rate than the first and the third frame, see Figure 5.1. As the capture process is lossy, one cannot always capture three consecutive frames travelling over the wireless channel. Another issue is when a sampling event occurs immediately before a best-rate change in the RAA. This is shown in Figure 5.2, where the RAA changes its current best rate immediately following a sampling event. In this figure, frame 100 is transmitted using the current best rate of rate A, followed by a sampling frame transmitted using rate B. The RAA, in this case, runs its best-rate calculation immediately following frame 101, decides that rate B is better than rate A, and changes its current best rate to B. Thus, this event is not considered to be a sampling event by the analysis part, even though it is a sampling event for the RAA. Being able to classify this as a sampling event is hard. One can, in SampleRate's case, assume the event to be a sampling event by using knowledge about the RAA, its frequent and predictable sampling characteristics, but this would result in an assumption that is not necessarily true as one cannot assume the RAA to be SampleRate without prior knowledge about the station's driver. Listing 5.7 shows pseudo-code for detecting a sample event. This code represents Figure 5.1 where there are three consecutive captured frames with the rate of the middle frame being different from the first and the third.
Table 5.2: Comparison between the RAC capture engine and Wireshark using the Buffalo WLI-UC-G300N USB Wireless Adapter

<table>
<thead>
<tr>
<th>MCS index</th>
<th>Matched frames</th>
<th>GI = 800ns</th>
<th>GI = 400ns</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>20MHz</td>
<td>40MHz</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>43</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>91</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>66</td>
<td>185</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>31</td>
<td>389</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>259</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>119</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>441</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>13</td>
<td>49</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>52</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1115</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>19</td>
<td>1339</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>21</td>
<td>1414</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>26</td>
<td>79</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>18</td>
<td>75</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.3: Rate match between the RAC capture engine and Wireshark using the Atheros AR9285 Wireless Network Adapter

<table>
<thead>
<tr>
<th>MCS index</th>
<th>Matched frames</th>
<th>GI = 800ns</th>
<th>GI = 400ns</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>20MHz</td>
<td>40MHz</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>260</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>288</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>217</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>821</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1223</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1964</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>2881</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1062</td>
<td>0</td>
</tr>
</tbody>
</table>

Listing 5.7: Pseudocode for sample event detection

```javascript
function is_sample (a, b, c)
if (a.rate != c.rate)
    return false /* first and third rate differ */
if (a.rate == b.rate)
    return false /* first and second rate is the same */
if ((a.seq == b.seq - 1) && (b.seq == c.seq - 1))
    return true /* sequential */
return false
```
5.3.2 Best Rate Change

In order to determine a best rate change from the RAA, we need to analyse at least two frames, but three frames are preferred. By analysing three frames, one can first run the sampling classification before running the best-rate classification, thus not classifying a sampling event as a best-rate change event. We define four types of packet sequentiality which classify a rate-change event. The four types are shown in Table 5.4 and are determined by the sequence numbers of the captured frames.

In the event of analysing **SEQUENTIAL** frames, one can determine if it is a sampling event by checking the parameters as defined in Section 5.3.1, Listing 5.7. If it is a sampling event, do not classify it as a best-rate change event. If the captured frames are of the types **SEQUENTIAL_BEFORE**, **SEQUENTIAL_AFTER** or **NON_SEQUENTIAL**, we cannot classify them as a sampling event, but we may still look at

<table>
<thead>
<tr>
<th>Type</th>
<th>Sequence number</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQUENTIAL</td>
<td>(n - 1) (n) (n + 1)</td>
</tr>
<tr>
<td>SEQUENTIAL_BEFORE</td>
<td>(n - 1) (n) (n + j)</td>
</tr>
<tr>
<td>SEQUENTIAL_AFTER</td>
<td>(n - i) (n) (n + 1)</td>
</tr>
<tr>
<td>NON_SEQUENTIAL</td>
<td>(n - i) (n) (n + j)</td>
</tr>
</tbody>
</table>

Table 5.4: Packet capture sequentiality
at best-rate change event. A best-rate change event is an event where frame $n + 1$ is transmitted at a different rate from frame $n$ and the rate change is not a result of a sampling event as stated above.

### 5.3.3 Capturing Frames from Device

When enabling the RAC Capture Engine before enabling the driver, thus capturing all management and control data before any data has been transmitted, this may fool the capture engine into believing it has lost several frames. This is because of the sequence number reset early in the communication between the station and the access-point, which similar to a reset event that occurs when the access-point and the station have been without contact for a longer period of time. Although this event may cause some errors in the statistics reported by the RAC Classification Engine, it can be an advantage when analysing the results. By capturing a reset event, one can gain insight into the early stages of the RAA because of the reset of the algorithm. All algorithms needs a starting point, a "base" rate for early communication, and this can be observed by looking at a reset event.

### 5.3.4 Measuring Frame Loss

RAC tries to measure the number of lost frames using the sequence numbers from captured frames. RAC knows the sequence number of the last captured frame and uses it to calculate the number of lost frames when a new frame arrives. If the new frame is the expected frame, i.e. the last sequence number plus one, no frames are lost. If the sequence number is larger than expected, RAC calculates the number of lost frames by subtracting the sequence number of the previously captured frame from the sequence number of the new frame. The difference is the number of lost frames. If the sequence number is lower than expected, RAC calculates the number of lost frames using Equation 5.1. This calculation is only performed after a sequence number reset event.

$$\text{lost} = \text{lost} + ((\text{new}\_\text{seq} + 4096) - \text{prev}\_\text{seq})$$  \hspace{1cm} (5.1)

### 5.4 RAC Statistics and Classification Engine

This section will explain the statistics and classification engine of RAC.

#### 5.4.1 Statistics

There are several statistics which are calculated and kept track of by RAC.

The statistics engine is responsible for several capture related metrics. It separates between the different IEEE 802.11 frame types; DATA, CTRL and MGMT. The engine counts the number of received
frames, additionally counts RTS and CTS CTRL frames, and for DATA frames keeps track of the following:

- number of received frames
- number of received retransmits
- number of retransmitted frames
- estimation of lost frames
- keeping track of current:
  - Sequence number
  - Physical rate
  - Frequency
  - Channel
- additional IEEE 802.11n metrics:
  - MCS index
  - Guard Interval
  - Bandwidth

5.4.2 Classification

The classification engine of RAC is responsible for classifying the following:

- Retransmissions
- Best-rate change event
- Sample event

The classification engine always keeps a pipeline of five frames. The middle frame is the one being analysed, and the other four are there to aid the classification process.

Sampling

A sampling event is defined by three successively received frames where the middle frame’s bit-rate is different from the first and the last. In addition, the first and the last frame’s bit-rate must be the same as the current best-rate.

Best-rate Change

A best-rate change event is classified by the frame not being a sample frame and the bit-rate of the frame being different than the bit-rate of the previous frame.
5.5 Procedure for RAA classification

This section presents a methodology for classifying RAAs using RAC. There are several steps required to be able to recognize a rate adaptation algorithm based on the output of RAC.

5.5.1 Capture Process

Capturing frames using RAC is the first step. RAC runs on a computer having installed a wireless network device capable of running in monitor mode. RAC needs two arguments: the WLAN device name and the MAC address of the target station. The WLAN device additionally needs to be configured to listen on the correct channel. This can be done in many ways, but using the `iw` tool is preferred as it can configure 40MHz channels.

The traffic pattern is important during the capture process. To get the information needed, RAC is dependant on the station sending as many frames as possible. Frames that the target station receives are discarded as the bit-rates used for these frames are decided by the access point. The bit-rate selection is only applied to the station's transmitted frames. The preferred way of generating traffic on the target station is by running a network performance tool transmitting frames as quickly as possible. The tests should be measured for an extended time period, preferably over a minute. The station can be moved closer to or further away from the access point in order to trigger rate adaptation in the driver. If the station is too close to the access point and there is little to no interference in the wireless channel, most algorithms will select the best rate and stay there. To get the best quality of captured data, we need to try to degrade the quality of the channel, which can be achieved by moving the station farther away from the access point. If possible, we should also try to keep the capture device close to the station.

After capturing data and quitting RAC, the output is written by default to `/tmp`. RAC writes seven files to `/tmp`, and these are listed below:

- **losses.dat** Log of estimated lost frames. This is losses to the capture process, not necessarily losses between the station and the access point. The file contains two columns, the first is unix time, the second is the estimated number of lost frames.

- **rate_changes.dat** This file contains the rate changes seen during the capture process. The first column is unix time, the second is the new bit-rate and the third is the sequence number on the first frame seen with this bit-rate.

- **rates80211bg.dat** This file contains the number of frames seen for the different 802.11 legacy bit-rates. The first column is the bit-rate,
the second is the number of frames and the last is the frame count in percent to the total number of frames.

rates80211n.dat This file contains the distribution of IEEE 802.11n frames seen, it has five columns with data. The first is the MCS index (see Section 2.4 on page 16 for more information). The second and third column is the number of captured frames for the long Guard Interval (GI) frames in 20MHz and 40MHz channel widths. Columns four and five represent the number of frames received for the short GI 20MHz and 40MHz channel widths.

retries.dat This file contains captured retransmission rates. The file contains the time, the original bit-rate and the bit-rate of the retransmitted frames. The first column is unix time, the second is the total number of retransmissions, the third column is the original frame's bit-rate and the remaining columns are the bit-rates of every retransmission following the original. If a frame has been retransmitted four times, there will be four columns following the three first.

samples.dat This file contains captured samples. The file structure is unix time in the first column and the bit-rate of the captured sample in the second.

statistics.dat This file is a dump of the statistics kept by RAC. These are:

- Number of captured frames
- Estimated lost frames
- Captured retransmits
- Loss ratio
- Total number of rate changes
- How many rate changes with a change of more than one rate step
- Maximum rate changes per second
- Minimum time between rate changes
- Minimum number of frames between rate changes
- Total number of samples
- Sampling ratio
- Minimum number of frames between samples
- Number of RTS frames
- Number of CTS frames
- Maximum number of retransmissions for a single frame
5.5.2 Analysis

Analysing the captured data is the next step in classifying observed traffic. By using the output of RAC from the capture process, we will systematically look at the captured data.

The statistics.dat file contains most of the information we need. We will use this file to figure out if the observed algorithm uses RTS/CTS, sampling (and the sampling ratio), rate jumping and determine the maximum number of observed retransmissions for a frame. The file also contains the maximum number of rate changes per second and the minimum time between rate changes.

To determine the retransmission strategy used, we need to use the retries.dat file. By looking at the bit-rates for frames needing the most retransmissions and comparing this to the defined strategies seen in Section 4.2, we can determine the retransmission strategy.

Excluded rates can be determined by looking at the rate distribution in rates80211bg.dat. If we see zero frames being transmitted using the 9Mbps rate, but there are several frames being sent at both 6Mbps and 11Mbps, we can assume an exclusion of the 9Mbps rate.

To figure out the starting rate, we need to locate events where the RAA is being reset. There are mainly two events which cause the RAA to be reset. When the station connects to an access point and when the driver is being reset because of a temporary disruption in network connectivity. By observing the rate changes immediately after event such as these, we may be able to determine the starting rate. The most effective way is to run the capture process when connecting a station to the access point.

5.5.3 Comparison

Finally, we compare our results to metrics gathered by research and previously measured values. In Section 4.1, we have gathered information about the RAAs present in the madwifi driver and in Section 8.1, we will present measurements of the same driver.

There are measurements which weigh more than others. The retry strategy, for example, is one of the most important measurements. Below is a list of measured ranging from the most important at the top to the least important at the bottom.

- Retry strategy and max retries
  - From our research, we see a big difference in how the different algorithms handle retransmissions.

- Rate jumping
  - Rate jumping is the main separator between madwifis two least advanced RAAs (AMRR and Onoe) and the two advanced RAAs, SampleRate and Minstrel.
• Max rate changes per second and minimum time between rate changes
  – This metric is very dependent on how fast the algorithm reacts to changes in the wireless channel.
• Sampling rate
  – SampleRate and Minstrel can be separated by looking at this value.
• Starting rate
  – Different algorithms choose the bit-rate of the first frame based on different rules. Some chose the rate from a static table, others, i.e. SampleRate choose the bit-rate of the first frame based on the Signal-To-Noise (SNR) ratio of the wireless channel.
• Excluded rates
  – We know that SampleRate never used the 9Mbps rate.
• Use RTS/CTS
  – Some algorithms may use RTS/CTS when communicating with other wireless devices.

After weighing the different metrics against each other, it should be possible to classify observed traffic against a known RAA.

Summary

This section presented the design and implementation of the Rate Adaptation Classifier and a method of classifying rate adaptation algorithms by using the output from said tool. We have presented a thorough work on fingerprinting RAAs implemented in the madwifi driver and identified important metrics and behaviour of the algorithms. We have explained the implementation of RAC and finally explained the output generated by RAC and how to use this output when trying to classify algorithms. The next part of this thesis will show how we use the tool and method presented here in order to validate its effectiveness.
Part IV

Experimental setup
Chapter 6

802.11 Testbed

This section contains information about the testbed used throughout this thesis. The testbed contains a number of computers equipped with a number of different wireless network adapters. All nodes are connected through a gigabit backbone wired network allowing all machines to be remote controlled while running wireless experiments.

6.1 Hardware

Testbed nodes consist of a number of Dell OptiPlex GX620 computers. These have the technical specifications shown in Table 6.1. The nodes are equipped with wireless network adapters as shown in Table 6.2.

6.2 Software

The testbed nodes are running Fedora 14 with the stock Linux Kernel (2.6.35). Some nodes are configured with different kernels which is mentioned where applicable. Support tools and software have been installed in all testbed nodes.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel(R) Pentium(R) 4 CPU 3.00GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>1GB PC2-4200 (533MHz)</td>
</tr>
<tr>
<td>Chipset</td>
<td>Intel® 945 Express</td>
</tr>
<tr>
<td>Network</td>
<td>Broadcom Corporation NetXtreme BCM5751 Gigabit Ethernet PCI Express (rev 01)</td>
</tr>
<tr>
<td>Video</td>
<td>Integrated Intel Graphic Media Accelerator 950</td>
</tr>
<tr>
<td>BIOS</td>
<td>Version: A11</td>
</tr>
</tbody>
</table>

Table 6.1: Testbed Technical Specifications
<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Model</th>
<th>Chipset</th>
<th>802.11 standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-Link</td>
<td>DWL-G520</td>
<td>Atheros AR5001X+</td>
<td>802.11b/g</td>
</tr>
<tr>
<td>D-Link</td>
<td>DWA-547</td>
<td>Atheros AR922X</td>
<td>802.11b/g/n</td>
</tr>
<tr>
<td>Compex</td>
<td>WLM54AG</td>
<td>Atheros AR5413</td>
<td>802.11b/g</td>
</tr>
<tr>
<td>AirForce One</td>
<td>-</td>
<td>Broadcom BCM4318</td>
<td>802.11b/g</td>
</tr>
</tbody>
</table>

Table 6.2: Testbed Wireless Hardware
Chapter 7

Experiment

This chapter contains details about the experiments conducted in this thesis. The main experiment are the validation tests run on the madwifi rate adaptation algorithms.

The madwifi driver is the driver previously used for most of the Atheros Wireless Network Cards. The rate adaptation algorithms implemented in this driver are Onoe, AMRR, SampleRate and Minstrel.

In our experiments, we will use madwifi driver version svn r4180. This is a custom compiled driver used in a Fedora 14 system running the Linux Kernel 2.6.35.

7.1 Test Details

These tests have been conducted using a variety of different machines and wireless interface cards. The main goal of these experiments is to compare the observed behaviour of open-source drivers to the expected theoretical behaviour of the same driver. By using the results from these initial experiments, we are better equipped to analyse the behaviour of other drivers which may be proprietary and where no source code have been released by the manufacturer of the product.

To ensure correct results from the madwifi validation tests, two types of tests were conducted, a stationary and one moving. The stationary tests were conducted in order to observe the algorithms in a semi-stable environment, while the moving tests were conducted in order to observe the algorithms in a highly changing environment. The moving tests were done using the natural structure of the forth floor of the Informatics building at the University of Oslo. The test were conducted using a moving AP and a stationary station. This because of the difficulty of moving the station wireless hardware. By moving the AP, we could perform the tests using a normal laptop running on batteries while walking with the device around the facility. The route of the moving AP is as shown in Figure 7.2.

The stationary tests were conducted by fixing the Access Point in an office down the hall from the lab. Both the stations and the AP were fixed during all the stationary experiments. The distance between
the AP and the stations was chosen after several other positions were considered, both closer to the station and further away. We wanted to see the behaviour of the algorithms when the channel was somewhat congested such that we do lose some packets, but we did not want too much interference. The tests were meant to be as close to a normal working scenario as possible where several APs are placed in close vicinity and there are multiple networks occupying most, if not all channels. The reason for wanting some degree of congestion in the network is that we want the algorithm to behave as it would during normal operation.

All tests were executed six times, three times in the stationary configuration and three times in the moving configuration. Tests were run for 100 seconds. Iperf was used as the traffic generator, transmitting as much data as it possibly could to a receiving iperf instance. The receiver was running on either a wired computer attached to the access point or on the access point directly.

During all tests, a computer running a wireless monitor interface and RAC was placed close to the station. RAC was started and kept running during all the 100 second experiments.

The moving tests were conducted by configuring a laptop as an access point using the hostapd software. The laptop was carried around the path seen in Figure 7.2 while using a stop-watch to keep track of time.
Figure 7.1: Validation tests: Fixed AP

Figure 7.2: Validation tests: Moving AP. 10 and 50 is the time (in seconds) where the AP is located during the test.
Part V

Evaluation Results
Chapter 8

Classifying Known Algorithms

In this section we will present tests of RAC using different wireless hardware and drivers.

8.1 Madwifi validation tests

This section contains results from validation tests conducted on the madwifi driver for Atheros based chipsets. All algorithms, AMRR, Onoe, SampleRate and Minstrel have been tested and validated against expected results.

8.1.1 Analysing AMRR

Using knowledge from Part II, we will run a set of tests and observe how the driver behaves. We will try to match this with the previously researched behaviour of the algorithm.

First, we will analyse the retry chain of AMRR. The AMRR retry chain is built based on the information from Table 4.3 on page 34 which states that the AMRR algorithm populates the retry chain using the following rules:

1. current best rate/once
2. next lower rate/once
3. next next lower rate/once
4. lowest base rate/once

We would expect to see this behaviour when trying to fingerprint AMRR in our experiments. Listing 8.1 shows several snippets of the result file generated by RAC when capturing iperf traffic governed by the algorithm. The file contains three or more columns of information, starting with the time-stamp of when the original frame was received in the first column. The next column contains the total number of
retransmissions and the third column contains the physical rate of the originally transmitted frame. The next $n$ columns contains the physical rate of the retransmitted frames, and if no losses occurred in the capture process, one line in this file represents the current retry chain until the frame is either transmitted successfully or discarded following the information set in the retry chain.

As expected, the retry chain is populated using AMRRs procedure as shown in the list above. Starting from the top of the results, we can confirm that when the current best rate is 18Mbps, the retry chain is populated as shown in Table 8.1. Following the results in Listing 8.1, we see that the behaviour of the retries follows the same structure as the result in line 1 of Listing 8.1. We also see that, when the current best rate is 5.5Mbps or lower, there are not enough rates to choose from when populating the retry chain, and hence, the retry chain is not fully utilized. When the original frame is transmitted using 5.5Mbps, the retry chain is populated as seen in Table 8.2. As the current best rate falls even lower, the retry chain is populated using fewer and fewer records until the current best rate is 1Mbps, in which case there are no retries.

Analysing AMRRs rate changes is the second step in deciphering AMRR governed traffic. Using RAC, we have captured all rate changes during 100 second upload experiments. We see from Figure 8.1 how AMRR behaves in our validation tests. The main confirmation we get from these results is the behaviour of AMRR when changing the best rate. AMRR is expected to change rates one rate-index at a time, and this is seen in Figure 8.1. We observe AMRR fluctuating strongly, especially in the moving tests, see Figure 8.1(a) where we have spikes in the graph. This is because of the driver reconnecting to the AP when loosing connectivity over a predetermined length of time. If the connection is having difficulties exchanging packets at 1Mbps and the driver is struggling with management packets, the connection is reset and the session is renegotiated. The selected RAA is then notified by the reset event and the algorithm is reset. Although the reset code is dependent on the implementation on the algorithm, all algorithms in the madwifi driver reset the algorithm to the initial state. AMRR never changes best rate directly from 1Mbps to 36Mbps, but 36Mbps is the default starting rate in madwifis implementation of AMRR.

In addition to the rate changes observed by RAC, we have included the statistics output created by the application. RAC creates some simple statistics, see Section 5.4.1 on page 55, and this can be used when analysing RAAs. We see from the statistics of AMRR shown in Table 8.3 that AMRR at most changed rate twice a second and that, as expected, it does not use sampling, it does not jump rates when changing best-rate, and it has at maximum three retry attempts. The minimum time between rate changes is reported to be 397ms which is a bit more frequent than we expect from the implementation. We did not spend time trying to uncover why we see this discrepancy between the implementation and the captured data.
Listing 8.1: Retry chain experiment 1, AMRR (sections of retries.dat)

<table>
<thead>
<tr>
<th>Segment</th>
<th>Rate</th>
<th>Retry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>18 Mbps</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>12 Mbps</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>11 Mbps</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1 Mbps</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 8.1: AMRR retry chain with best rate of 18Mbps

<table>
<thead>
<tr>
<th>Segment</th>
<th>Rate</th>
<th>Retry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.5 Mbps</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2 Mbps</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1 Mbps</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>N/A</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 8.2: AMRR retry chain with best rate of 5.5Mbps
Figure 8.1: AMRR Rate Changes
<table>
<thead>
<tr>
<th>Metric</th>
<th>AMRR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use RTS/CTS</td>
<td>no</td>
</tr>
<tr>
<td>Use Sampling</td>
<td>no</td>
</tr>
<tr>
<td>Sampling Ratio</td>
<td>0</td>
</tr>
<tr>
<td>Rate Jumping</td>
<td>no</td>
</tr>
<tr>
<td>Max Retries</td>
<td>3</td>
</tr>
<tr>
<td>Retry Strategy</td>
<td>A</td>
</tr>
<tr>
<td>Max Rate Chg. Pr. sec.</td>
<td>2/2</td>
</tr>
<tr>
<td>Starting Rate</td>
<td>24, 36</td>
</tr>
<tr>
<td>Excludes Rates</td>
<td>none</td>
</tr>
<tr>
<td>Min time betw. Rate Changes</td>
<td>397ms</td>
</tr>
</tbody>
</table>

Table 8.3: AMRR Validation Statistics

**AMRR characteristics observed**

After analysing AMRR using RAC, we have discovered a few characteristics which can be used when trying to fingerprint AMRR.

We have observed that the retry chain of AMRR is as shown in Listing 8.1 and that this is what we would expect after studying the theory behind its behaviour.

Additionally, AMRR starts off transmitting at 36Mbps after encountering a reset event which is what the source code of the driver implemented in madwifi states.

AMRR is also easy to spot as it does not change its best rate by more than one rate index at a time. This means that it has to step through all possible rates when changing from the lowest to the highest rate, or the other way, changing from the highest to the lowest rate. We also observed that AMRR did at most change its current best rate every 500ms, with the exception of a few cases where rate changes seem to happen more often. This is not really important because we have uncovered and proven the basics of the behaviour of AMRR.

### 8.1.2 Analysing Onoe

Onoe is somewhat broken in madwifi 0.9.4. This prevents us from doing extensive tests on Onoe governed traffic. We have conducted a few experiments using Onoe to get the basic general behaviour as the implementation is not working properly.

The multiRate Retry chain used in Onoe is populated as shown in Table 4.4 on page 35. We see that each transmitted frame can be retransmitted at most 9 times, bringing the maximum total transmission attempts to 10. We can confirm this behaviour from the results of the validation tests by looking at the retransmission behaviour of Onoe. Listing 8.2 shows sections of the results file generated by RAC and this does show that the retry chain is identical to the one that we expect to see.

Unfortunately, since the madwifi implementation of Onoe is not
working as advertised, we are having difficulties validating other aspects of the algorithm against its theoretically expected behaviour. After running six validation tests, three stationary and three moving, we see that in the stationary tests, Onoe is stuck at one rate, thus not generating any interesting results for us to examine. Although there are losses, they are not enough to make Onoe step down to the next immediate lower rate (more than 10% of packets needing retransmission). In the moving tests, we see that Onoe steps down as expected, stepping down one rate index at a time as shown in Figure 8.2. Additionally, we observe that Onoe seems to perform a reset in situations where there are temporary failures in link connectivity, which forces the algorithm to start transmitting at its initial rate. Onoe’s initial rate is the highest negotiated rate equal to, or below 36Mbps (according to the Onoe source code). We can, as mentioned before, confirm though, that Onoe does not step rates downwards with more than one rate index at a time, which is in accordance with what we expect from the algorithm. We cannot confirm the upwards stepping since the algorithm never increases the current best rate.

We know from the source code of Onoe that the madwifi implementation of the algorithm runs its evaluation code every 1000ms. When evaluating, Onoe looks at the statistics for the previously elapsed 1000ms and analyses the credits the current best rate has gained or lost in this period. This evaluation could result in the current best rate being stepped down, remaining the same or stepping up. Either way, we should not see rate changes more frequently than every 1000ms. RAC dumps information about rate changes to the file rate_changes.dat, and we analyse this file to confirm our rate-change theory. The results can be seen in Listing 8.3 and do confirm what we expect to see.

The way RAC calculates the frequency of the rate changes is somewhat broken. It does so by looking at how many rate changes there are in the same second, e.g. two rate changes during second three from the launch of the application. To be able to create statistics about the rate change frequency, one needs to have a rolling interval of one second which moves seamlessly over all rate changes, not just looking at rate changes within one second of Unix-time. The reason for this is shown in Figure 8.3 where one actually have four rate changes during an interval of one second, although RAC will only report this as a maximum of two rate changes per second.

<table>
<thead>
<tr>
<th>1</th>
<th>1340905959.656405</th>
<th>3265</th>
<th>1</th>
<th>36000</th>
<th>36000</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1340905959.661670</td>
<td>3267</td>
<td>9</td>
<td>36000</td>
<td>36000</td>
</tr>
<tr>
<td>3</td>
<td>1340905959.700243</td>
<td>3275</td>
<td>2</td>
<td>36000</td>
<td>36000</td>
</tr>
<tr>
<td>4</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Listing 8.2:** Retry chain experiment 1, Onoe (sections of retries.dat)
Table 8.4: Onoe validation test statistics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Onoe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use RTS/CTS</td>
<td>no</td>
</tr>
<tr>
<td>Use Sampling</td>
<td>no</td>
</tr>
<tr>
<td>Sampling Ratio</td>
<td>0</td>
</tr>
<tr>
<td>Rate Jumping</td>
<td>no</td>
</tr>
<tr>
<td>Max Retries</td>
<td>9</td>
</tr>
<tr>
<td>Retry Strategy</td>
<td>A</td>
</tr>
<tr>
<td>Max Rate Chg. Pr. sec.</td>
<td>1.6/2</td>
</tr>
<tr>
<td>Starting Rate</td>
<td>24, 36</td>
</tr>
<tr>
<td>Excludes Rates</td>
<td>-</td>
</tr>
<tr>
<td>Min time betw. Rate Changes</td>
<td>485ms</td>
</tr>
</tbody>
</table>

Figure 8.3: RAC rate changes per second deficiency
Onoe characteristics observed

From our validation tests we have observed Onoe's behaviour in both stationary and moving tests. We have observed that Onoe, like AMRR, only changes rates by one rate index at a time. This is in accordance to the theory and we observed this throughout all our tests. We also observed that when the algorithm resets, it starts off using the 36Mbps rate which is set in the madwifi driver. We also observed that the rate control algorithm in Onoe does not seem to change the current best rate more often than at most once every 500ms. This is what we expect from the implementation, see Table 8.4. This table is a summary of the validation tests for Onoe. We also observed that the retry chain of Onoe is populated the way we expect it to be, using retry strategy A. There is no sampling present, nor any use of the RTS/CTS mechanism.

8.1.3 Analysing SampleRate

SampleRate is the third bundled algorithm in the madwifi driver. This algorithm is not as simple as the AMRR and the Onoe algorithms, but this does not necessarily make it harder to detect. We will start off by analysing the retry chain from a result set captured and generated by RAC, then we will analyse the sampling performed by SampleRate, after which we will look at different statistics from the result of our experiments.

Retry chain

The retry chain of SampleRate is shown in Table 4.5 on page 36, and is the result of the work in [23] written by John Bicket which was the original proposal for SampleRate. The authors of the madwifi project have changed this, and the retry chain of SampleRate is now populated as shown in Table 8.5 and is classified as the Best-rate or Base-rate strategy for retransmissions. The main difference between the changed
SampleRate retry chain and the originally proposed retry chain is the addition of one segment. Originally, the retry chain was populated using two segments, one for the current-rate and one for the base-rate, each with three attempts. The madwifi driver populates the retry chain by adding one segment before the two previously existing ones. This new segment carries either the current best rate in normal operation, or the sample rate during a sampling event. The new segment is set to retry twice, thereby increasing the original total 6 attempts to 8.

Segments from the retry chain of the SampleRate validation test are shown in Listing 8.5. From the above statements, the madwifi retry chain population is confirmed in the results from the validation test. We see that when sampling, SampleRate populates the retry chain using the picked sample rate as the first segment with two retry attempts (see lines 1, 6 and 16 in Listing 8.5). The second segment is populated with the current best rate and the third is populated with the lowest base rate, both retrying three times.

In addition, SampleRate should never exceed 7 retries per single packet transmission attempt, in total 8 transmission attempts including the original transmitted packet. We can confirm this using RAC by looking at the Max num retries variable in the statistics output, see Table 8.6.

Results from one experiment did show the maximum number of retransmissions for a single frame to be 8. Why we see 8 retry attempts once is uncertain, but we suspect this to be because of the bad state the wireless channel was in during these parts of the experiments. The 8-retry-attempt-event occurred during the low signal part of the moving experiment where the AP was furthest away from the station and the lowest base rate was used in all segments of the retry chain. Although we see one case of more than 7 retransmission attempts, the other maximum of 7 retry attempts confirms what we expected to see from the implementation of SampleRate in madwifi.

After analysing the results from the test showing a maximum of 8 retry events, the culprit is line 11 shown in Listing 8.4. We also see that immediately after capturing the 8 retry attempts from SampleRate, RAC captures a reset event, i.e. sequence numbers and rate statistics are being reset. The reason for SampleRate starting up from 36Mbps is the condition of the link at the time of transmitting the first packet after a new connection or a reset event. SampleRate operates using three initial rates, 54−, 36− or 11Mbps depending on the link RSSI. This happens when there is a long period of time with no communication between the AP and the station, basically renegotiating the link between them.

Rate exclusion

Moving on, we know that SampleRate will never use the 9Mbps physical rate, as stated in [23]. This property has been implemented in the madwifi version of SampleRate and we can see from the rate statistics
Listing 8.4: SampleRate exceeding 7 retry attempts in validation test

<table>
<thead>
<tr>
<th>Segment</th>
<th>Rate</th>
<th>Retry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>best/sample</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>current</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>base</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>N/A</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 8.5: SampleRate Retry Chain as implemented in madwifi
Listing 8.5: Retry chain experiment 1, SampleRate (sections of retries.dat)
show in Table 8.7 that 9Mbps phy-rate is not used.

Rate changes

Rate changes in SampleRate are slightly different from the previous two algorithms, AMRR and Onoe. AMRR and Onoe had the limitation of only being able to change by one rate index at the time when changing the current best rate. This is not applicable to SampleRate. SampleRate can change the current best rate from 1Mbps to 54Mbps without the need to step through every intermediate rate. We can see this behaviour when we look at the results from the validation tests. Looking at Figure 8.4 which is based on the rate change results generated by RAC, we can see that the rates do not seem to follow the stepping pattern seen in Figure 8.1 on page 74 and 8.2 on page 77.

In addition to not having the stepping pattern seen with AMRR and Onoe, SampleRate shows an increase in rate change frequency. While AMRR and Onoe did their best rate calculation at most once every 500ms, SampleRate can change its current best rate more frequently as it continuously updates its statistics whenever a packet has been transmitted, and can therefore react to changing channel conditions faster. We should therefore be able to see from the statistics of SampleRate that the current best rate can change very frequently, and with some variations between change events.

SampleRate characteristics observed

Our validation tests show that SampleRate behaves very much like we expect it to. Table 8.6 is a summary of metrics recorded for the madwifi implementation of SampleRate. We have confirmed that SampleRate does not use RTS/CTS and it samples the wireless channel with a frequency of 5.65%. This is less than expected, but as sampling events can be interpreted as rate-change events, see Section 5.3.1, we did not expect to see a sampling ratio of the implemented 10%. We have

<table>
<thead>
<tr>
<th>Metric</th>
<th>SampleRate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use RTS/CTS</td>
<td>no</td>
</tr>
<tr>
<td>Use Sampling</td>
<td>yes</td>
</tr>
<tr>
<td>Sampling Ratio</td>
<td>5.65%</td>
</tr>
<tr>
<td>Rate Jumping</td>
<td>yes</td>
</tr>
<tr>
<td>Max Retries</td>
<td>7</td>
</tr>
<tr>
<td>Retry Strategy</td>
<td>B</td>
</tr>
<tr>
<td>Max Rate Chg. Pr. sec.</td>
<td>5.17</td>
</tr>
<tr>
<td>Starting Rate</td>
<td>11, 36, 54</td>
</tr>
<tr>
<td>Excludes Rates</td>
<td>9</td>
</tr>
<tr>
<td>Min time betw. Rate Changes</td>
<td>34.8ms</td>
</tr>
</tbody>
</table>

Table 8.6: SampleRate validation test statistics
Figure 8.4: SampleRate Rate Changes
confirmed that the algorithm jumps rates, that it retries a frame at most 7 times and uses the Best-rate or Base-rate strategy. We have seen that it changes rate at most 7 times a second, although we could have seen up towards 10 times a second as SampleRate does its best rate calculation every 100ms. We see that the minimum time between rate changes is reported to be 34.8ms which is far less than the expected value of 100ms, but this can be explained by the sampling mechanism which in some cases can be interpreted as a rate-change event by RAC. This happens when three successive frames cannot be captured during a sampling event, and thus, the event is interpreted as a best-rate change event.

8.1.4 Analysing Minstrel

Minstrel is the final algorithm bundled with the madwifi driver for Atheros based wireless devices. Minstrel is one of the more complex algorithms implemented in madwifi and, like SampleRate, probes the wireless channel in order to select the best rate under changing channel conditions.

Retry chain

We will start by analysing Minstrel’s retry chain and confirm that the theoretical behaviour of Minstrel is the same as the behaviour seen by RAC during our validation experiments.

Minstrel’s method for populating the retry chain is somewhat different from SampleRate, AMRR and Onoe. While these three algorithms populate the retry chain using fixed retry values, Minstrel populates the count column using metrics observed from the wireless channel conditions. The MultiRate Retry chain population is carried out using the scheme from Table 4.7 and 4.8 on page 40. The maximum

<table>
<thead>
<tr>
<th>Rate</th>
<th>Num packets</th>
<th>Num packets</th>
<th>Num packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5500</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>6000</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>9000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11000</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>12000</td>
<td>25</td>
<td>97</td>
<td>124</td>
</tr>
<tr>
<td>18000</td>
<td>1311</td>
<td>3798</td>
<td>4432</td>
</tr>
<tr>
<td>24000</td>
<td>10142</td>
<td>16224</td>
<td>21897</td>
</tr>
<tr>
<td>36000</td>
<td>39879</td>
<td>39129</td>
<td>31147</td>
</tr>
<tr>
<td>48000</td>
<td>35111</td>
<td>33955</td>
<td>28936</td>
</tr>
<tr>
<td>54000</td>
<td>27902</td>
<td>13627</td>
<td>22476</td>
</tr>
</tbody>
</table>

Table 8.7: Rate Statistics, experiment 1, SampleRate (rates80211bg.dat aggregated, 3 runs)
number of retries per segment is calculated using the TX-time of a frame using the current segment physical rate. The number of retransmissions in a segment depends on how many transmissions can be fit in under 6ms.

**Rate changes**

Minstrel's rate changes are expected to be more like SampleRates than AMRR and Onoe. The design of the Minstrel algorithm is very much like SampleRate, still there are areas where they behave differently, thus making us able to separate the two. Figure 8.5 shows the results of the six validation experiments using the Minstrel algorithm as the RAA. We see from these figures that minstrel is more stable compared to the other algorithms, and that the transmission rates have a tendency to stabilise quickly and stay stable throughout the test. Compared to SampleRate in Figure 8.4 it is easy to see the difference between the two.

When looking at the statistics output from RAC in Table 8.8, we see that Minstrel is changing the current best rate very often compared to the other three algorithms. Additionally, the number of rate changes is far less in the moving tests than in the stationary tests, which could be explained by the access point in the stationary test being placed at such a spot where the difference in effectiveness between two neighbouring rates was negligible. Minstrel varies between the two rates as the quality of the wireless channel changes.

**Lookaround**

Minstrel's equivalent of SampleRate's sampling is referred to as “Lookaround". The authors of the Minstrel RAA observed the frequency of rates being sampled and found out that the time spent sampling rates which would perform worse than the current best rate directly affected the throughput of the wireless channel. Then they came up with a better scheme of probing the channel. As explained in 4.1.5 on page 39, we know that Minstrel only samples the wireless channel when it sees that it has something to gain from it. Minstrel will also very seldom sample down, which means that we should not see much sampling of lower rates when the current best rate is high and the channel quality is good. This is directly confirmed when looking at all the results in Figure 8.5.

Figures 8.5(b), 8.5(d) and 8.5(f) also show that when the channel quality drops and the current best rate is lowered, Minstrel starts probing the channel more aggressively compared to when channel conditions are stable. This in order to quickly adapt to the changes in channel quality and be able to rapidly recover when the channel quality increase.

Comparing Minstrel and SampleRates sampling shows differences between the algorithms when looking at the sampling events. When
comparing the Figures 8.4 and 8.5, we see the difference in sampling aggressiveness and are able to tell the difference between SampleRate and Minstrel by looking at the sampling alone. Minstrel's ability to refer from sampling rates other than the current best is easily seen when the channel conditions are good and stable unlike SampleRate which always picks a rate other than the current best and samples this every 10 transmissions.

**Minstrel characteristics observed**

After running the validation tests trying to validate Minstrel against its theoretical expected behaviour, we did manage to see many of its expected characteristics.

The population of the retry chain is easily distinguishable from the other three rate adaptation algorithms. It changes the retry count based on the rates. Minstrel adjusts every segment in the retry chain in order for each segment to finish in less than 6ms. We also see the retry chain populated not based on the immediate lower rates, as with AMRR and Onoe, but based on the constantly updated statistical analysis performed by the algorithm. This retransmission strategy is called Intelligent Selection and was presented in Section 4.2.3. We can see that the current best throughput rate, segment 1, can be the same rate as the third segment in the retry chain which is the rate with the best probability. Neither AMRR, Onoe nor SampleRate populate their retry chain this way. We see in Table 8.8 that the maximum retry attempts are 12.

Minstrel's rate changes are also distinguishable from the other algorithms, especially from AMRR and Onoe, as Minstrel, like SampleRate, can change its current rate from the lowest to the highest rate in one decision step.

Minstrel's way of probing the wireless channel, the "Lookaround" mechanism, which is somewhat different from SampleRates sampling, is very usable when trying to fingerprint Minstrel. Minstrel does

<table>
<thead>
<tr>
<th>Metric</th>
<th>Minstrel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use RTS/CTS</td>
<td>no</td>
</tr>
<tr>
<td>Use Sampling</td>
<td>yes</td>
</tr>
<tr>
<td>Sampling Ratio</td>
<td>0.23%</td>
</tr>
<tr>
<td>Rate Jumping</td>
<td>yes</td>
</tr>
<tr>
<td>Max Retries</td>
<td>12</td>
</tr>
<tr>
<td>Retry Strategy</td>
<td>C</td>
</tr>
<tr>
<td>Max Rate Chg. Pr. sec.</td>
<td>10</td>
</tr>
<tr>
<td>Starting Rate</td>
<td>&lt;= 36</td>
</tr>
<tr>
<td>Excludes Rates</td>
<td>9</td>
</tr>
<tr>
<td>Min time betw. Rate Changes</td>
<td>15.7ms</td>
</tr>
</tbody>
</table>

**Table 8.8:** Minstrel validation test statistics
<table>
<thead>
<tr>
<th>Time</th>
<th>Circuit</th>
<th>Bounce</th>
<th>Retries</th>
<th>Retries</th>
<th>Retries</th>
<th>Retries</th>
<th>Retries</th>
<th>Retries</th>
<th>Retries</th>
<th>Retries</th>
<th>Retries</th>
<th>Retries</th>
<th>Retries</th>
<th>Retries</th>
<th>Retries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1340979802.221967</td>
<td>1756</td>
<td>1</td>
<td>48000</td>
<td>48000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979802.228326</td>
<td>1764</td>
<td>3</td>
<td>48000</td>
<td>48000</td>
<td>48000</td>
<td>48000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979802.231186</td>
<td>1765</td>
<td>1</td>
<td>48000</td>
<td>48000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979802.231227</td>
<td>1767</td>
<td>11</td>
<td>48000</td>
<td>48000</td>
<td>48000</td>
<td>48000</td>
<td>54000</td>
<td>54000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979802.239078</td>
<td>1779</td>
<td>1</td>
<td>48000</td>
<td>48000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979802.264104</td>
<td>1780</td>
<td>1</td>
<td>48000</td>
<td>48000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979802.266105</td>
<td>1784</td>
<td>3</td>
<td>48000</td>
<td>48000</td>
<td>48000</td>
<td>48000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979802.268321</td>
<td>1785</td>
<td>1</td>
<td>48000</td>
<td>48000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979802.270469</td>
<td>1787</td>
<td>5</td>
<td>48000</td>
<td>48000</td>
<td>48000</td>
<td>48000</td>
<td>54000</td>
<td>54000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979802.282228</td>
<td>1804</td>
<td>1</td>
<td>48000</td>
<td>48000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979802.286293</td>
<td>1810</td>
<td>2</td>
<td>48000</td>
<td>48000</td>
<td>48000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979802.286326</td>
<td>1811</td>
<td>4</td>
<td>48000</td>
<td>48000</td>
<td>48000</td>
<td>48000</td>
<td>36000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979802.291450</td>
<td>1813</td>
<td>2</td>
<td>48000</td>
<td>48000</td>
<td>48000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979802.291508</td>
<td>1815</td>
<td>1</td>
<td>48000</td>
<td>48000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979802.293824</td>
<td>1816</td>
<td>2</td>
<td>48000</td>
<td>48000</td>
<td>48000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979802.293864</td>
<td>1817</td>
<td>2</td>
<td>48000</td>
<td>48000</td>
<td>48000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979810.34712</td>
<td>3276</td>
<td>3</td>
<td>54000</td>
<td>54000</td>
<td>54000</td>
<td>54000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979810.350840</td>
<td>3277</td>
<td>1</td>
<td>54000</td>
<td>54000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979810.350864</td>
<td>3278</td>
<td>12</td>
<td>54000</td>
<td>54000</td>
<td>54000</td>
<td>54000</td>
<td>48000</td>
<td>48000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979810.370260</td>
<td>3279</td>
<td>6</td>
<td>54000</td>
<td>54000</td>
<td>54000</td>
<td>54000</td>
<td>48000</td>
<td>48000</td>
<td>48000</td>
<td>48000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979810.377925</td>
<td>3288</td>
<td>2</td>
<td>54000</td>
<td>54000</td>
<td>54000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340979810.382260</td>
<td>3292</td>
<td>1</td>
<td>54000</td>
<td>54000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340980985.027925</td>
<td>2668</td>
<td>3</td>
<td>54000</td>
<td>54000</td>
<td>54000</td>
<td>54000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340980985.035258</td>
<td>2679</td>
<td>1</td>
<td>54000</td>
<td>54000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340980985.037278</td>
<td>2680</td>
<td>1</td>
<td>54000</td>
<td>54000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340980985.041373</td>
<td>2689</td>
<td>2</td>
<td>54000</td>
<td>54000</td>
<td>54000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1340980985.041410</td>
<td>2691</td>
<td>12</td>
<td>54000</td>
<td>54000</td>
<td>54000</td>
<td>54000</td>
<td>48000</td>
<td>48000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Listing 8.6:** Retry chain experiment 1, Minstrel (sections of retries.dat)
Figure 8.5: Minstrel Rate Changes
not probe the wireless channel as aggressively as SampleRate, and it
does very seldom probe down from a high rate as this would lower
the throughput of the algorithm. We see from our results that the
“Lookaround” mechanism is easy to spot when comparing the results
from Minstrel and SampleRate.

Table 8.8 shows a summary of the characteristics observed. We see
that the algorithm has the ability to change best rate up to 10 times
per second. The minimum time between rate changes is very short, but
this, as with SampleRate, can be explained by RACs inability to reliably
distinguish a rate change event from a sample event when there are
losses in the capture process. We see that the sample ratio is much lower
than that of SampleRate, which is expected as Minstrel implements a
more intelligent sampling scheme.

8.2 Measured RAA characteristics

This section presents the results of the measured characteristics of
madwifi RAA algorithms.

8.2.1 Madwifi

After running the experiments, we have gathered some statistics
from the Madwifi implementation of AMRR, Onoe, SampleRate and
Minstrel. This section present our findings with regards to the
measured characteristics of the RAA.

We see from table 8.9 that the RAAs are different in many ways.
When using RAC, one can look at this table and is able to recognize the
algorithms even when not knowing the physical hardware or the driver
implemented.

<table>
<thead>
<tr>
<th>Metric</th>
<th>AMRR</th>
<th>Onoe</th>
<th>SampleRate</th>
<th>Minstrel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use RTS/CTS</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Use Sampling</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Sampling Ratio</td>
<td>0</td>
<td>0</td>
<td>5.65%</td>
<td>0.23%</td>
</tr>
<tr>
<td>Rate Jumping</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Max Retries</td>
<td>3</td>
<td>9</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>Retry Strategy</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>Max Rate Chg. Pr. sec.</td>
<td>2/2</td>
<td>1.6/2</td>
<td>5.1/7</td>
<td>8.0/10</td>
</tr>
<tr>
<td>Starting Rate</td>
<td>24,36</td>
<td>24,36</td>
<td>11,36,54</td>
<td>&lt;= 36</td>
</tr>
<tr>
<td>Excludes Rates</td>
<td>-</td>
<td>-</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Min time betw. Rate Changes</td>
<td>397ms</td>
<td>485ms</td>
<td>34.8ms</td>
<td>15.7ms</td>
</tr>
</tbody>
</table>

Table 8.9: RAA Comparison Table (Madwifi)
8.3 Comparing Rate Adaptation Algorithm Behaviour

After running validation tests for the madwifi implemented algorithms, we ran a few tests using a variation of different wireless hardware. The hardware used were mostly laptop computers borrowed from fellow students. One test were also conducted using an Android mobile phone. Some with known algorithms, some not. The wireless hardware used in these tests is shown in Table 8.10. The tests were ran in the same manner as the validation tests, with three stationary tests and three moving tests. The outputs from RAC were averaged over all tests when looking at sampling ratio, max rate change pr. second and minimum time between rate changes. Rate jumping was decided by looking at the statistics output over all tests, and the exclude 9Mbps metric was decided by looking at the rate transmit distribution. The retry strategy were decided by looking at the retry output from RAC.

What we learn from these tests is that there are many differences between drivers and implementations. Even the Minstrel implementations in the mac80211 framework has major differences from the Minstrel algorithm implemented in madwifi. For example, we see that, in Table 8.11, the Minstrel algorithm in mac80211 does not use the Intelligent Selection strategy when retransmitting frames. In the two tests where the Minstrel algorithm from the mac80211 framework was used, we see that the retry strategies are Best-rate or Base-rate (B). The Minstrel HT algorithm implemented in mac80211, however, is using the Intelligent Selection (C) Strategy as expected in all Minstrel implementations. The retry strategies were explained in Section 4.2.

We do see that sampling is a popular mechanism in RAAs. Most of the drivers are using sampling, although not as much as the SampleRate implementation in madwifi. The SampleRate implementation in madwifi has a sampling ratio of 5.65% while the Minstrel implementation only samples 0.23% of the time. There are no results where the sampling ratio even comes close to that of SampleRate. We see in Table 8.11 that the most frequent samplers are the Minstrel HT algorithm from the mac80211 framework, followed closely by Minstrel, also from the mac80211 framework.

The maximum number of retransmission for a frame is varying. We see that the Intel drivers are all having a maximum of 15 retransmissions as well as the same retransmit strategy, the Simple Fallback.

Rate Adaptation Algorithm Unfortunately, we cannot say much about the RAA used in these devices. The only RAA we can determine using what we know about these algorithms is the mac80211 implementation of the Minstrel HT. We see in Table 8.11 that we have a good match against what we know about Minstrel from the
Atheros AR5001X+ results. These results tell us that the sampling matches what we know from tests run on the madwifi implementation of Minstrel. We also see that the retry strategy is the one we expect. It jumps rates and the number of rate changes per second is 10. The expected number of rate changes per second is 8 from the madwifi implementation, but the source code of the mac80211 framework changes this so the Minstrel implementation can change the best rate every 100ms.

From the results shown in Table 8.11 we can conclude that none of the devices are running AMRR or Onoe because of the rate jumping seen in all tests. Furthermore, we see that the sampling ratio is very low which we know does not happen in SampleRate as well as we see the Broadcom BCM4318 [AirForce One 54g] devices use 9Mbps. The max retry results tells us that neither AMRR nor Onoe are used, although both SampleRate and Minstrel are likely candidates.

We can from this conclude that both SampleRate and Minstrel are likely candidates for most if not all of the devices. Minstrel is in most cases the most likely candidate, although the retry strategy is not always the expected one.
<table>
<thead>
<tr>
<th>Hardware</th>
<th>MAC80211</th>
<th>Driver module</th>
<th>version</th>
<th>Operating system</th>
<th>RAA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atheros AR5001X+</td>
<td>minstrel_ht</td>
<td>ath9k</td>
<td>in-tree</td>
<td>Fedora 14 (3.4.0)</td>
<td>Minstrel HT</td>
</tr>
<tr>
<td>RaLink RT2860</td>
<td>-</td>
<td>rt2860sta</td>
<td>1.8.1.1</td>
<td>Ubuntu 10.04</td>
<td>Unknown</td>
</tr>
<tr>
<td>LAN-Express AS IEEE 80211g miniPCI Adapter</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Windows XP</td>
<td>Unknown</td>
</tr>
<tr>
<td>Intel WiFi Link 5100</td>
<td>iwl-agn-rs</td>
<td>iwlwifi</td>
<td>in-tree</td>
<td>Ubuntu 12.04</td>
<td>Unknown</td>
</tr>
<tr>
<td>Atheros AR922X</td>
<td>ath9k_rate_control</td>
<td>ath9k</td>
<td>in-tree</td>
<td>Fedora 14 (3.4.0)</td>
<td>Unknown</td>
</tr>
<tr>
<td>Intel 3945ABG #1</td>
<td>iwl-3945-rs</td>
<td>iwl3945</td>
<td>in-tree</td>
<td>Ubuntu 12.04</td>
<td>Unknown</td>
</tr>
<tr>
<td>Intel 3945ABG #2</td>
<td>iwl-3945-rs</td>
<td>iwl3945</td>
<td>in-tree</td>
<td>Ubuntu 11.10</td>
<td>Unknown</td>
</tr>
<tr>
<td>Broadcom BCM4311 #1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Ubuntu 10.04</td>
<td>Unknown</td>
</tr>
<tr>
<td>Broadcom BCM4311 #2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Windows XP</td>
<td>Unknown</td>
</tr>
<tr>
<td>Broadcom BCM4318 [AirForce One 54g] #1</td>
<td>minstrel</td>
<td>b43</td>
<td>-</td>
<td>Ubuntu 10.04</td>
<td>Minstrel</td>
</tr>
<tr>
<td>Broadcom BCM4318 [AirForce One 54g] #2</td>
<td>minstrel</td>
<td>b43</td>
<td>in-tree</td>
<td>Fedora 14 (2.6.35)</td>
<td>Minstrel</td>
</tr>
<tr>
<td>Samsung i9300</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Android 4.0.4</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

Table 8.10: RAA Test Hardware
<table>
<thead>
<tr>
<th>Hardware</th>
<th>Sampling Ratio (%)</th>
<th>Max retries</th>
<th>Rate jumping</th>
<th>Max changes pr. sec</th>
<th>Excludes 9Mbps</th>
<th>Min. time betw. rate chg.</th>
<th>Retry strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atheros AR5001X+</td>
<td>0.76</td>
<td>9</td>
<td>yes</td>
<td>10.5</td>
<td>yes</td>
<td>-</td>
<td>C</td>
</tr>
<tr>
<td>RaLink RT2860</td>
<td>0.02</td>
<td>15</td>
<td>yes</td>
<td>3.1</td>
<td>yes</td>
<td>0.196</td>
<td>A</td>
</tr>
<tr>
<td>LAN-Express AS IEEE 80211g miniPCI Adapter</td>
<td>0.09</td>
<td>14</td>
<td>yes</td>
<td>12.7</td>
<td>yes</td>
<td>0.002</td>
<td>A</td>
</tr>
<tr>
<td>Intel WiFi Link 5100</td>
<td>0.03</td>
<td>15</td>
<td>yes</td>
<td>26.5</td>
<td>yes</td>
<td>0.0015</td>
<td>A</td>
</tr>
<tr>
<td>Atheros AR922X</td>
<td>0.29</td>
<td>9</td>
<td>yes</td>
<td>41.8</td>
<td>yes</td>
<td>0.000015</td>
<td>A</td>
</tr>
<tr>
<td>Intel 3945ABG #1</td>
<td>0.02</td>
<td>15</td>
<td>yes</td>
<td>7.1</td>
<td>yes</td>
<td>0.016</td>
<td>A</td>
</tr>
<tr>
<td>Intel 3945ABG #2</td>
<td>0.0</td>
<td>15</td>
<td>yes</td>
<td>9.1</td>
<td>yes</td>
<td>0.005</td>
<td>A</td>
</tr>
<tr>
<td>Broadcom BCM4311 #1</td>
<td>0.06</td>
<td>6</td>
<td>yes</td>
<td>2</td>
<td>yes</td>
<td>-</td>
<td>B</td>
</tr>
<tr>
<td>Broadcom BCM4311 #2</td>
<td>0.0</td>
<td>6</td>
<td>yes</td>
<td>14.3</td>
<td>yes</td>
<td>0.011</td>
<td>C</td>
</tr>
<tr>
<td>Broadcom BCM4318 [AirForce One 54g] #1</td>
<td>0.44</td>
<td>7</td>
<td>yes</td>
<td>6.3</td>
<td>no</td>
<td>0.043</td>
<td>B</td>
</tr>
<tr>
<td>Broadcom BCM4318 [AirForce One 54g] #2</td>
<td>0.65</td>
<td>6</td>
<td>yes</td>
<td>4.4</td>
<td>no</td>
<td>0.241</td>
<td>B</td>
</tr>
<tr>
<td>Samsung i9300</td>
<td>0.0</td>
<td>6</td>
<td>yes</td>
<td>10.8</td>
<td>yes</td>
<td>0.008</td>
<td>C</td>
</tr>
</tbody>
</table>

**Table 8.11: RAA Test Results**
Part VI

Conclusion and future work
This thesis presented the Rate Adaptation Classifier (RAC) tool and a method to assist in classifying rate adaptation algorithms used in IEEE 802.11 wireless network devices. The tool enables network professionals and researchers to capture network traffic from specific devices, enabling later analysis of the output for classifying the rate adaptation algorithm used. The RAAs present in the madwifi driver is AMRR, Onoe, SampleRate and Minstrel. There exists other RAAs in other drivers, but this thesis have focused on the ones present in madwifi. The tool and accompanying method were successfully able to detect algorithms used in the madwifi driver. We looked at other devices, but they did not show the behaviour of the madwifi RAAs.

The tool is capable of capturing and storing information which enables us to distinguish different rate adaptation algorithms. This requires there to be a classification process beforehand. Once a driver has been classified and the metrics presented in this thesis have been defined, the tool and accompanying method can be applied to determine the rate adaptation method used.

The RAC tool and assisting method for determining RAAs were chosen because of the power it gives to the user. RAC may not be the best solution in terms of flexibility, but the details it provides to the user about the observed traffic is very precise.

We introduced retry strategy classifications: Simple Fallback, Best-rate or Base-rate and Intelligent Selection. We have seen that every device we have tested falls into one of these strategies.

A considerable amount of work was done researching and studying the RAAs implemented in the madwifi driver as well as the original design and implementation of these algorithms. We have looked at other works trying to fingerprint RAAs using other methods, but we decided that the manual method of RAC was most precise. We have studied the IEEE 802.11 wireless network standard to be able to implement the parsers required by RAC. Designing and implementing the RAC tool with the accompanying method for determining RAAs was done incrementally throughout the thesis timespan. The RAC tool was written in C because of its speed and access to libraries such as libpcap.

The thesis presented experiments conducted to determine the correctness and effectiveness of RAC and accompanying method. Captured data from the RAC tool was compared to the captured data from Wireshark, and this did show that the capture engine of RAC is working as expected. The RAC tool captures and interprets wireless data correctly. Experiments conducted on IEEE 802.11 hardware running the madwifi driver did show that RAC and accompanying method is able to determine the drivers RAA. Experiments which was run on wireless devices where the RAA was unknown or the implementation was different than that of madwifi did show different metric values than the madwifi implementation. We could therefore not map any of these algorithms to the ones implemented in the madwifi driver.
Future work

This thesis presented the statistics and results from experiments ran on four RAAs implemented in the madwifi driver: AMRR, Onoe, SampleRate and Minstrel. We did a comprehensive study on the behaviour of these algorithms and presented the results from our experiments. We have seen that these implementations are recognized by RAC and accompanying method. For this to work with other algorithms and implementations, there need to be a study of algorithms and mapping the observed metric values for later use.

In the future, the RAC tool and accompanying method for classifying RAAs can be merged together into one single tool. To accomplish this, the classification – and statistics engine of RAC has to be extended with knowledge on the behaviour of the different RAAs. RAC needs to be able to perform the mapping between the observed traffic patterns and the defined metrics of an RAA. As of now, this process is performed manually by the user.
Bibliography


the 7th annual international conference on Mobile computing and networking, MobiCom ’01, pages 236–251, New York, NY, USA, 2001. ACM.


Abbreviations

AARF  Adaptive Auto Rate Fallback.
ACK   Acknowledgement.
AMRR  Adaptive MultiRate Retry.
AP    Access Point.
ARF   Auto Rate Fallback.
BEB   Binary Exponential Backoff.
BSS   Basic Service Set.
CD    Derivative Coefficient.
CI    Integral Coefficient.
CP    Proportional Coefficient.
CSMA/CA Carrier Sense Multiple Access/Collision Avoidance.
CSMA/CD Carrier Sense Multiple Access/Collision Detection.
CTS   Clear To Send.
DSSS  Direct Sequence Spread Spectrum.
FHSS  Frequency Hopping Spread Spectrum.
HAL   Hardware Abstraction Layer.
HR/DSSS High Rate Direct Sequence Spread Spectrum.
MAC   Media Access Control.
MRR   Multi-Rate Retry.
NACK  Negative Acknowledgement.
OFDM  Orthogonal Frequency Division Multiplexing.
PC    Point Controller.
PHY   Physical Layer.
PID   Proportional–Integral–Derivative.
PLCP  Physical Layer Convergence Protocol.
PPDU  PLCP Protocol Data Unit.
PSDU  Physical layer Service Data Unit.

RAA   Rate Adaptation Algorithm.
RAC   Rate Adaptation Classifier.
RBAR  Receiver-Based AutoRate.
RCA   Rate Control Algorithm.
RSA   Rate Selection Algorithm.
RTS   Request To Send.

SNR   Signal-To-Noise Ratio.
WLAN  Wireless Local Area Network.