INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

IEEE 802.11 Rate Adaptation: A Practical Approach

Mathieu Lacage — Hossein Manshaei — Thierry Turletti

N° 5208

Mai 2004

Thème COM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--5208--FR+ENG

IEEE 802.11 Rate Adaptation: A Practical Approach

Mathieu Lacage* , Hossein Manshaei* , Thierry Turletti*

Thème COM — Systèmes communicants
Projet Planète

Rapport de recherche n° 5208 — Mai 2004 — 25 pages

Abstract: Today, three different physical (PHY) layers for the IEEE 802.11 WLAN are available (802.11a/b/g); they all provide multi-rate capabilities. To achieve high performance under varying conditions, these devices need to adapt their transmission rate dynamically. While this rate adaptation algorithm is a critical component of their performance, only very few algorithms such as Auto Rate Fallback (ARF) or Receiver Based Auto Rate (RBAR) have been published and the implementation challenges associated with these mechanisms have never been publicly discussed. In this paper, we first present the important characteristics of the 802.11 systems that must be taken into account when such algorithms are designed. Specifically, we emphasize the contrast between *low latency* and *high latency* systems, and we give examples of actual chipsets that fall in either of the different categories. We propose an Adaptive ARF (AARF) algorithm for low latency systems that improves upon ARF to provide both short-term and long-term adaptation. The new algorithm has very low complexity while obtaining a performance similar to RBAR, which requires incompatible changes to the 802.11 MAC and PHY protocol. Finally, we present a new rate adaptation algorithm designed for high latency systems that has been implemented and evaluated on an AR5212-based device. Experimentation results show a clear performance improvement over the algorithm previously implemented in the AR5212 driver we used.

Key-words: Rate Adaptation, 802.11, multi-rate, ARF, RBAR, AARF, AMRR

* INRIA, Planete Project, 2004 route des Lucioles, BP 93 FR-06902 Sophia Antipolis, {lacage,hmanshae,turletti}@sophia.inria.fr

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Une approche pratique au problème de l'adaptation de mode pour le standard IEEE 802.11

Résumé : Les trois couches physiques IEEE 802.11a/b/g disponibles aujourd'hui offrent toutes des services multi-modes. Afin d'obtenir des performances élevées dans des conditions de transmission variables, les systèmes qui utilisent ces couches physiques doivent continuellement changer le mode de transmission utilisé pour maximiser le débit disponible au niveau applicatif. Bien que les algorithmes de sélection de mode constituent un composant important de la performance de ces réseaux locaux sans fil, aucun algorithme à l'exception de *ARF* (*Auto Rate Fallback*) et de *RBAR* (*Receiver Based Auto Rate*) n'a été publié et les problèmes fondamentaux liés à l'implémentation de tels algorithmes n'ont jamais été pris en considération dans des articles publiés. Nous présentons donc le paramètre de ces systèmes 802.11 qui a la plus grande influence sur la conception d'algorithmes de contrôle de mode de transmission et nous identifions deux types de systèmes: les systèmes à *faible latence* et ceux à *forte latence* de communication entre les couches physiques et les couches MACs. Puis, nous proposons deux nouveaux algorithmes de contrôle de mode de transmission dont les performances approchent l'optimum représenté par *RBAR* et qui sont, contrairement à *RBAR*, réalisables aujourd'hui sans modifications incompatibles du standard 802.11. *AARF* (*Adaptive ARF*) et *AMRR* (*Adaptive Multi Rate Retry*), conçus respectivement pour les système dis à *faible latence* et *forte latence*, ont été implementés et simulés à l'aide de ns. Nous avons aussi évalué une implémentation de *AMRR* dans un driver Linux pour cartes 802.11 basées sur le chipset AR5212 qui confirme les résultats de simulation et montre une amélioration importante du débit obtenu au niveau applicatif.

Mots-clés : Adaptation de Mode, 802.11, multi-mode, AARF, AMRR, ARF, RBAR

1 Introduction

IEEE 802.11 is the most popular WLAN system in the world today and it is likely to play an important role in the next generation of wireless and mobile communication systems. Originally, IEEE 802.11 DSSS (Direct Sequence Spread Spectrum) offered only two physical data rates: all transmission was done at either the 1Mbps or the 2Mbps rate. In 1999, the IEEE defined two high rate extensions: 802.11b based on DSSS technology, with data rates up to 11Mbps in the 2.4GHz band, and 802.11a, based on OFDM (Orthogonal Frequency Division Multiplexing) technology, with data rates up to 54 Mbps in the 5GHz band. In 2003, the 802.11g standard that extends the 802.11b PHY layer to support data rates up to 54 Mbps in the 2.4 GHz band was finalized.

There are many reasons for the highly volatile nature of the wireless medium used by the IEEE 802.11 standard: fading, attenuation, interference from other radiation sources, interference from other 802.11 devices in an ad hoc network, etc. We can classify these transmission quality variations as either transient short-term modifications to the wireless medium or durable long-term modifications to the transmission environment.

Typically, if someone walks around, closes a door, or moves big objects, this will have an effect on the transmission medium for a very short time. Its throughput capacity might drop sharply but not for long. If one decides to move to another office, thus approaching the AP (Access Point), the attenuation will decrease and this will have a longer lasting effect on the energy of the radio signal that will probably decrease the BER (Bit Error Rate). This, in turn, will allow higher application-level throughput since the PER (Packet Error Rate) is lower.

Algorithms that adapt the transmission parameters to the channel conditions can be designed to optimize a number of parameters depending on the network topology and the type of device:

- Power consumption: mobile devices which implement 802.11 radios usually have a fixed energy budget (due to finite battery life). As such, it is of utter importance to minimize the amount of energy consumed by their private 802.11 radios.
- Throughput: the higher 802.11 transmission rates often provide important potential throughput but they usually have higher BERs. Higher BERs require more retransmissions for error-free transmissions, which decreases the application-level throughput.

In this paper, we focus on the task of maximizing the application-level throughput in infrastructure networks through practical rate-adaptation algorithms. Because no published paper discusses the issues surrounding real implementations of 802.11 rate adaptation algorithms, we believe our main contribution to be the identification of two classes of 802.11 devices: *low latency* and *high latency* systems. *low latency* systems allow the implementation of per-packet adaptation algorithms while *high latency* systems require periodic analysis of the transmission characteristics and updates to the transmission parameters.

Our second contribution is the proposal of two simple novel algorithms each designed for one of the two classes of devices identified. Their performance is close to the optimum represented by the impractical RBAR[3] in the case of infrastructure networks. We present experimental results that show that our new algorithms can be readily implemented on existing hardware and offer considerable performance improvements over existing solutions.

This paper is organized as follows. In Section 2, we present the existing algorithms that try to address the task of rate adaptation and their shortcomings. Then, we identify in Section 3 the key architectural feature of a 802.11 system that must be taken into account when designing a rate adaptation algorithm: namely, the distinction between low and high communication latency between the radio baseband and the block that implements the rate control algorithm. We describe in Section 4 the AARF algorithm designed for low communication latency systems that is based on per-packet short-term adaptation but introduces a specific long-term adaptation mechanism to improve the application level throughput.

In Section 5, we present a new rate control algorithm named Adaptive Multi Rate Retry (AMRR) based on the same ideas developed for *AARF* that has been designed, implemented, and evaluated on a high communication latency system based on an Atheros AR5212 chipset. The experimental measurements obtained confirm the simulation results and offer convincing evidence that our *AMRR* algorithm achieves higher performance than the previously implemented algorithm. Finally, we present a summary of our work and future research directions in Section 6.

2 Related Work

2.1 ARF

ARF [2] was the first rate adaptation algorithm to be published. It was designed to optimize the application throughput in WaveLan II devices, which implemented the 802.11 DSSS standard[1]. In ARF, each sender attempts to use a higher transmission rate after a fixed number of successful transmissions at a given rate and switches back to a lower rate after 1 or 2 consecutive failures. Specifically, the original ARF algorithm decreases the current rate and starts a timer when two consecutive transmissions fail in a row. When either the timer expires or the number of successfully received per-packet acknowledgements reaches 10, the transmission rate is increased to a higher data rate and the timer is reset. When the rate is increased, the first transmission after the rate increase (commonly referred to as the *probing* transmission or *probing* packet) must succeed or the rate is immediately decreased and the timer is restarted rather than trying the higher rate a second time. This scheme suffers from two problems:

- If the channel conditions change very quickly, it cannot adapt effectively. For example, in an ad hoc network where the interference bursts are generated by other 802.11 packet transmissions, the optimum rate changes from one packet to the next. Because ARF requires 1 or 2 packet failures to decrease its rate and up to 10 successful packet

[1] Rumors claim that ARF has been used in Agere and Intersil Prism II 802.11b devices but it is hard to come up with any meaningful information since this is considered sensitive intellectual property.

transmissions to increase it, it will never be synchronized with the sub-packet channel condition changes.

- If the channel conditions do not change at all, or change very slowly, it will try to use a higher rate every 10 successfully transmitted packets; this results in increased retransmission attempts and thus decreases the application throughput.

2.2 RBAR

RBAR [3] is the only other published rate adaptation algorithm whose goal is to optimize the application throughput. This algorithm requires incompatible changes to the IEEE 802.11 standard: the interpretation of some MAC control frames is changed and each data frame must include a new header field. While this algorithm is of little practical interest because it cannot be deployed in existing 802.11 networks, it is of important theoretical interest because it can be used as a performance reference.

The RBAR algorithm mandates the use of the RTS/CTS mechanism: a pair of Request To Send/Clear To Send control frames are exchanged between the source and the destination nodes prior to the start of each data transmission. The receiver of the RTS frame calculates the transmission rate to be used by the upcoming data frame transmission based on the Signal To Noise Ratio (SNR) of the received RTS frame and on a set of SNR thresholds calculated with an *a priori* wireless channel model. The rate to use is then sent back to the source in the CTS packet. The RTS, CTS, and data frames are modified to contain information on the size and rate of the data transmission to allow all the nodes within transmission range to correctly update their Network Allocation Vector (NAV).

This protocol suffers from numerous flaws that are summarized below:

- The threshold mechanism used in each receiver to pick the best possible rate requires a calculation of the SNR thresholds based on an *a priori* channel model.
- The algorithm assumes that the SNR of a given packet is available at the receiver, which is not generally true: some (but not all) 802.11 devices provide an estimation of the SNR by measuring the energy level prior to the start of the reception of a packet and during the reception of the packet.
- The RTS/CTS protocol is required even though no hidden nodes are present that can lead to a major performance hit.
- The interpretation of the RTS and CTS frames and the format of the data frames is not compatible with the 802.11 standard. Thus, RBAR cannot be deployed in existing 802.11 networks.

2.3 MiSer

MiSer [1] is an algorithm based on the 802.11a and 802.11h[2] standards whose goal is to optimize the local power consumption (and not the application throughput which is our stated goal). To do so, it adapts both the transmission rate and the transmission power. The set of optimal rate/transmission power pairs is calculated offline with a specific wireless channel model. At runtime, the wireless nodes execute simple table lookups to pick the optimum rate/transmission power combination.

The main problems with this algorithm (other than mandating the use of the RTS/CTS protocol) are twofold:

- It requires the choice of an *a priori* wireless channel model for the offline table calculation.
- It requires *a priori* knowledge of the number of contending stations on the wireless network.

3 Low and high communication latency systems

All the 802.11 systems contain at least the following two subsystems:

- The 802.11 radio: this integrates the modulation, demodulation, encoding, decoding, ADC, DAC, and filtering functions. These functions are always entirely implemented in hardware; they correspond to the 802.11 PHY layer.
- The 802.11 MAC layer: this is always implemented by a combination of dedicated hardware and dedicated software. The exact split between these two domains is entirely device specific.

The rate control algorithms we are interested in are part of the MAC layer; their function is to choose the rate to be used for each packet that is sent to the PHY layer. The exact architecture of the MAC layer, i.e., how much of the MAC layer is implemented in hardware, varies a lot from one device to another. Therefore, it is very hard to design a device independent rate control mechanism. However, it is clear that the *communication latency* between the PHY layer and the block that implements the rate control algorithm within the MAC layer is one of the most important parameters to take into account when designing the algorithm. If the communication latency is low, it is possible to devise a rate control algorithm that implements per-packet adaptation. If it is higher than the threshold described in the following section, more complex methods must be employed. Actual examples of both types of devices are presented in the two other following sections.

[2] 802.11h is an extension of the current 802.11 MAC and the high-speed 802.11a PHY, to implement an intelligent transmission power control.

3.1 Communication latency requirements

Low latency systems allow us to implement per-packet adaptation. This means that for each packet sent, we must get feedback information on the transmission status of this packet before sending the next packet. As such, we calculate below the minimum time interval between two successive packet transmissions during a fragment burst ($T_{fragment}$), when a packet transmission fails ($T_{failure}$) and when a packet transmission succeeds ($T_{success}$).

In the case of transmission failure because the required ACK has not been received, the sending device starts a backoff procedure after a $DIFS$ at the end of $ACKTimeout$. The transmission of other packets by this device does not start until the end of the backoff procedure which cannot happen until $T_{failure} = ACKTimeout + DIFS + aCW_{random} \times aSlotTime$ after the end of the transmission (aCW_{random} is a uniformly distributed variable between zero and aCW where $aCW_{min} < aCW < aCW_{max}$).

In the case of successful transmission, the sending device either starts a $SIFS$ timer if it wants to keep on bursting the following MAC fragments ($T_{fragment} = SIFS$) or it starts a backoff procedure after a $DIFS$ if it wants to transmit another packet that is not a MAC fragment ($T_{success} = DIFS + aCW_{random} \times aSlotTime$).

Because we are interested in the worst case scenario (that is, the minimum time interval between two transmissions), we assume that $aCW_{random} = 0$ which gives $T_{success} = DIFS$ and $T_{failure} = ACKTimeout + DIFS$. Since $SIFS < DIFS$, we have: $T_{fragment} < T_{success} < T_{failure}$. The values of these parameters are shown in Table 1. If we do not allow the user to use a different rate for each fragment of a burst (which is a very reasonable assumption in practice since no newer information is available for the fragment), the minimum latency requirement for all the values presented in Table 1 column $T_{success}$ is 28 μs.

Table 1: Communication latency constraints in the IEEE 802.11 standards

Standard	$T_{fragment}$	$T_{success}$
802.11 DSSS	10 μs	50 μs
802.11a	16 μs	34 μs
802.11b	10 μs	50 μs
802.11g	10 μs	28 μs

Any multi-standard system where the two-way communication latency between the PHY layer (where the transmission status is known) and the rate-control algorithm (where the information on the transmission status is acted upon) is higher than 28 μs cannot implement per-packet rate adaptation.

3.2 Low latency systems

The pressure to achieve short "Time To Market" schedules has led a lot of WLAN designers to move a lot of functionality into software. The hardware is often reduced to the bare minimum and a small embedded CPU (a dedicated home-made RISC processor or an ARM or MIPS core) is integrated in the 802.11 chip that controls all the time-critical tasks (including the rate-adaptation). The WaveLAN 802.11b Chipset[9] follows this trend. It is based on a three chip architecture:

- The monomode WaveLAN WL60010 MAC whose architecture is similar to the multi-mode WL60040 MAC[10], i.e., a MAC controller which interfaces with the host system (through a PCI or a USB bus), and the baseband controller,
- The WaveLAN N4080 802.11 Baseband Controller, and
- The WaveLAN W4050 RF Transceiver.

The WL60010 MAC controls the baseband controller through a specialized communication bus that we can assume to be low latency since it is not shared with any other device. It contains an embedded processor that executes the firmware provided by Agere. This firmware has access both to the baseband controller and to the host through a DMA Bus Master block. The chipset product brief notes that the firmware implements a "*Fallback rate algorithm that sets the optimum rate based on actual signal to noise ratio and packet loss information.*" While this is marketing material and does not constitute technical documentation, it seems clear to us that the architecture of the WL60010 MAC (namely, the presence of an embedded processor and of a dedicated communication bus with the baseband controller) allows specialized firmware to implement per-packet adaptation.

A lot of other chipsets (for example, all the chipsets designed by Texas Instruments around an ARM core: *ACX100*, *TNETW1100B*, *TNETW1130*, and *TNETW1230*) seem to be designed with similar architectures and thus probably provide enough flexibility to firmware designers to implement various rate control algorithms.

Some of these chipsets can be used in standalone AP designs (this means that they can implement all the AP functions on the embedded processor). However, a lot of them always require a host system with a host CPU that handles the higher-layer networking protocols and that runs the user applications when it is located in a 802.11 terminal.

3.3 High latency systems

Because the market for 802.11 devices is growing very quickly, a lot of pressure has been put on chip designers to come up with designs that provide cost-effective ways to add 802.11 STA capability to terminals. High volume electronic systems always try to minimize the number of discrete components used[3] and the cost of each component. To keep the number of discrete components down, the only solution is to integrate multiple functions on each component. For example, TI's TNETW1100B integrates the MAC controller and the radio baseband on a single device.

The cost of each chipset component is directly related to the surface of its silicium die: the larger the surface, the higher the cost. This is partly due to the fact that the number of

[3] Less components means less costly boards and less costly board testing.

dies on a single silicium wafer is obviously related to the area occupied by each die on the wafer but it is also due to the decrease of die yield. Typically, the probability of chip failure increases with its surface.

The two conflicting requirements described above are usually reconciled by removing the less useful components, while more important components are integrated together. For example, the AR5212 chip [14] which is part of many Atheros 802.11 chipsets integrates a MAC and a baseband controller. In this chipset, the MAC controller does not contain an embedded processor. Some time-critical functions, are left in the MAC controller but less critical functions usually handled by the firmware running on the traditionally integrated RISC CPU, are now shared between the MAC controller and the host system. Specifically, rate adaptation is now shared between the host CPU and its MAC controller, this makes the communication latency between these two devices very important.

While it is possible to design a host system such that the communication latency between the host CPU and the AR5212 PCI device is extremely small (small enough to support per-packet adaptation), none of the terminals to which the device will be connected in practice are designed that way. Only a few APs where the system architecture is entirely under control can achieve such low latency requirements.

Typically, on desktop PCs or laptops, which run a generic OS with generic applications and which contain arbitrary peripheral devices, it is not possible to state upper bounds on the device/host communication latency. While the hardware component of the communication latency cannot be ignored (typically, the time required to acquire access to the different communication path components between the device and the host CPU or system memory), the most important factor is the software part. As detailed in [13], Real Time Operating Systems can easily achieve as low as 3 to $4\mu s$ interrupt latency minimum with 50 to $60\mu s$ maximum but generic operating systems sometimes reach up to $0.1s$ interrupt latency.

4 The Adaptive ARF algorithm

4.1 Motivations

ARF was designed for a low-latency system based on the second generation of WaveLAN devices. While it is reasonably good at handling the short-term variations of the wireless medium characteristics in an infrastructure network, it fails to handle stable conditions that are the norm efficiently. Typically, office workers setup their laptop, sit in a chair or at their desk and work there for a few hours. They rarely walk around while typing on their computer keyboard!

In this environment, the *best* rate to choose to optimize the application throughput is the highest rate whose PER is low enough such that the number of retransmissions is low. Typically, higher rates can achieve higher application-level throughput but their higher PERs generate more retransmissions, which then decreases the application-level throughput. Typically, ARF can recognize this *best* rate and use it extensively but it also tries constantly (every 10 successfully transmitted consecutive packet) to use a higher rate to be able to react to channel condition changes. This process however can be costly since the regular transmission failures generated by ARF decrease the application throughput.

The inability of ARF to stabilize for long periods is a direct consequence of the belief that the long-term variations of the wireless medium can be handled by the same mechanism used to handle its short-term variations. While this is true, there is no reason for it to be very efficient.

4.2 AARF

To avoid the scenario described above, an obvious solution is to increase the threshold used to decide when to increase the current rate from 10 to 40 or 80. While this approach can indeed improve performance in certain scenarios, it does not work in practice since it completely disables the ability of ARF to react to short-term channel condition changes.

This problem led us to the idea that forms the basis of AARF: the threshold is continuously changed at runtime to better reflect the channel conditions. This adaptation mechanism increases the amount of history available to the algorithm, which helps it to make better decisions. In AARF, we have chosen to adapt this threshold by using a Binary Exponential Backoff (BEB, as first introduced in [12]).

When the transmission of the probing packet fails, we switch back immediately to the previous lower rate (as in ARF) but we also multiply by two the number of consecutive successful transmissions (with a maximum bound set to 50) required to switch to a higher rate. This threshold is reset to its initial value of 10 when the rate is decreased because of two consecutive failed transmissions. Detailed pseudo code that describes formally the behavior of ARF and AARF is included in the Appendix A.

The effect of this adaptation mechanism is to increase the period between successive failed attempts to use a higher rate. Fewer failed transmissions and retransmissions improves the overall throughput. For example, Figure 1 shows a period of time where the most efficient transmission mode is mode 3. ARF tries to use mode 4 after 10 successful transmissions with mode 3 while AARF uses the history of the channel and does not increase the rate at each 10 successful packet boundary.

4.3 Performance evaluation

4.3.1 Simulation environment

Because we specifically designed AARF to work well in an infrastructure network[4], we focused on comparing its performance in this environment with ARF and RBAR. To do so, we performed simulations based on the simulation environment described in [11] that uses

[4]AARF is based on ARF which requires at least 10 packets to switch to a higher rate when the transmission conditions improve. In a dense ad hoc network, the wireless medium characteristics can vary many times during the transmission of 1 or 2 packets mainly because of the high collision probability. In this context, it is impossible for ARF or AARF to adapt to the channel characteristics correctly which is why we do not present any ad hoc simulation results.

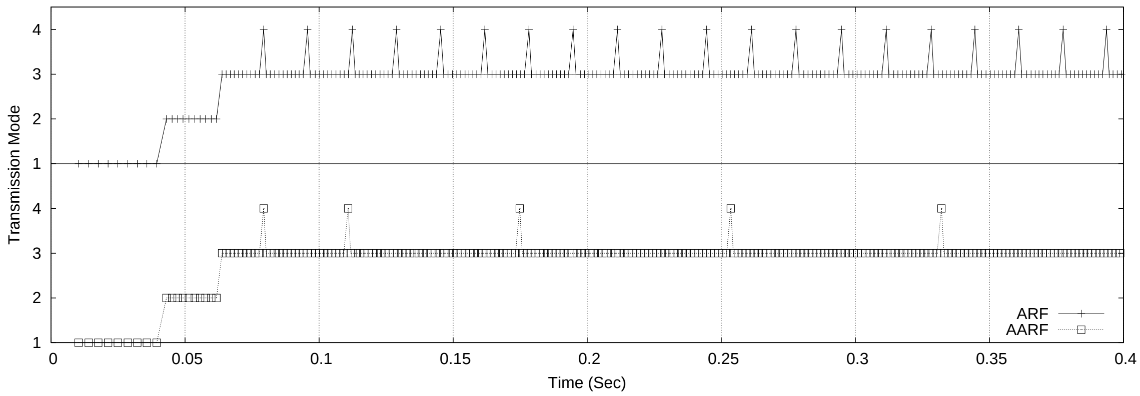

Figure 1: Mode selection comparison between ARF and AARF.

the ns-2 network simulator, with extensions from the CMU Monarch project and the RBAR implementation from [3].

The version of ns-2 on which the RBAR implementation is based does not directly support infrastructure networks. It only supports ad hoc networks and offers the choice of numerous ad hoc routing algorithms. As such, it was impossible to evaluate this algorithm in a multi-node infrastructure network. We thus decided to use the methodology described in [3]: infrastructure networks were simulated with a 2-node ad hoc network. One of the motivations for doing this was also to be able to reproduce the simulation conditions of the results published in this paper exactly and thus to be able to achieve a fair comparison with RBAR.

Our network contains two stations. Station A remains static while station B moves toward station A. The movement of station B is not continuous: it stays static for 60 seconds before moving 5 meters towards station A. Whenever station B stops, a single CBR (Continuous Bit Rate) data transmission towards station A is started. Each CBR packet is 2304 bytes long. Each CBR flow attempts the transmission of 30000 packets 0.8ms apart which generates a 24s continuous data flow. Because the simulations that do not use the RTS/CTS mechanism can achieve a higher throughput peak than what these default CBR flows provide, for these simulations, we used a CBR flow of 50000 packets, each 0.46ms apart.

As shown in Figure 6, the transmission modes 24 Mega Bits Per Second (Mbps) and 48Mbps always perform worse than all other modes during our simulations. We thus chose not to use them in all further experiments and simulations. We also removed the 9Mbps mode because its coverage range is always worse than that of the 12Mbps transmission mode as suggested by [11] and [6].

4.3.2 Algorithm parameters

To analyze the influence of the AARF algorithm parameters, we ran a set of simple simulations which kept all parameters except one constant. The default fixed values as well as the variation range of these parameters are shown in Table 2

Table 2: AARF parameters

Parameter	Default	Variation Range
TimerTimeout (number of packets)	15	11-100
MinSuccessThreshold (number of packets)	10	1-49
MaxSuccessThreshold (number of packets)	50	11-100
SuccessFactor (no unit)	2	1.01-5

The results of these simulations are shown in Figures 2, 3, 4, and 5. As detailed in [15], we used a packet-based timer for ARF and AARF rather than the time-based timer originally described in [2]. The authors of [3] and [15] had already established that its value had little influence on the behavior of ARF (be it time-based or packet-based) and Figure 2 shows that it also has no noticeable influence on the behavior of AARF. The variations of the number of packets transmitted during one simulation represent less than 0.2% of the minimum number of packets transmitted. Similarly, the variations of *MinSuccessThreshold* in Figure 3 represent less than 0.4% of the minimum number of packets transmitted and those of *SuccessFactor* shown in Figure 4 less than 0.45% of the minimum number of packets transmitted.

MaxSuccessThreshold is the only parameter that has a relatively important influence on the performance of the algorithm (see Figure 5). The performance of the algorithm obviously reaches a plateau towards the values of 80 to 90. We chose 50 since it seems to offer a good compromise between the performance obtained and the ability of the algorithm to increase its rate within a reasonable amount of time when the user moves toward the Access Point.

4.3.3 Simulation results

The results for the single rate transmissions are presented in Figure 6. In these simulations, RTS frames, CTS frames, ACK frames, and PLCP headers are sent with BPSK modulation with a FEC (Forward Error Correction) rate equal to 1/2 and a 6 Mbps data rate (basic mode). Note also that all throughput shown in this paper exclude the MAC and PHY headers. Figure 7 shows the mean goodput (the goodput represents the application throughput) achieved by ARF, AARF, and RBAR in the same conditions.

These results show that ARF fails to perform as well as the fixed rates for mode 2, 3, and 4. The main reason for this was explained in Section 4.1: ARF periodically generates transmission failures. RBAR always picks the best available rate which means that the number of transmission failures is much lower. Its mean goodput is thus much higher.

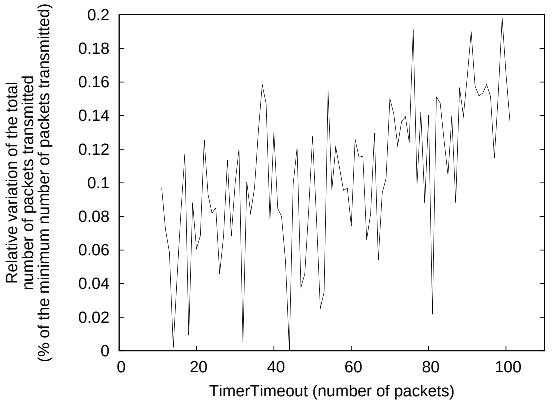

Figure 2: Influence of the value of TimerTimeout on the performance of AARF.

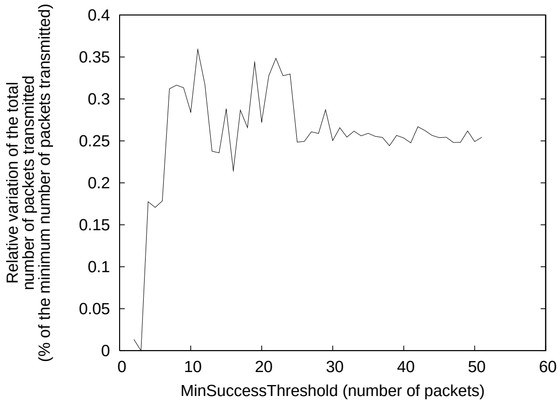

Figure 3: Influence of the value of MinSuccessThreshold on the performance of AARF.

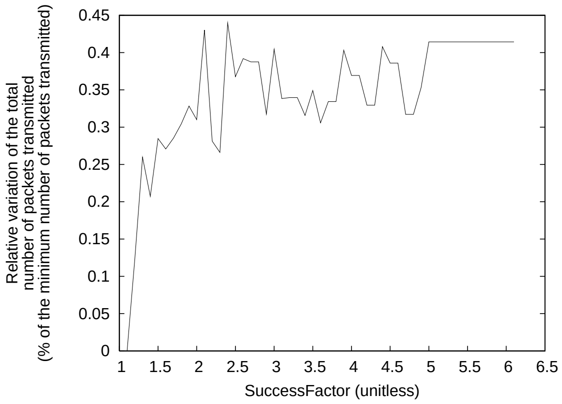

Figure 4: Influence of the value of SuccessFactor on the performance of AARF.

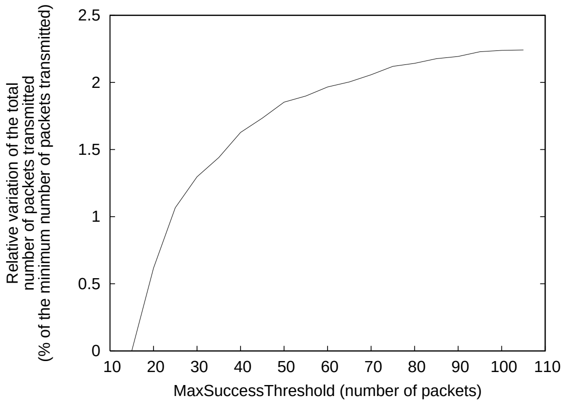

Figure 5: Influence of the value of MaxSuccessThreshold on the performance of AARF.

Figure 7 shows that AARF performs on average the rate selection as well as RBAR and better than ARF. One of its main advantage over RBAR is that it does not require the use of the RTS/CTS protocol. In this case, its performance, as expected, is much higher than that achieved with RBAR as shown in Figure 7.

4.4 Conclusion

The simulation results presented in the previous section clearly show the performance improvement offered by AARF over ARF: it can reach on average the near-optimum perfor-

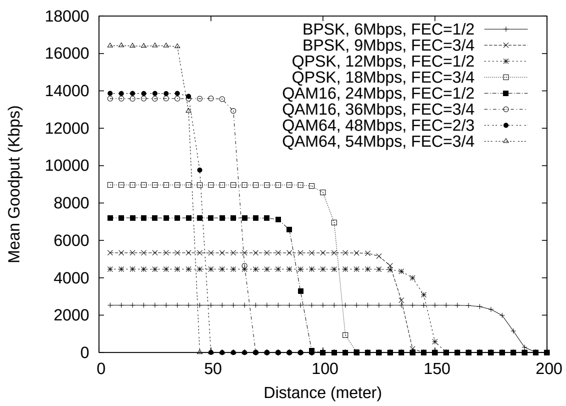

Figure 6: Mean goodput for a single hop with the IEEE 802.11a transmission modes.

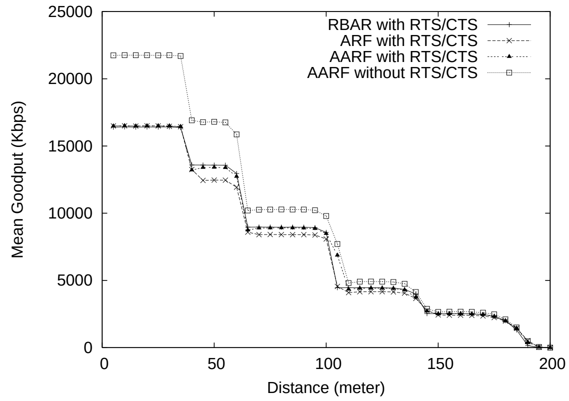

Figure 7: Mean goodput for a single hop with three different automatic rate selection algorithms.

mance of the RBAR algorithm without requiring any incompatible changes to the 802.11 protocol. Furthermore, all it requires from the hardware is a low communication latency between the block which implements the rate control algorithm and the transmission block which handles the ACK timeouts. This new algorithm can thus be easily and incrementally deployed in existing infrastructure networks with a simple firmware or driver upgrade on each node.

5 The Adaptive Multi Rate Retry algorithm

While AARF had been designed to work in a low-latency system, the AR5212-based 802.11 devices to which we had access fell in the high-latency category. This led us to adapt the use of a Binary Exponential Backoff to the hardware we had at hand.

5.1 The AR 5212 chipset

A complete Linux driver for the AR 5212 chipset is available from the Multiband Atheros Driver for WiFi (MADWIFI) project, hosted on SourceForge [7]. This project contains a binary-only Hardware Abstraction Layer (HAL) which hides most of the device-specific registers, a 802.11 MAC implementation imported from the BSD kernel and a Linux AR 5212 driver, heavily inspired by a BSD AR 5212 kernel driver. The HAL exports a very classic interface to the AR 5212 chipset. It allows the user to create up to 9 unbounded FIFOs (First In First Out queue) of transmission descriptors to schedule packets for transmission. Each descriptor contains a status field that holds the transmission status of the descriptor, a pointer, and the size of the data to be transferred. Each transmission descriptor also contains an ordered set of 4 pairs of rate and transmission count fields (r_0/c_0, r_1/c_1, r_2/c_2, r_3/c_3).

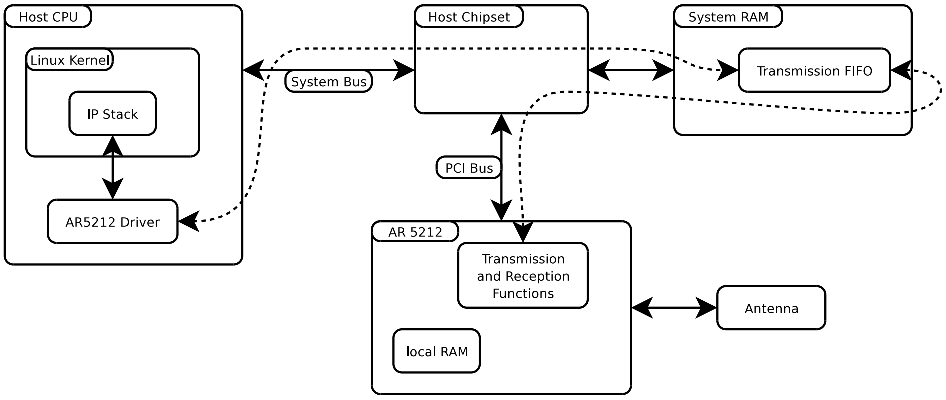

Figure 8: System architecture of an AR 5212-based device.

To schedule the transmission of a data buffer, the software driver inserts in one of the FIFOs a properly initialized transmission descriptor. Whenever the wireless medium is available for transmission, the hardware triggers the transmission of the descriptor located at the head of the FIFO. To do so, it transfers the descriptor and the data pointed to by the descriptor from the system RAM to its local RAM and then starts the transmission of the data with the rate r_0 specified in the descriptor. If this transmission fails, the hardware keeps on trying to send the data with the rate r_0, $c_0 - 1$ times. If the transmission keeps on failing, the hardware tries the rate r_1, c_1 times then the rate r_2, c_2 times and finally the rate r_3, c_3 times. When the transmission has failed $c_0 + c_1 + c_2 + c_3$ times, the transmission is abandoned: the status field of the descriptor is updated and it is transferred back from the local RAM to the system RAM. This process is summarized in Figure 8.

When the transmission is finally completed, or finally abandoned, the hardware also reports in the transmission descriptor the number of missed ACKs for the transmission of this descriptor. It is interesting to note that this number indirectly indicates the final transmission rate of the packet as well as the transmission rate of each retry.

For example, if $c_0 = 1$, $c_1 = 1$, $c_2 = 1$, and $c_3 = 1$, and if the number of missed ACKs is zero, it means that the transmission completed successfully at the first rate r_0. If the number of missed ACKs is 1, it means that the first transmission failed and the second one was completed successfully. If the number of missed ACKs is 3, it means that the first 3 transmissions failed and the fourth one succeeded. Finally, if the number of missed ACKs if 4, it means that all transmissions failed.

5.2 The Madwifi algorithm

The existing MADWIFI driver implements rate control with a two-stage process which is quite natural given the capabilities exported by the HAL. The short-term variations are handled by the Multi Rate Retry mechanism described in the previous section while the long-term variations are handled by changing the value of the r_0/c_0, r_1/c_1, r_2/c_2 and r_3/c_3 pairs at regular fixed intervals (from 0.5 to 1 second intervals).

5.3 The AMRR algorithm

A natural way to introduce a Binary Exponential Backoff in the MADWIFI algorithm is to adapt the length of the period used to change the values of the rate/count pairs and this is exactly what AMRR does. To simplify the logic of the code, we also decided to use heuristics simpler than those in the MADWIFI algorithm to choose the rate/count pairs at the period boundaries.

To ensure that short-term variations of the wireless medium are quickly acted upon, we chose $c_0 = 1$, $c_1 = 1$, $c_2 = 1$ and $c_3 = 1$ (while MADWIFI uses $c_0 = 4$, $c_1 = 2$, $c_2 = 2$ and $c_3 = 2$). The rate r_3 is always chosen to be the minimum rate available (typically, 6Mbps in 802.11a networks). The rates r_1 and r_2 are determined by r_0: we implemented the simplest heuristic possible by setting r_1 and r_2 to the immediately lower available rates. Finally, our rate control algorithm determines r_0 from the previous value of r_0 and the transmission results for the elapsed period. The Appendix B describes the exact heuristics used.

5.4 Performance evaluation

5.4.1 Simulation environment

We used the simulation environment described in Section 4.3 to evaluate the performance of the AMRR algorithm and compare it to that of the MADWIFI and RBAR algorithms. The MADWIFI algorithm we simulated is a trivial copy of the code available in the MADWIFI driver, slightly modified for the simulation environment to use only the 5 transmission modes chosen for our 802.11a networks.

5.4.2 Implementation issues

Our implementation of the MADWIFI algorithm in the simulator and of the AMRR algorithm both in the simulator and in the driver is straightforward except for the way the transmission FIFO, which is shared between the AR5212 chip and the Linux kernel driver, is handled.

More specifically, the original MADWIFI driver initialized the transmission descriptors present in the FIFO only once, when they were inserted into the FIFO. A rather annoying consequence of this behavior is that it can generate wide oscillations of the algorithm due to the different rates of the packets located at the head and at the tail of the FIFO.

For example, when the user application generates a 15 Mb/s data flow and if the wireless channel conditions allow the 802.11a 12Mb/s transmission mode with a reasonable PER ($r_3 = 12$, $r_2 = 6$, $r_1 = 6$ and $r_0 = 6$), the source buffers quickly fill (the transmission descriptor FIFO is thus full) and the user application encounters a lot of packet drops at the source.

If the PER is low-enough at this rate set, the rate control algorithm will try to increase the rate set to $r_3 = 18$, $r_2 = 12$, $r_1 = 6$, and $r_0 = 6$, this means that every new packet that enters the FIFO uses this new rate set. However, at the next decision period boundary, the transmission statistics used to adapt the current rate set are those generated by the transmission of the packets whose rate set is $r_3 = 12$, $r_2 = 6$, $r_1 = 6$, and $r_0 = 6$ and that are still present in the FIFO. Because the PER of this rate set is low enough, the rate control algorithm thus will try to increase the rate set again, yielding something like ($r_3 = 24$, $r_2 = 18$, $r_1 = 12$, and $r_0 = 6$).

At one point, the packets whose rate set is high will reach the front of the FIFO and will be treated by the hardware: they are likely to fail which will make the rate control algorithm drop the current rate set quickly. However, it is likely to decrease the rate set too much for the same reasons it increased it too much previously. We have observed this phenomenon during preliminary experiments and we have reproduced it in simulation as shown in the curve named *Original MADWIFI* in Figure 10.

We chose to avoid this problem by modifying the MADWIFI driver to parse the transmission FIFO each time a rate change happens to apply the rate change to each transmission descriptor concerned immediately. All further simulations and experiments (unless explicitly stated) were conducted with this modified version of the MADWIFI algorithm.

It should be noted that the painful and costly process of parsing the transmission descriptor FIFO whenever a rate change needs to be applied could be alleviated if proper hardware support for this had been provided. For example, it should be possible to include in each transmission descriptor a pointer to the rate/count pairs rather than the rate/count pairs themselves. Alternative designs that use an on-chip cache of rate/count patterns on a per-destination basis would also be possible.

5.4.3 Algorithm parameters

Because the AMRR algorithm is based on the same set of ideas developed for AARF, that is, the use of a BEB to adapt the success threshold, similar parameters can be tweaked. Among these, the *MinSuccessThreshold* and the *MaxSuccessThreshold* parameters are the two most important parameters. We did not bother to evaluate the influence of *MinSuccessThreshold* since increasing it would further decrease the ability of the AMRR algorithm to react to channel condition changes. Figure 9 shows how *MaxSuccessThreshold* influences the output of the AMRR algorithm.

As expected, *MaxSuccessThreshold* follows the same pattern observed in Figure 5: the throughput increases with its increase. As in Section 4.3.2, we do not choose the highest value possible to avoid decreasing its ability to react to channel condition changes rapidly and settle for the value of 15 which is quite close to the plateau maximum.

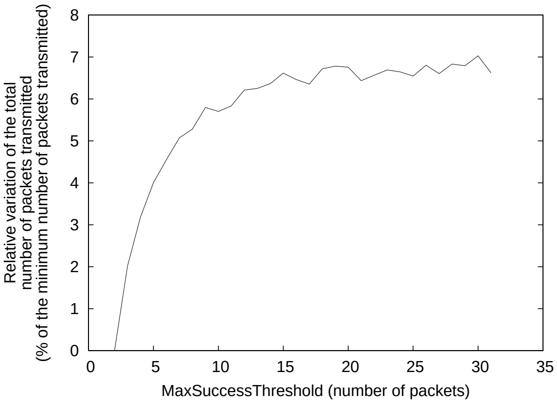

Figure 9: Influence of the value of MaxSuccessThreshold on the performance of AMRR

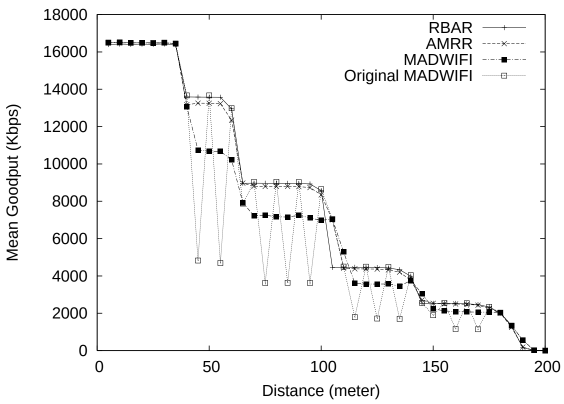

Figure 10: Mean goodput for a single hop with RBAR, MADWIFI, and AMRR mode selection.

5.4.4 Performance results

The simulation results, which are summarized in Figure 10, clearly show that AMRR performs much better than the original rate control algorithm used in the MADWIFI driver and that it achieves similar performance to RBAR on average. Here again, the BEB-based adaptive mechanism is the main reason for this throughput improvement: the probability of trying a rate set which requires numerous retransmissions is greatly diminished.

5.5 Experimental results

Our test setup was created to approximate as closely as possible real-world use cases. As such, we chose a typical office environment with many people walking from one office to the other: a 802.11b/g Access Point (a Netgear WG602) was setup with a private SSID in an office and a laptop with a *Proxim Orinoco Gold* pcmcia card based on the AR5212 chipset was setup in another office approximatively 10 meters away from the Access Point. We first installed an unmodified 2.6.5 Linux kernel and a RedHat 8.0 Linux distribution on the test laptop and then tested three versions of the Madwifi driver:

- *Original Madwifi*: the original unmodified Madwifi driver.
- *Madwifi*: the Madwifi driver modified to apply immediately rate changes on its transmission FIFO as described in Section 5.4.2.
- *AMRR*: the Madwifi driver modified to apply immediately rate changes and implement the AMRR rate control algorithm.

To mitigate the variations of the transmission conditions with time, we ran three sets of experiments whose results are shown in Figures 11, 12, and 13. The goal of each of the three sets of experiments was to compare the average throughput achieved by two of the three drivers.

For each set of experiments, we loaded in the Linux kernel alternatively each of the two selected drivers and started a 600 second continuous 30Mbps UDP stream from the laptop to the only machine located on the 100Mbps ethernet link of the Access Point. We executed this test 5 times for each of the two selected drivers and recorded the average throughput achieved during each experiment.

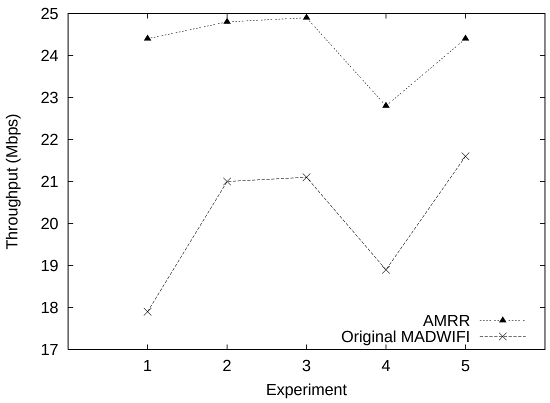

Figure 11: AMRR versus Original MADWIFI

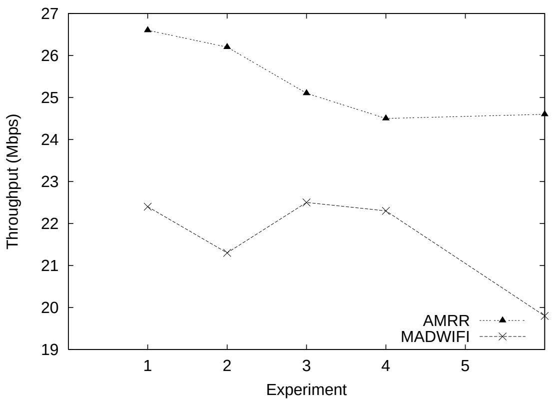

Figure 12: AMRR versus MADWIFI

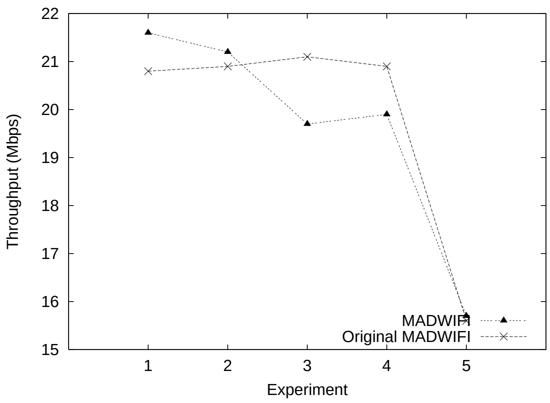

Figure 13: MADWIFI versus Original MADWIFI

Despite the variability of the experiments, we can observe the clear performance improvement achieved by *AMRR* over both *Original Madwifi* and *Madwifi* in Figures 12 and 13. Figures 12 and 13 show that *AMRR* reached on average 24Mbps while *Original Madwifi* and *Madwifi* reached on average 20Mbps. Figure 11 shows that *Original Madwifi* and *Madwifi* do not have a significative average throughput difference even though we could observe a clear throughput oscillation for *Original Madwifi* during these experiments.

6 Conclusion

In this paper, we described the fundamental difference between two classes of 802.11 devices and its influence on the design of practical rate adaptation algorithms. Two novel algorithms, AARF based on ARF[2] and AMRR based on MADWIFI[7] respectively designed for *low latency* and *high latency* systems are presented. Simulations show that they both perform close to the optimum represented by the impractical RBAR[3] in the case of infrastructure networks. An implementation of the AMRR algorithm in a Linux kernel driver for AR5212-based[14] devices brings further evidence that these algorithms improve the achievable performance and can be readily implemented in existing devices.

Acknowledgments

We would like to thank the Madwifi[7] project for the Linux driver that we used to implement our AMRR algorithm and Gavin Holland for providing his ns implementation of RBAR[3]. Special thanks go to Slim Chabbouh, Ph.D student at the ENST in the Digital Communications group who spent long hours explaining us digital radio architecture and digital signal processing techniques applied to digital radios.

References

[1] D. Qiao, S. Choi, A. Jain, K. G. Shin, "MiSer: an optimal low-energy transmission strategy for IEEE 802.11a/h", Proceeding of MOBICOM 2003, Pages 161-175.

[2] A. Kamerman and L. Monteban, "WaveLAN-II: A High-performance wireless LAN for the unlicensed band", Bell Lab Technical Journal, Pages 134-140, ARRL, 1990.

[3] G. Holland, N. Vaidya and P. Bahl, "A Rate-Adaptive MAC Protocol for Multi-Hop Wireless Networks", in Proc. of ACM MOBICOM, Rome, July 2001.

[4] B. Sadeghi, V. Kanodia, A. Sabharwal, and E. Knightly, "Opportunistic Media Access for Multirate Ad Hoc Networks", in Proc. of ACM MOBICOM, Atlanta, September 2002.

[5] http://www.phys.uu.nl/~vdvegt/docs/gron/

[6] D. Qiao, Sunghyun Choi, "Goodput Enhancement of IEEE 802.11a Wireless LAN via Link Adaptation", in Proc. of IEEE ICC'01, Helsinki, Finland, June 2001.

[7] http://sourceforge.net/projects/madwifi/

[8] http://www.isi.edu/nsnam/ns/

[9] Agere systems, "WaveLAN 802.11b chipset for Standard Form Factors", Preliminary Product Brief, December 2002.

[10] Agere systems, "WaveLAN WL60040 Multimode Wireless Lan Media Access Controller", Product Brief, August 2003.

[11] M.H. Manshaei, T. Turletti, "Simulation-Based Performance Analysis of 802.11a Wireless LAN", In Proceeding of International Symposium on Telecommunications, Iran, August 2003.

[12] Robert M.Metcalfe, David R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", In Communications of the ACM, vol 19, No. 5, July 1976, pp 395-404.

[13] TimeSys Corporation, "A TimeSys Perspective on the Linux Preemptible Kernel", White Paper version 1.0

[14] http://www.atheros.com/pt/index.html

[15] Daji Qiao, Sunghyun Choi, and Kang G. Shin, "Goodput Analysis and Link Adaptation for IEEE 802.11a Wireless LANs", In IEEE Transaction on mobile computing Vol. 1, No. 4, October-December 2002

A AARF Pseudo Code

The following variables are defined:

- *timer*: incremented for each packet transmitted (regardless of success or failure). Reset to zero when two or more consecutive retries are done or when 10 consecutive packets have been successfully transmitted.
- *success*: number of consecutive successful packet transmissions.
- *recovery*: if it is set to TRUE, it means that we have just sent a probing packet to try a higher rate. Otherwise, it is set to FALSE.
- *retry*: number of consecutive retries for a given packet.
- *success_threshold*: number of consecutive successful transmissions required to send a probe packet (to try a higher rate).
- *timeout*: if *timer* reaches this value, a probe packet is sent.
- *retry_threshold*: number of consecutive failures for one packet required to drop this packet.
- *min_success_threshold*: value used to initialize the variable *success_threshold* when the rate is decreased because of two consecutive transmission failures.
- *max_success_threshold*: maximum possible value of *success_threshold*.
- *success_k*: multiplicative factor used to calculate the new value of *success_threshold* when the probe packet fails.
- *min_timeout*: minimum value of *timeout*.
- *timeout_k*: in AARF, *timeout* is set to either *min_timeout* or *success_threshold* $\times$ *timeout_k*.

```
static int g_success = 0;
static int g_timer = 0;
static int g_recovery = 0;
static int g_success_threshold = 10;
static int g_timeout = 15;
static int g_retry_threshold = 4;

#ifdef AARF
static int g_min_success_threshold = 10;
static int g_max_success_threshold = 50;
static int g_min_timeout = 15;
static double g_success_k = 2;
static double g_timeout_k = 1.5;
```

```
#define report_recovery_failure() \
{ \
  g_success_threshold = min (g_success_threshold * g_success_k, g_max_success_threshold); \
  g_timeout = max (g_timeout_k * g_success_threshold, g_min_timeout); \
}

#define report_failure() \
{ \
  g_success_threshold = g_min_success_threshold; \
  g_timeout = g_min_timeout; \
}
#else /* AARF */
#define report_recovery_failure()
#define report_failure()
#endif /* AARF */

int send_one_packet (Packet packet, int rate)
{
  int retry = 0;
  Status status;

  while (retry < g_retry_threshold) {
    status = send_packet (packet, rate);
    if (status == SUCCESS) {
      g_success++;
      retry = 0;
      if ((g_success == g_success_threshold
           || g_timer == g_timeout)
          && !is_max_rate (rate)) {
        rate = increment_rate (rate);
        g_timer = 0;
        g_success = 0;
        recovery = TRUE;
      } else {
        g_timer++;
        g_recovery = FALSE;
      }
      break;
    } else {
      g_timer++;
      retry++;
      g_success = 0;
      if (g_recovery) {
        g_timer = 0;
        if (retry == 1) {
          report_recovery_failure ();
          if (!is_min_rate (rate)) {
            rate = decrement_rate (rate);
          }
        }
      } else {
        if (retry == 2
            || retry == 4
```

```
          || retry == 6
          || retry == 8
          || retry == 10) {
        report_failure ();
        if (!is_min_rate ()) {
          rate = decrement_rate (rate);
        }
      }
    }
    if (retry >= 2) {
      g_timer = 0;
    }
  }
}
return rate;
}
```

B AMRR Pseudo Code

All variables used in the AMRR Pseudo Code have already been defined for the AARF Pseudo Code. The following functions are used:

- *is_success*: returns TRUE if less than 10% of the packet transmissions failed during the previous period, FALSE otherwise.
- *is_failure* returns TRUE if more than 33% of the packet transmissions failed during the previous period, FALSE otherwise.
- *is_enough*: returns whether or not enough packets were transmitted during the previous period to get significative statistics. By default, *is_enough* returns TRUE if the transmission of 10 distinct packets was attempted during the previous period, FALSE otherwise.

```
void
update_timer (void)
{
  int need_change = FALSE;
  if (is_success () && is_enough ()) {
    g_success++;
    if (g_success >= g_success_threshold
        && !is_max_rate ()) {
      g_recovery = TRUE;
      g_success = 0;
      increase_rate ();
      need_change = true;
    } else {
      g_recovery = FALSE;
    }
  } else if (is_failure ()) {
    g_success = 0;
```

```
    g_recovery = FALSE;
    if (!is_min_rate ()) {
      if (m_recovery) {
        m_success_threshold *= 2;
        m_success_threshold = min (m_success_threshold, m_max_success_threshold);
      } else {
        m_success_threshold = m_min_success_threshold;
      }
      decrease_rate ();
      need_change = true;
    }
  }
  if (is_enough () || need_change) {
    reset_cnt ();
  }
}
```

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399