DISSERTAÇÃO DE MESTRADO

AVALIAÇÃO DE DISCIPLINAS DE CONSULTA EM PROTOCOLO DE CONTROLE DE ACESSO AO MEIO INICIADO PELO RECEPTOR PARA REDES SEM FIO *AD HOC*

Fadhil Firyaguna

Brasília, Dezembro de 2014

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia

DISSERTAÇÃO DE MESTRADO

AVALIAÇÃO DE DISCIPLINAS DE CONSULTA EM PROTOCOLO DE CONTROLE DE ACESSO AO MEIO INICIADO PELO RECEPTOR PARA REDES SEM FIO *AD HOC*

Fadhil Firyaguna

Relatório submetido ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Mestre em Engenharia de Sistemas Eletrônicos e de Automação

Banca Examinadora

Prof. Marcelo Menezes de Carvalho, ENE/UnB ______________________
Orientador

Prof. Renato Mariz de Moraes, ENE/UnB ______________________
Examinador interno

Prof. José Ferreira de Rezende , COPPE/UFRJ ______________________
Examinador externo

Dedicatória

Ao leitor que pesquisou algum termo deste trabalho, por acaso encontrou este e ficou curioso em lê-lo. Ao leitor que não encontrou o que estava procurando e leu este trabalho por ter sido sua última opção. Ao leitor que estava querendo buscar algo sobre o tema deste trabalho e o encontrou na primeira opção. Ao leitor ao qual pedi para procurar este trabalho para ler e o fez. E a todos que desejaram o sucesso deste trabalho.

Fadhil Firyaguna

Agradecimentos

Agradeço a todos os que acreditaram em mim nesta jornada. Em especial, segue a lista: Igor Coelho, que sempre me ajudou em programação desde que eu era um calouro de graduação e foi meu primeiro exemplo que me inspirou a seguir na academia e entrar para o Mestrado. Tiago Bonfim, que praticamente me deu todo este trabalho para fazer e sempre foi atencioso comigo desde a era da Iniciação Científica. Larissa Eglem e Lucas Soares, que me acompanharam de perto nessa jornada que fizemos juntos, sempre ajudando uns aos outros em problemas específicos do trabalho de cada um, debugando códigos, dando ideias novas, tendo epifanías conjuntas! Mateus Marcuzzo, que me ajudou no desenvolvimento do código no simulador. Juan Camilo, Ruben Ortega, Luciano Mauro, Stephanie Alvares, Helard Becerra, Ricardo Kehrle, e Dário Morais, hermanos de diversos países latino-americanos que me proporcionaram uma rica e excelente convivência no GPDS. Evandro Costa, Éverton Andrade, Camila Lumy, Thayane Viana, Eduardo Vargas e Eduardo Dias, meus calouros que foram sempre atenciosos quando eu tentava explicar ou ensinar alguma coisa e que me deixavam participar dos seus trabalhos, mesmo que sendo dando apoio moral, ajudando no código, ou no meio da rua segurando tablets! Professor Marcelo M. Carvalho, que sempre motivou a mim e os outros alunos do NERdS a sempre tentar fazer um pouco mais que o nosso melhor (olha quantos prêmios e conquistas esse grupo já teve!). Raíssa Kapiski, que sempre me acolheu e me apoiou até nos momentos mais sombrios desta jornada. E minha família, que me deu os valores para chegar onde estou e para seguir em frente.

Fadhil Firyaguna

"Nothing is true, everthing is permitted".

— The Creed's maxim.

RESUMO

O estudo de disciplinas de consulta para protocolos da sub-camada de controle de acesso ao meio (MAC, do inglês, *Medium Access Control*) iniciados pelo receptor para redes *ad hoc* não tem recebido muita atenção na literatura, e esquemas simples como a consulta cíclica e a priorização uniforme são normalmente assumidos. Porém, não apenas a ordem, mas também a taxa com a qual os nós são consultados é importante: uma taxa de consulta que é muito baixa pode levar a uma baixa vazão e longos atrasos, enquanto que o oposto pode acarretar um tráfego de controle excessivo e um número maior de colisões de quadros. Idealmente, um protocolo MAC iniciado pelo receptor teria seu melhor desempenho se os nós pudessem saber "quem" e "quando" consultar baseados na disponibilidade de dados em seus vizinhos. A primeira parte desta dissertação investiga um protocolo MAC para comunicação ponto-a-ponto ("unicast") que segue o paradigma de transmissão com iniciativa do receptor, baseado na reversão do algoritmo de recuo exponencial binário (BEB, do inglês, *binary exponential backoff*) do padrão IEEE 802.11, como forma de controlar a taxa com que os nós são consultados. Com o algoritmo BEB, a taxa de consulta é auto-regulada de acordo com as condições de canal e de tráfego. Além disso, o reordenamento de quadro nas filas — onde um quadro pode ser transmitido ao ser consultado sem a necessidade de estar na cabeça da fila — e um novo quadro de controle, o NTS (do inglês, *Nothing-to-send*), cujo papel é avisar ao nó consultor que não há quadros de dados disponíveis, são apresentados para agilizar os turnos de consulta. O desempenho do protocolo MAC iniciado pelo receptor baseado no algoritmo BEB é investigado sob três disciplinas de consulta: uma consulta cíclica sem prioridades (*"Round-robin"*), uma que visa a justiça de vazão entre os nós, a disciplina de *justiça proporcional* (PF, do inglês, *proportional fair*) e uma que prioriza os nós de acordo com a *probabilidade de sucesso de estabelecimento de conexão* (LSH, do inglês, *likelihood of successful handshake*). Comparações com o padrão IEEE 802.11 em relação à sobrecarga de controle, atraso, justiça, e vazão, de acordo com diferentes topologias e cenários de tráfego, sÃčo apresentadas.

A partir dos resultados obtidos na avaliação das três disciplinas, é proposta uma variação da estratégia de consulta que seleciona dinamicamente o algoritmo a ser utilizado na escolha do destino da consulta. O protocolo MAC iniciado pelo receptor com o algoritmo BEB revertido combinado a esta nova estratégia de consulta denominou-se de Receiver-Initiated MAC with Adaptive Polling Discipline (RIMAP), um protocolo MAC para comunicação ponto-a-ponto ("unicast") que dinamicamente seleciona uma disciplina de consulta de acordo com a contenção do canal e a homogeneidade da qualidade do enlace de todos os vizinhos. Para isso, duas disciplinas de consulta são consideradas: o LSH e PF. O comportamento adaptativo é controlado por dois parâmetros de comutação que podem ser ajustados para se obter um compromisso entre o desempenho de justiça e de vazão/atraso. O desempenho do RIMAP é avaliado com simulações a eventos discretos sob topologias com terminais escondidos, transmissões concorrentes, e tráfego saturado. Adicionalmente, seu desempenho é comparado com o mesmo protocolo baseado no algoritmo BEB com as disciplinas de consulta fixadas (LSH e PF somente), assim como comparado com o MAC do padrão

IEEE 802.11, o representante do paradigma iniciado pelo transmissor.

ABSTRACT

The study of polling disciplines for receiver-initiated MAC protocols for ad hoc networks has not received much attention in the literature, and simple schemes such as round-robin or uniform prioritization are usually assumed. However, not only the order, but also the rate at which nodes are polled is significant: a polling rate that is too slow may render low throughput and high delays, whereas the opposite may lead to excessive control traffic and frame collisions. Ideally, a receiver-initiated MAC would perform best if nodes could know "whom" and "when" to poll based on data availability. The first part of this work investigates a receiver-initiated unicast MAC protocol that is based on reversing the binary exponential backoff (BEB) algorithm of the IEEE 802.11 as a means to control the rate at which nodes are polled. With the BEB algorithm, the polling rate is self-regulated according to channel and traffic conditions. Additionally, frame reordering at queues — where a frame can be transmitted when polled with no need to be in the head of queue — and a new control frame, the Nothing-to-send (NTS), whose role is to notify the polling node that there is no data frame available, are introduced to speed up polling rounds. The performance of the BEB-based receiver-initiated MAC is investigated under three polling disciplines: a cyclic polling without priorities ("Round-robin"), one that targets throughput fairness among nodes, the *proportional fair* (PF) discipline, and one that prioritizes nodes according to the *likelihood of successful handshake* (LSH). Comparisons with the IEEE 802.11 with respect to control overhead, delay, fairness, and throughput, according to different topologies and traffic scenarios, are presented.

From the results obtained in the evaluation of the three disciplines, it is proposed a variation of the polling strategy that selects dynamically the algorithm to be utilized in the choice of the polling destination. The receiver-initiated MAC protocol with the BEB algorithm combined with this new strategy is named Receiver-Initiated MAC with Adaptive Polling Discipline (RIMAP), a unicast MAC protocol that dynamically selects a polling discipline according to channel contention and link quality homogeneity to all neighbors. For that, two polling disciplines are considered: the LSH and the Proportional Fair (PF). The adaptive behavior is controlled by two switching parameters that can be tuned to trade off fairness with throughput-delay performance. RIMAP performance is evaluated with discrete-event simulations under topologies with hidden terminals, concurrent transmissions, and saturated traffic. Also, its performance is compared with the same BEB-based MAC protocol under fixed polling disciplines (LSH or PF only), as well as with the IEEE 802.11 DCF MAC, a representative of sender-initiated paradigms.

SUMMARY

1 Introdução **1**
- 1.1 Contextualização 3
- 1.2 Definição do problema e Objetivos da Dissertação 3
- 1.3 Contribuições 5
- 1.4 Apresentação da Dissertação 5

2 Introduction **7**
- 2.1 Contextualization 9
- 2.2 Problem Definition and Dissertation Objectives 9
- 2.3 Contributions 10
- 2.4 Dissertation Organization 11

3 Related Work **12**

4 Protocol Description **16**
- 4.1 Binary Exponential Backoff (BEB) Algorithm 16
- 4.2 Initialization 18
- 4.3 Frame Reordering 19
- 4.4 Virtual Carrier Sensing 21
- 4.5 Data Acknowledgment and Retransmission 23
- 4.6 Summary 25

5 Polling Disciplines **27**
- 5.1 Round-Robin Discipline 27
- 5.2 Proportional Fair Discipline 28
- 5.3 Likelihood of Successful Handshake (LSH) Discipline 29
- 5.4 Summary 31

6 Performance Evaluation **32**
- 6.1 Simulation Setup 33
- 6.2 Control Overhead 35
- 6.2.1 Scenario A 35
- 6.2.2 Scenario B 38
- 6.3 Average Delay per DATA Frame 39

6.3.1 Scenario A ... 39
6.3.2 Scenario B ... 40
6.4 Fairness ... 42
6.4.1 Scenario A ... 42
6.4.2 Scenario B ... 43
6.5 Average Throughput ... 44
6.5.1 Scenario A ... 44
6.5.2 Scenario B ... 45
6.6 Conclusions ... 45

7 RIMAP: Receiver-Initiated MAC Protocol with Adaptive Polling Discipline ... 50
7.1 Adaptive Polling Discipline ... 51
7.2 Performance Evaluation of RIMAP for Ad Hoc Networks ... 52
7.2.1 Average Point-to-Point Delay per DATA frame ... 54
7.2.2 Average Throughput per Flow ... 54
7.2.3 Fairness ... 55
7.3 Conclusion ... 56

8 Conclusions ... 57
8.1 Future Work ... 59

REFERENCES ... 62

Appendix ... 66

I Performance of the Dynamic Polling Discipline in RIMAP Protocol for Ad Hoc Networks ... 67
I.1 Average Point-to-Point Delay per DATA frame ... 67
I.1.1 Scenario A ... 67
I.1.2 Scenario B ... 68
I.2 Average Throughput per Flow ... 69
I.2.1 Scenario A ... 69
I.2.2 Scenario B ... 70
I.3 Fairness ... 71
I.3.1 Scenario A ... 71
I.3.2 Scenario B ... 72

II Topology Generation ... 73

III NS-3 Main Script ... 76

IV NS-3 Changelog ... 86
IV.1 DcaTxop ... 86

IV.1.1 dca-txop.h 86
IV.1.2 dca-txop.cc 86
IV.2 DcfManager 91
IV.2.1 dcf-manager.h 91
IV.2.2 dcf-manager.cc 92
IV.3 EdcaTxopN 92
IV.3.1 edca-txop-n.h 92
IV.3.2 edca-txop-n.cc 92
IV.4 MacLow 93
IV.4.1 mac-low.h 93
IV.4.2 mac-low.cc 95
IV.5 RegularWifiMac 102
IV.5.1 regular-wifi-mac.h 102
IV.5.2 regular-wifi-mac.cc 102
IV.6 WifiMacHeader 102
IV.6.1 wifi-mac-header.h 102
IV.6.2 wifi-mac-header.cc 103
IV.7 WifiMacQueue 104
IV.7.1 wifi-mac-queue.h 104
IV.7.2 wifi-mac-queue.cc 104
IV.8 WifiRemoteStationManager 106
IV.8.1 wifi-remote-station-manager.h 106
IV.8.2 wifi-remote-station-manager.cc 108

FIGURES LIST

4.1 Example of backoff contention window growth. 17
4.2 Example of backoff timer decrement. 17
4.3 Receiver Initiated MAC initialization flowchart. 18
4.4 Example of frame reordering technique compared to FIFO queue. (a) FIFO operation. (b) Frame Reordering (FR). Average delay of the DATA frames is reduced with FR technique. Blue frames are polling RTR frames, red frames are NTS, green frames are ACK. 20
4.5 Frame reordering in MAC queue. The first packet addressed to RTR source is transmitted. 21
4.6 NTS transmission. Since there is no packet addressed to RTR source, a negative response is transmitted. 21
4.7 Receiver-Initiated handshake cases. (a) Polling with positive DATA response and acknowledgement. (b) Polling with negative response (backoff starts earlier). (c) Polling with no response. (d) DATA transmission with failed acknowledgment. 22
4.8 Example of ACK timeout. Data packet is enqueued back in MAC queue head-of-line. 23
4.9 Receiver-Initiated MAC flowchart. 25

5.1 Example of Round-Robin discipline in neighborhood table. 28
5.2 Example of Proportional Fair discipline in neighborhood table. 29
5.3 Example of Likelihood of Successful Handshake discipline in neighborhood table. 29
5.4 Monte Carlo simulations for the estimated probability of successful handshake computed by using both approaches (Eq. (5.3) and Eq. (5.4)). 31

6.1 Average traffic distribution per node in the neighborhood. In Scenario A, all neighbors have data available. In Scenario B, only one third of the neighbors have data available. 33
6.2 Topologies with different sparsity levels used in simulations. Green lines indicate nodes within carrier sensing range of each other, and black lines indicate transmit/receive pairs. Top row shows topologies from scenario A. Bottom row shows topologies from scenario B. 34
6.3 Average number of packets in the MAC queue of node 5. 35
6.4 (a) Control overhead for each polling discipline compared to IEEE 802.11b, across different groups of topology sparsity in Scenario A. (b) Control overhead gain over IEEE 802.11b. 37

6.5 (a) Control overhead for each polling discipline compared to IEEE 802.11b, across different groups of topology sparsity in Scenario B. (b) Control overhead gain over IEEE 802.11b. 38
6.6 (a) Average point-to-point (or link) delay for each polling discipline compared to IEEE 802.11b, across different groups of topology sparsity in Scenario A. (b) Gain on average point-to-point delay over IEEE 802.11b. 40
6.7 (a) Average point-to-point (or link) delay for each polling discipline compared to IEEE 802.11b, across different groups of topology sparsity in Scenario B. (b) Gain on average point-to-point delay over IEEE 802.11b. 41
6.8 (a) Jain's fairness index for different polling disciplines and different topologies in Scenario A, according to their average number of hops. All disciplines are compared to sender-initiated IEEE 802.11 DCF MAC. (b) Gain of fairness with respect to IEEE 802.11b. 43
6.9 (a) Jain's fairness index for different polling disciplines and different topologies in Scenario B, according to their average number of hops. All disciplines are compared to sender-initiated IEEE 802.11 DCF MAC. (b) Gain of fairness with respect to IEEE 802.11b. 44
6.10 (a) Average throughput for different polling disciplines and different topologies in Scenario A, according to their average number of hops. All disciplines are compared to sender-initiated IEEE 802.11 DCF MAC. (b) Gain of average throughput with respect to IEEE 802.11b. (c) Average aggregate throughput. 48
6.11 (a) Average throughput for different polling disciplines and different topologies in Scenario B, according to their average number of hops. All disciplines are compared to sender-initiated IEEE 802.11 DCF MAC. (b) Gain of average throughput with respect to IEEE 802.11b. (c) Average aggregate throughput. 49

7.1 Sample topologies: green lines indicate nodes within carrier sensing range of each other, and black lines indicate transmit/receive pairs. 53
7.2 Average delay per data frame. 54
7.3 Average flow throughput. 55
7.4 Jain Fairness Index. 55

I.1 Average delay per data frame in Scenario A topology Type 0 (fully-connected). 67
I.2 Average delay per data frame in Scenario A topology Type 2. 67
I.3 Average delay per data frame in Scenario A topology Type 4. 67
I.4 Average delay per data frame in Scenario B topology Type 0 (fully-connected). 68
I.5 Average delay per data frame in Scenario B topology Type 2. 68
I.6 Average delay per data frame in Scenario B topology Type 4. 68
I.7 Average throughput per flow in Scenario A topology Type 0 (fully-connected). 69
I.8 Average throughput per flow in Scenario A topology Type 2. 69
I.9 Average throughput per flow in Scenario A topology Type 4. 69
I.10 Average throughput per flow in Scenario B topology Type 0 (fully-connected). 70
I.11 Average throughput per flow in Scenario B topology Type 2. 70

I.12 Average throughput per flow in Scenario B topology Type 4. 70
I.13 Jain Fairness Index in Scenario A topology Type 0 (fully-connected). 71
I.14 Jain Fairness Index in Scenario A topology Type 2. 71
I.15 Jain Fairness Index in Scenario A topology Type 4. 71
I.16 Jain Fairness Index in Scenario B topology Type 0 (fully-connected). 72
I.17 Jain Fairness Index in Scenario B topology Type 2. 72
I.18 Jain Fairness Index in Scenario B topology Type 4. 72

TABLES LIST

6.1 Types of topologies used in simulations are classified according to the average number of neighbors per node and average number of hops in the topology. 34
6.2 Physical layer parameters. 36
6.3 MAC layer parameters 36
6.4 Application layer parameters 36
6.5 Simulation Parameters 37

II.1 Parameters for topology generation. 73

Chapter 1

Introdução

Nos últimos anos, houve uma rápida expansão na indústria da computação móvel devido à popularização e às interfaces cada vez mais amigáveis do dispositivo com o usuário. Porém, os dispositivos de comunicação sem fio atuais, aplicações e protocolos são projetados, em sua grande maioria, para uso em redes celulares ou redes sem fio locais (WLANs, do inglês, *wireless local area networks*), sem levar em conta o grande potencial oferecido pelas redes *ad hoc*. Uma rede *ad hoc* é um conjunto autônomo de dispositivos móveis (tablets, smartphones, sensores, etc.) que comunicam-se entre si através de enlaces sem fio e cooperam de maneira distribuída através de repasse de pacotes e de atividade de roteamento executados por todos os dispositivos, a fim de prover uma funcionalidade de rede necessária na ausência de uma infraestrutura fixa [1].

Os dispositivos de redes *ad hoc* são geralmente alimentados à bateria (restrição energética), podem prover diversos tipos de funcionalidades (heterogeneidade), são capazes de associar-se e desassociar-se da rede livremente, de modo que podem se mover aleatoriamente (topologia dinâmica), e é possível se organizarem dinamicamente a fim de implantar uma rede funcional na ausência de uma administração central (autonomia). Entretanto, uma rede *ad hoc* enfrenta os mesmos problemas tradicionais inerentes às comunicações sem fio, tais quais uma menor confiabilidade em relação a um meio cabeado, segurança da camada física limitada, canais variantes no tempo, interferência, etc. Mas, apesar das várias restrições, as redes *ad hoc* são altamente satisfatórias para o uso em situações onde uma infraestrutura fixa é inexistente, não confiável, ou muito cara. Devido à sua capacidade de auto-criação, auto-organização e auto-administração, redes *ad hoc* podem ser rapidamente implantadas com o mínimo de intervenção do usuário, sem a necessidade de um planejamento detalhado de instalação de estações rádio-base ou de cabeamento.

Como consequência, há uma expectativa de que as redes *ad hoc* se tornem uma importante parte da futura arquitetura das redes de nova geração (4G, 5G...), que visam prover ambientes que dêem suporte ao usuário em realizar suas tarefas, acessando informação e se comunicando a qualquer hora, em qualquer lugar, e de qualquer dispositivo [2, 3]. Neste contexto, há uma vasta gama de aplicações para o uso de redes *ad hoc* em diversas áreas:

- Redes táticas: comunicações e operações militares, campos de batalha automatizados [4];

- Serviços de emergência: operações de busca e salvamento, recuperação de desastres [5];
- Serviços veiculares: orientação de rodovias e acidentes, transmissão de condições da via e de condições climáticas, rede de táxis, redes inter-veiculares [6];
- Redes de sensores: sensores domésticos (smarthomes), rastreamento de condições ambientais, monitoramento de fauna [7];
- Entretenimento: jogos multi-usuários [8], comunicação par-a-par (P2P) [9], redes sociais móveis [10];
- Extensão de cobertura: extensão do acesso à rede (*onloading*) [11], escoamento de tráfego da infraestrutura celular pelos dispositivos móveis (*offloading*) [12], redes oportunísticas [13].

As características específicas de redes *ad hoc* impõem vários desafios no projeto de protocolos de redes em todas as camadas da pilha de protocolos. A camada física deve lidar com mudanças rápidas nas características do enlace. A camada de controle de acesso ao meio (MAC, do inglês, *Medium Access Control*) deve permitir um acesso justo ao canal, minimizar colisões de pacotes e lidar com os problemas de terminal escondido e exposto. Na camada de rede, os nós devem cooperar para calcular rotas. A camada de transporte deve ser capaz de manipular perda de pacotes e atrasos característicos que são muito diferentes de redes cabeadas. Aplicações devem estar aptas a tratar possíveis desconexões e reconexões. Além disso, o desenvolvimento dos protocolos deve levar em conta possíveis problemas de segurança.

Neste trabalho, abordamos especificamente o projeto de protocolo na camada MAC. As tecnologias principais utilizadas para o controle de acesso ao meio em redes *ad hoc* são o padrão IEEE 802.11 (WiFi), o padrão IEEE 802.15 (Bluetooth, Zigbee, UWB, etc.), o padrão IEEE 802.16 (Broadband Wireless), entre outras tecnologias. Todas essas tecnologias citadas usam o paradigma *iniciado pelo transmissor*, no qual a negociação da conexão entre os nós é iniciado pelo remetente dos dados. Este paradigma já vem sendo amplamente utilizado nas últimas décadas e está bem consolidado. Porém, a premissa de que o protocolo MAC deve permitir um acesso justo ao canal não é satisfatória para o caso do paradigma iniciado pelo transmissor. Nesta questão, a utilização de um paradigma *iniciado pelo receptor* pode ser mais apropriada já que a distribuição do acesso ao canal é ponderada entre vários fluxos de dados oriundos dos vizinhos de um dado nó (dá mais oportunidades de acesso aos demais fluxos), enquanto que no paradigma iniciado pelo transmissor cada fluxo compete pela sua própria oportunidade de acesso. Entretanto, não há estudos mais práticos que comparem os dois paradigmas, e por isso temos como objetivo descobrir o quão eficiente pode ser, de fato, uma implementação de um protocolo MAC iniciado pelo receptor. Além disso, apesar dos ganhos de desempenho do paradigma iniciado pelo receptor em relação ao iniciado pelo transmissor relatados na literatura [14, 15], não existem muitos estudos práticos que avaliem o desempenho de protocolos MAC iniciados pelo receptor em cenários com múltiplos saltos, com transmissões concomitantes e com a presença de terminais escondidos e de terminais expostos, além de estudos em modelos analíticos [16]. Em tais cenários, deve-se levar em conta o método (ou disciplina) em que os nós são consultados pelo receptor, pois não adianta a negociação ser realizada com menos pacotes de controle se o método utilizado leva a ocorrência de várias

tentativas de conexão mal sucedidas. Por isso, a escolha de uma disciplina de consulta inadequada pode suprimir o ganho obtido na diminuição do controle. Dessa forma, nosso objetivo também é avaliar as estratégias de consulta sob o protocolo MAC iniciado pelo receptor.

1.1 Contextualização

Protocolos MAC *iniciados pelo receptor* para redes *ad hoc* sem fio têm sido estudados devido aos seus potenciais benefícios em reduzir o número de quadros de controle necessários para um estabelecimento de conexão (*handshake*). Mais importante ainda, o apelo em usar a abordagem iniciada pelo receptor vem do fato de que o *destinatário* de um quadro de DADOS está melhor posicionado para avaliar as condições do canal para uma recepção bem sucedida em um enlace de comunicação. Consequentemente, colisões de quadros no receptor intencionado podem ser potencialmente diminuidos se o próprio receptor decidir quando iniciar o recebimento de um quadro de DADOS [17]. De fato, trabalhos teóricos anteriores sugerem que protocolos MAC iniciados pelo receptor podem superar os iniciados pelo transmissor devido à redução na sobrecarga de controle [14, 15]. Idealmente, um melhor desempenho pode ser alcançado se os receptores souberem não somente *quem* tem quadros de DADOS endereçados a eles, mas também *quando* um quadro de DADOS está pronto para ser transmitido de um dado transmissor. Obviamente, isto não é uma tarefa trivial a ser cumprida na prática.

Parte deste esforço já tem sido iniciada em aplicações onde alguma sincronização temporal entre os nós é possível, especialmente em protocolos baseados em "ciclos de trabalho" (*duty-cycle*) ou em múltiplo acesso por divisão de tempo (TDMA, do inglês, *time division multiple access*), para redes de sensores, onde alguns nós agem como agregadores (*sink*) de dados coletados por outros nós [18, 19, 20]. Em tais cenários, é possível ter nós agregadores decidindo *quem* e *quando* eles se comunicam baseando-se na informação entregue pelos nós sensores em intervalos de tempo (*slots*) anteriores. Entretanto, pelo fato de que: *i*) protocolos MAC baseados em TDMA (e seus variantes) requerem rigorosa sincronização temporal; *ii*) o agendamento ótimo de *slots* em cenários de múltiplos saltos é um problema NP-difícil [21, 22]; *iii*) há outros tipos de redes *ad hoc* que não contêm nós "especiais" que agem por conta de outros nós, isto é, todos os nós são considerados *igualmente* importantes, a adoção de um protocolo MAC de *acesso aleatório* se torna a opção mais viável para uma rápida e escalável implementação de rede.

1.2 Definição do problema e Objetivos da Dissertação

Atualmente, a *iniciativa pelo transmissor* tem sido o paradigma preferido para protocolos MAC de acesso aleatório, especialmente depois do enorme sucesso do padrão IEEE 802.11 nas últimas décadas. Além disso, o paradigma iniciado pelo receptor não se encaixa tão naturalmente como o iniciado pelo transmissor em respeito ao paradigma "armazenar-e-enviar" adotado pela maioria das arquiteturas de rede para atividades de roteamento. Juntamente com a carência de estudos em disciplinas de consulta adequadas para os vários cenários de aplicação de redes *ad hoc*, protocolos

MAC iniciados pelo receptor com acesso aleatório ainda não são amplamente adotados hoje.

Assim, a fim de contribuir para o desenvolvimento (e entendimennto) das disciplinas de consulta para protocolos MAC iniciados pelo receptor com acesso aleatório, este trabalho investiga o desempenho de três disciplinas de consulta quando aplicadas a um protocolo MAC iniciado pelo receptor para comunicação ponto a ponto *unicast* específico apresentado anteriormente por Bonfim e Carvalho [23]. O protocolo MAC proposto é baseado na reversão do algoritmo de recuo exponencial binário (BEB, do inglês, *binary exponential backoff*) do padrão IEEE 802.11 como um meio de controlar a taxa em que um nó consulta seus vizinhos. De fato, uma importante questão de um protocolo MAC iniciado pelo receptor é sua *taxa de consulta*, porque uma taxa de consulta que é muito baixa leva a uma baixa vazão e longos atrasos, enquanto que uma taxa de consulta que é muito alta pode resultar em um alto número de colisões de quadro, que também resulta em um mal desempenho de rede. Usando uma versão reversa do algoritmo BEB do IEEE 802.11, a taxa de transmissão dos quadros de consulta é auto-regulada de acordo com a contenção do canal, condições de propagação do sinal, e disponibilidade de tráfego nos nós consultados. Além disso, é proposta uma disciplina adaptativa de consulta que controla a *prioridade* com a qual vizinhos são consultados baseada na probabilidade do estabelecimento de uma conexão bem sucedida (LSH, do inglês, likelihood of successful handshake). Este protocolo MAC é denominado como *Receiver Initiated with Binary Exponential Backoff* (RIBB).

Este trabalho também apresenta duas importantes extensões ao RIBB: primeiro, propõe-se uma técnica de *reordenamento de quadro* nas filas de transmissão. De acordo com este mecanismo, toda vez que um nó é consultado por alguém, o nó deve procurar por um quadro de DADOS endereçado ao nó consultante em *toda* sua fila de transmissão. Dessa forma, o processo de consulta não é desperdiçado simplesmente porque não há um quadro de DADOS endereçado ao nó consultante na cabeça da fila. Segundo, um quadro de controle *nothing-to-send* (NTS, nada-a-enviar) é apresentado. O papel deste quadro de controle é deixar o nó consultante saber que não há quadros de DADOS endereçados a ele em *toda* a fila de transmissão do nó consultado. Fazendo isso, o envio de um quadro NTS agiliza o processo de consulta deixando o nó consultante escolher outro vizinho para consulta assim que possível.

O desempenho do RIBB é avaliado baseado em simulações a eventos discretos a partir de sua implementação no Network Simulator 3 [24]. Seu desempenho é também comparado ao desempenho de outras duas disciplinas de consulta (aplicadas ao mesmo mecanismo de taxa de consulta): o primeiro é o simples mecanismo de consulta cíclica (*round-robin*), que denominamos como *Receiver Initiated Round Robin* (RIRR), e o outro é baseado no agendamento de justiça proporcional (*proportional fair*) usado em redes 4G, que denominamos como *Receiver Initiated Proportional Fair* (RIPF). Todas as três disciplinas de consulta são avaliadas em relação à sobrecarga de controle, atraso, justiça, e vazão (todas em nível MAC), e seus desempenhos são também comparados ao padrão IEEE 802.11 DCF iniciado pelo transmissor. Dois diferentes cenários de tráfego são considerados sob topologias de rede com diferentes esparsidades (isto é, diferentes graus de conectividade).

Baseado nos resultados obtidos, propomos o uso de um mecanismo de consulta adaptativo que

seleciona dinamicamente a disciplina de consulta de acordo com a contenção do canal e com a qualidade do enlace, denominado de *Receiver Initiated MAC with Adaptive Polling Discipline* (RIMAP), um protocolo MAC de acesso aleatório para comunicação ponto a ponto. O procedimento de comutação ajusta o compromisso entre maximizar vazão e justiça. Resultados de simulação mostram que é possível alcançar melhor desempenho de justiça sem muita perda na vazão, além de melhorar o atraso de pacote em relação ao protocolo iniciado pelo transmissor (representado pelo IEEE 802.11).

1.3 Contribuições

- Uma implementação sem precedente do protocolo MAC iniciado pelo receptor apresentado por Bonfim em um simulador a eventos discretos bem conhecido e de código livre, o Network Simulator 3 (NS-3);
- Uma extensão deste protocolo apresentando um novo quadro de controle, o "Nada-a-Enviar" (NTS, do inglês, Nothing-to-Send), que ajuda a mitigar pedidos inúteis de transmissão e melhorar a utilização do canal;
- A incorporação do mecanismo de reordenamento de quadro na fila de transmissão MAC no protocolo extendido;
- Um melhoramento na disciplina de consulta proposta por Bonfim no protocolo RIBB, alterando o cálculo da estimação das probabilidades;
- A investigação de três disciplinas de consulta em relação à sobrecarga de controle, à justiça, ao atraso, e à vazão, sob diferentes condições de rede;
- A avaliação do desempenho do protocolo MAC extendido proposto e a comparação ao protocolo IEEE 802.11b iniciado pelo transmissor;
- Baseada nesta avaliação, a proposta de um mecanismo adaptativo que seleciona a disciplina de consulta de acordo com as condições da rede;
- A avaliação do desempenho do mecanismo adaptativo proposto.

1.4 Apresentação da Dissertação

Primeiramente, apresentamos no Capítulo 3 os trabalhos relacionados mostrando o histórico dos protocolos MAC iniciados pelo receptor, buscando mostrar como os mecanismos de disciplina de consulta e de controle da taxa de consulta têm sido tratados na literatura, para então apresentar as lacunas a serem preenchidas com o nosso trabalho. Em seguida, no Capítulo 4 descrevemos as especificações do protocolo MAC iniciado pelo receptor, as funcionalidades de suas características, e ilustramos o processo de estabelecimento de conexão entre os nós iniciado a partir do receptor. A decisão do receptor em escolher o alvo de sua conexão é realizada pelo mecanismo da disciplina

de consulta, para a qual são investigados três tipos de disciplinas e apresentadas na sequência no Capítulo 5. Posteriormente, no Capítulo 6 são apresentados os cenários de simulação utilizados nas avaliações de desempenho dos protocolos RIBB, RIRR e RIPF, e os resultados numéricos com respeito ao desempenho da sobrecarga de controle, atraso de pacote, justiça e vazão são exibidos. A partir destes resultados, propomos uma disciplina de consulta adaptativa, e descrevemos seu mecanismo atuando sobre o protocolo MAC iniciado pelo receptor no Capítulo 7. Então, apresentamos os resultados numéricos das simulações utilizadas para a avaliação de desempenho. Finalmente, concluimos no Capítulo 8 a respeito das contribuições do trabalho, refletindo sobre a aplicação do protocolo e possíveis trabalhos futuros.

Chapter 2

Introduction

In the last years, there was a quick expansion of the mobile computing industry due to its popularization and the more friendly interfaces between the device and the user. However, the present wireless communication devices, applications and protocols are mostly designed for using in cellular networks and wireless local area networks, disregarding the great potential offered by ad hoc networks. An ad hoc network is an autonomous set of mobile devices (tablets, smartphones, sensors, etc.) that communicate with each other over wireless links and cooperate in a distributed manner by forwarding packets and by routing activities executed by all devices, in order to provide the necessary network functionality in the absence of a fixed infrastructure [1].

The ad hoc devices are generally powered by battery (energy restriction), they can provide several types of functionalities (heterogeneity), they can associate and disassociate from the network freely, so that they can move randomly (dynamic topology), and they can organize themselves dynamically in order to deploy a functional network in the absence of centralized administration (autonomy). However, a wireless ad hoc network faces the same traditional problems inherent to the wireless communications, e.g., lower confiability with respect to a wired medium, limited physical layer security, time varying channel, interference, etc. But, despite the various restrictions, the wireless ad hoc networks are highly satisfactory for using in situation where a fixed infrastructure is absent, non-trustable, or very expensive. Due to its capacity of self-criation, self-organization, and self-administration, ad hoc networks may be rapidly deployed with minimum user intervention, not requiring a detailed planning of base stations or cabling systems.

As consequence, there is an expectation that ad hoc networks become an important part of the future architecture of next generation networks (4G, 5G...), that aim to provide an environment that gives support to the user to perform its task, accessing the information and communicating at any time, any place, and from any device [2, 3]. In this context, there is a wide range of applications to be used in ad hoc networks in many areas:

- Tatical networks: militar operations and communications, automated battlefields [4];
- Emergency services: search and rescue operations, disaster recovery [5];
- Vehicular services: road or accident guidance, transmission of road and weather conditions,

taxi cab network, inter-vehicle networks [6];

- Sensor networks: house sensors (smarthomes), environmental tracking, fauna monitoring [7];
- Entertainment: multi-user games [8], peer-to-peer communication (P2P) [9], mobile social networks [10];
- Coverage extension: network access extension (*onloading*) [11], data *offloading* from the cellular infrastructure by the mobile devices [12], opportunistic networks [13].

The specific characteristics of ad hoc networks impose many challenges to network protocol design on all layers of the protocol stack. The physical layer must deal with rapid changes in link characteristics. The medium access control (MAC) layer must allow fair channel access, minimize packet collisions and deal with hidden and exposed terminal. At the network layer, nodes need to cooperate to calculate paths. The transport layer must be capable of handling packet loss and delay characteristics that are very different from wired networks. Applications must be able to manage possible disconnections and reconnections. Furthermore, the development of protocols must take into account possible security problems.

In this dissertation, we address specifically the project of protocol in MAC layer. The main technologies utilized for the medium access control in ad hoc layers are the standards IEEE 802.11 (WiFi), IEEE 802.15 (Bluetooth, Zigbee, UWB, etc.), IEEE 802.16 (Broadband Wireless), and other technologies. All these cited technologies use the *sender-initiated* paradigm, in which the handshake between the nodes is initiated by the sender of the data. This paradigm has already been widely employed in the last decades and it is well consolidated. But the assumption that the MAC protocol must allow a fair access to the channel is not satisfactory for the case of sender-initiated paradigm. In this issue, the utilization of a *receiver-initiated* paradigm may be more appropriate since the distribution of the channel access is weighted among the multiple data flows from the neighbors of a given node (it gives more opportunities to access to other flows), while in sender-initiated paradigm each flow competes for its own opportunity to access. However, there is not many practical studies that compares both paradigms, and that is why we have as objective to discover how efficient, in fact, the receiver-initiated paradigm performance could be. Furthermore, despite the receiver-initiated paradigm performance gains with respect to the sender-initiated reported in literature [14, 15], there is not many works that evaluates the performance of receiver-initiated MAC protocols in multi-hop scenarios, with concurrent transmissions, and hidden and exposed terminals, besides studies with analytical models [16]. In such scenarios, one must take into account the method (or discipline) in which the nodes are polled by the receiver, because it is pointless to establish a connection with fewer control packets if the utilized method leads to the occurrence of various unsuccessful conection attempts. Hence, the choice of an unappropriate polling discipline may suppress the gains obtained in diminishing the control. Thus, our objective is also to evaluate the strategies of polling under the receiver-initiated MAC protocol.

2.1 Contextualization

Receiver-initiated MAC protocols for wireless ad hoc networks have long been studied due to their potential benefits in reducing the number of control frames needed for a handshake. Most importantly, the appeal for using a receiver-initiated approach comes from the fact that the *recipient* of a DATA frame is certainly better positioned to evaluate channel conditions for a successful DATA frame reception in a communication link. Consequently, frame collisions at the intended receiver may be potentially diminished if the receiver itself is the one who decides when to start receiving a DATA frame [17]. In fact, previous theoretical works have suggested that receiver-initiated MAC protocols may overcome sender-initiated ones due to a reduction in control overhead [14, 15]. Ideally, best performance could be achieved if receivers were able to know not only *who* has DATA frames addressed to them, but also *when* a DATA frame is ready to be sent from a given transmitter. Obviously, this is not a trivial task to accomplish in practical scenarios.

Part of this effort has already started in applications where some time synchronization among nodes is possible, especially in duty-cycle or TDMA-based MAC protocols for sensor networks, where some nodes act as *sinks* for data collected by other nodes [18, 19, 20]. In such scenarios, it is possible to have *sink* nodes deciding *whom* and *when* they contact based on information delivered by sensor nodes in previous slot(s). However, because *i*) TDMA-based MAC protocols (and their variants) strictly require time synchronization; *ii*) the optimal scheduling of slots in multihop scenarios is an NP-hard problem [21, 22]; *iii*) there are other types of ad hoc networks that do not have "special" nodes that act on behalf of other nodes, i.e., all nodes are considered *equally* important, the adoption of a *random access* MAC protocol becomes the most viable option for a fast and scalable network deployment.

2.2 Problem Definition and Dissertation Objectives

To date, *sender initiation* has been the preferred paradigm for random access MAC protocols, especially after the tremendous success of the IEEE 802.11 standard in the last decades. In addition, the receiver-initiated paradigm does not fit as naturally as the sender-initiated one with respect to the "store-and-forward" paradigm adopted by the majority of network architectures for routing activities. Coupled with the lack of studies on polling disciplines suitable for the many application scenarios of ad hoc networks, random-access receiver-initiated MAC protocols are still not widely adopted today.

Hence, in order to contribute for the development (and understanding) of polling disciplines for random-access receiver-initiated MAC protocols, this work investigates the performance of three polling disciplines when applied to a specific receiver-initiated *unicast* MAC protocol earlier introduced by Bonfim and Carvalho [23]. The proposed MAC protocol is based on reversing the binary exponential backoff (BEB) algorithm of the IEEE 802.11 as a means to control the *rate* at which a node polls its neighbors. In fact, an important issue of a receiver-initiated MAC protocol is its *polling rate*, because a polling rate that is too low renders low throughput and long delays,

whereas a polling rate that is too high may result in a high number of frame collisions, which also results in poor network performance. By using a reversed version of the IEEE 802.11 BEB algorithm, the transmission rate of polling frames is self-regulated according to channel contention, signal propagation conditions, and traffic availability at polled nodes. Moreover, an adaptive *polling discipline* is also proposed, that controls the *priority* with which neighbors are polled based on the likelihood of a successful handshake. We name this MAC protocol as *Receiver Initiated with Binary Exponential Backoff* (RIBB).

This work also introduces two important extensions to RIBB: first, a *frame reordering* technique at transmit queues is proposed. According to this mechanism, every time a node is polled by someone, it has to look for a DATA frame addressed to the polling node in its *whole* transmit queue. This way, the polling process is not wasted simply because there is no DATA frame addressed to the polling node at the head of the queue. Second, a *nothing-to-send* (NTS) control frame is introduced. The role of this control frame is to let the polling node know that there is no DATA frame addressed to it in the *whole* transmit queue of the polled node. By doing so, the sending of an NTS frame speeds up the polling process by letting the polling node switch to another neighbor as fast as possible.

The performance of RIBB is evaluated based on discrete-event simulations using the popular Network Simulator 3 [24]. Its performance is also compared to the performance of two other polling disciplines (applied to the same polling rate mechanism): the first is the plain *round-robin* mechanism, which we name it as *Receiver Initiated Round Robin* (RIRR), and the other is based on the proportional fair scheduling used in 4G networks, which we name it as *Receiver Initiated Proportional Fair* (RIPF). All three polling disciplines are evaluated with respect to MAC-level control overhead, delay, fairness, and throughput, and their performance is also compared to the sender-initiated IEEE 802.11 DCF. Two different traffic scenarios are considered under network topologies with different sparsity levels (i.e., different degrees of connectivity).

Based on the obtained results, we propose the utilization of an adaptive polling mechanism that dynamically selects a polling discipline according to channel contention and link quality, denomiminated as Receiver Initiated MAC with Adaptive Polling Discipline (RIMAP), a random access unicast MAC protocol. The switching procedure tunes the trade-off between maximizing throughput and fairness. Simulation results show that it is possible to achieve better fairness performance without much loss in throughput, in addition to improving packet delay with respect to sender-initiated protocol (represented by IEEE 802.11).

2.3 Contributions

- An unprecedented implementation of the receiver-initiated MAC protocol introduced by Bonfim in the well-known open-source discrete-event simulator, the Network Simulator 3 (NS-3);
- An extension of this protocol by introducing a new control frame, the Nothing-To-Send (NTS), which helps mitigating useless requests for transmission and improve channel utilization;

- The incorporation of the mechanism of frame reordering in the MAC transmission queue in the extended protocol;
- An enhancement of the polling discipline proposed by Bonfim in RIBB protocol, by changing the computation of the probabilities estimation;
- The investigation of three different polling disciplines with respect to control overhead, fairness, delay, and throughput, under different network conditions;
- The evaluation of the performance of the proposed extended MAC protocol and the comparation to the sender-initiated protocol IEEE 802.11b;
- Based on this evaluation, a proposition of a adaptive mechanism that selects a polling discipline according to the network conditions;
- The evaluation of the performance of the proposed adaptive mechanism.

2.4 Dissertation Organization

First, we present in Chapter 3 the related works showing the history of receiver-initiated MAC protocols, indicating how the polling discipline and polling rate control mechanisms have been addressed in the literature, and then presenting the gaps to be filled by our work. Next, in Chapter 4 we describe the specifications of the receiver-initiated MAC protocol, the functionalities of the features, and we ilustrate the handshake process between the nodes initiated by the receiver. The decision of the receiver on choosing the target of its handshake is performed by the polling discipline mechanism, for which we investigate three types of disciplines and we present them in Chapter 5. Subsequently, in Chapter 6, we present the simulation scenarios utilized in the performance evaluation of protocols RIRR, RIPF, and RIBB, and the numerical results with respect to control overhead, packet delay, network fairness, and flow throughput are exhibited. From these results, we propose an adaptive polling discipline, and we describe its mechanism acting over the receiver-initiated MAC protocol in Chapter 7. Then, we present the numerical results of the simulations utilized for the performance evaluation. Finnaly, we conclude in Chapter 8 with respect to the contributions of the work, reflecting on the protocol application and possible future works.

Chapter 3

Related Work

In this chapter, we describe previous works carried out in the context of receiver-initiated MAC protocols. In particular, we look at how polling disciplines and polling rate control mechanisms have been treated in the literature. The first receiver-initiated MAC protocol proposed in the literature was the MACA By Invitation (MACA-BI) [14]. This work introduced the appealing features of such a strategy, which reduces the number of control frames used in a handshake by placing the responsibility of communication on the potential receiver of a DATA frame. In MACA-BI, a node polls some neighbor by sending a *ready-to-receive* (RTR) control frame. If the RTR is received successfully, the polled node may send a DATA frame back to the polling node if there is a head-of-line DATA frame addressed to it. If the DATA frame is received successfully, the polling node sends an acknowledgment (ACK) frame back to the polled node.

Later, Tzamaloukas and Garcia-Luna-Aceves [15] have shown that MACA-BI cannot ensure perfect collision avoidance in networks with hidden terminals, and they have proposed the Receiver-Initiated Multiple Access (RIMA) protocol. RIMA avoids hidden terminals with the use of a *No-Transmission-Request* (NTR) control frame. This control frame has the job of telling the polled node not to send any DATA frame after sensing channel activity in the end of an RTR transmission. RIMA performance was evaluated based on the assumption that polling rates were governed by a Poisson process (i.e., exponentially-distributed polling intervals), and polled nodes were chosen randomly, with equal probability. Fully-connected (i.e., single-hop) scenarios and perfect channel conditions were assumed for analysis.

Dhananjay-Lal *et al.* [25] have proposed a receiver-initiated MAC protocol that exploits space-division multiple access, where directional reception is used to receive more than one packet from spatially-separated transmitting nodes. In this protocol, the receiver-initiated paradigm is used as a means to synchronize neighboring nodes involved in a packet transmission. However, in this work, no mention is made to the polling discipline. Furthermore, it employs a strategy of independent polling rate, i.e., the polling rate is independent of network traffic. Later, Yi-Sheng Su *et al.* [26] have proposed MAC protocols for mobile ad hoc networks that apply the receiver-initiated concept with spread-spectrum technology. They have proposed two hybrid handshake schemes: the RIMA Common-Transmitter-Based (RIMA/C-T) and the RIMA Receiver-Transmitter-Based (RIMA/R-

T). In their work, however, nothing is mentioned regarding the polling discipline or polling rate control mechanisms. Takata *et al.* [27] have proposed the Receiver-Initiated Directional MAC (RI-DMAC) to address the issue of deafness in directional MAC protocols. They use a combination of receiver-initiated and sender-initiated protocols, where the sender-initiated approach is the default operation mode, while the receiver-initiated mode is triggered when the transmitter experiences deafness. Regarding the polling discipline, each node in RI-DMAC must maintain a polling table to poll only the potentially deaf nodes. However, regarding the polling rate control, just a single polling attempt is performed after each sender-initiated handshake attempt.

Sun *et al.* [18] have proposed an asynchronous duty cycle MAC protocol for wireless sensor networks, the Receiver-Initiated MAC (RI-MAC). It employs receiver-initiated transmissions to avoid energy waste due to the "idle listening" problem. Because their receiver-initiated design is centered on asynchronous duty cycle, RI-MAC may reduce overhearing substantially, while also achieving low collision probability and recovery cost. Inspired by RI-MAC, Qian Hu *et al.* [20] have proposed the Reordering Passive MAC (RP-MAC), a duty-cycle MAC protocol for wireless sensor networks. RP-MAC has better energy efficiency than RI-MAC because the sender, after receiving a DATA frame from upper layers, sleeps until it is awakened by the DATA receiver. In both protocols, there is a polling discipline where a DATA receiver sends a *broadcast* beacon message to request a DATA frame from its neighbors. Potential DATA transmitters must contend for channel access after hearing the broadcast message. All DATA senders must use a binary exponential backoff algorithm to contend for the channel, and the backoff window size is announced in the broadcast message sent by the DATA receiver. In the event of DATA frame collisions, the DATA receiver increases the backoff window size and send the information in the next beacon message. The polling rate is controlled by the duty cycle activity. Another feature of RP-MAC is the frame reordering (FR) mechanism: whenever the active time of any node arrives, the node searches its queue for the first DATA frame addressed to the polling node. Thus, the FR scheme can improve the frame delivery efficiency. The performance of RP-MAC is compared to RI-MAC with respect to latency and energy efficiency. Throughput and fairness are not evaluated, and the network topology considered for analysis is a simple star topology, consisting of only 11 nodes, with all data flows passing through the center node. Therefore, the impact of interference due to concurrent polling activities from other polling nodes in the terrain, and corresponding DATA transmissions from other DATA senders, is not considered in this work.

Dutta *et al.* [28] presents a receiver-initiated link layer for low-power wireless networks that supports several services under a unified architecture. In spite of offering support to extremely low duty cycles or high data rates, no reference is made regarding the adopted polling discipline. In addition, the control of the polling rate is related to the duty cycle activity only, disregarding thus, other network parameters. Recently, Liang and Zhuang [29] have proposed a MAC protocol for delay tolerant networks (DTNs) via roadside wireless local area networks (RS-WLANs), the Double-Loop Receiver-Initiated MAC (DRMAC). This protocol aims at resolving channel contention among multiple direct/relay links and exploits the predictable traffic characteristics of this scenario as a result of packet pre-downloading. The receiver-initiated mechanism is used to reduce the signalling overhead, where the ACK message is used as an invitation for channel contention.

Given that traffic characteristics are predictable, the polling rate and discipline can be adjusted adaptively according to the given scenario.

Zhi Ang Eu and Hwee-Pink Tan [19] have proposed the Energy Harvesting MAC protocol (EH-MAC) for multi-hop energy harvesting wireless sensor networks (EH-WSN). As the node requires the store of energy before starting any activity, the polling rate of this protocol matches the energy harvesting rate, so that, at the end of a charging state, a node sends a polling packet after some random time. Regarding the polling discipline, the polling packet is broadcasted and, for each polling node, the polling packet is associated with a contention probability, p_c, which is used to indicate the probability that a sender should transmit its data packet. So, every node that listens to the polling packet (and contains DATA frames addressed to the polling node) transmits its DATA frame with the given probability. Leonardi *et al.* [30] have proposed a protocol that can be considered as a hybrid solution, the Carrier Sense Multiple Access with Collision Avoidance by Receiver Detection (CSMA/CARD). It is both sender-and-receiver-initiated, where each receiver can predict the existence of a potential sender in a timely manner. The approach uses events occurring at the physical layer, and may interpret significant received signal power variations (probabilistically) as handshake messages initiated by a potential sender. The receiver then reacts accordingly by anticipating a handshake. The receiver reaction depends on whether the received signal power variation is a decodable RTS (received in NAV period) or non-decodable RTS (received in DATA reception) in order to know if the RRTS (request-for-RTS) frame is addressed as unicast (decodable RTS) or broadcast (non-decodable). In this case, the receiver cannot decode the RTS sender address, and the potential senders have to contend for channel access, sending the RTS again. Therefore, regarding the polling rate control mechanism, a polling attempt is performed only after a sender-initiated handshake attempt.

Lina Pu *et al.* [31] have proposed a traffic estimation-based receiver initiated MAC (FERI MAC) for underwater acoustic networks (UANs) to mitigate the problem of high overhead of control messages due to the long preamble problem. FERI MAC hava a data polling mechanism conditioned by the power consumption of control packets, and by the queueing delay for a packet awaiting for transmission. Thus FERI MAC can achieve an user-desired energy efficiency by adjusting the data polling frequency. Also, the protocol uses a traffic prediction-based on an adaptive data polling approach to estimate how much data to request from each sender. Then, the protocol achieves a trade-off between channel utilization and packet delivery delay, adjusting the amout of packets to poll. The performance of FERI MAC is evaluated with respect to energy efficiency, channel utilization and one-hop delivery delay. Throughput and fairness are not evaluated, and the network topology considered for analysis consists of eight nodes only deployed over a ring with about 1 km average distance between neighbor nodes.

Although previous works have considered receiver-initiated MAC protocols in different forms, to date, there is no coherent understanding of the effect of different polling mechanisms on overall network performance. Moreover, it has been observed that a recent trend in MAC protocols targeted at sensor networks is the fact that the polling packets are *broadcast* messages, instead of *unicast* messages. In this work, we want to investigate polling disciplines applied to *unicast polling* because in a random access scenario, a broadcast (or multicast) polling implies that the

potential transmitters must content for the channel, even after the receiver had already contended for polling. Thus, in order to avoid doubled channel contentions for a single handshake, unicast polling is adopted for random access networks, aided by the polling discipline. Moreover, we want to investigate the impact of different polling strategies under concurrent polling activity, where nodes are scattered in the terrain according to different sparsity levels, as opposed to the majority of previous works who have considered single-hop (fully-connected) networks. Furthermore, we focus on *unicast polling* for a general-purpose receiver-initiated MAC protocol that adaptively uses two polling disciplines that can be tuned to trade off fairness with throughput-delay performance.

Chapter 4

Protocol Description

In this chapter, we specify the details of the extended random-access receiver-initiated unicast MAC protocol that is used to investigate the polling disciplines. This protocol follows the work by Bonfim and Carvalho [23], who have proposed a reversed version of the IEEE 802.11 DCF binary exponential backoff (BEB) algorithm as a means to control the rate at which a node polls its neighbors. In their work, they have used an analytical model to evaluate the *steady-state* behavior of saturated, *not* fully-connected networks under channel propagation effects. In this work, we extend the protocol by including three new enhancements: a frame reordering technique, a new *nothing-to-send* (NTS) control frame, and an improved polling discipline (to be introduced in Chapter 5). In addition, we address some of the issues related to queue management and how neighborhood tables are built and maintained, which are important for actual protocol implementation and operation (as opposed to mathematical abstractions for modeling and analysis).

4.1 Binary Exponential Backoff (BEB) Algorithm

A key component of the proposed receiver-initiated MAC protocol is a *reversed* version of the binary exponential backoff (BEB) algorithm of the IEEE 802.11 DCF: following the polling discipline in place, every node picks a neighbor from its neighborhood table and executes the BEB algorithm in order to control the rate at which they poll the selected neighbor. The idea of using the BEB algorithm stems from the fact that, when polling a node, the RTR may not be received (or replied) due to numerous reasons, such as: *i*) the RTR is received with errors due to channel impairments or frame collisions; *ii*) the polled node does not contain a DATA frame addressed to the polling node; *iii*) the DATA frame is received with errors at the polling node. Since this is a *unicast* MAC protocol, a single polling attempt may not be enough for successful communication due to the aforementioned reasons. Therefore, a finite number of successive attempts should be encouraged, spaced by random time intervals dictated by the BEB algorithm, so that channel contention is alleviated and other nodes may have access to the channel as well. After a successful transmission or a finite number of failed attempts, the node picks another neighbor to poll from its neighborhood table, according to the adopted polling discipline.

Similar to the IEEE 802.11 DCF, the BEB algorithm uses a discrete-time backoff timer. Before transmission of each RTR, a backoff time is uniformly chosen in the interval $[0, W - 1]$. The integer value W is denoted as the *contention window size*, and it depends on the number of transmission attempts for the specific RTR frame, as illustrated in Figure 4.1, i.e., for each new RTR, the contention window size W takes an initial value $W_{\min}$ that doubles after each unsuccessful RTR transmission (for the given target node), up to a maximum $W_{\max}$. After reaching $W_{\max}$, it remains at this value until it reaches the maximum number of transsmission attempts `maxSSRC` (maximum station short retry count). As depicted in Figure 4.2, the backoff timer is decremented only when the medium is sensed idle, and it is frozen when the medium is sensed busy. After a busy period, the decrementing of the backoff timer resumes only after the medium is sensed idle longer than a DIFS time interval (the same length as the IEEE 802.11 DIFS). Then, the RTR is transmitted when the backoff timer zeroes out. While not transmitting an RTR, and during the backoff operation, the node may be polled by someone else, in which case it freezes the backoff operation to reply to the received RTR by sending a DATA frame (if any) addressed to the polling node. Next, we specify how the protocol initiates, including how the neighborhood table is built and maintained.

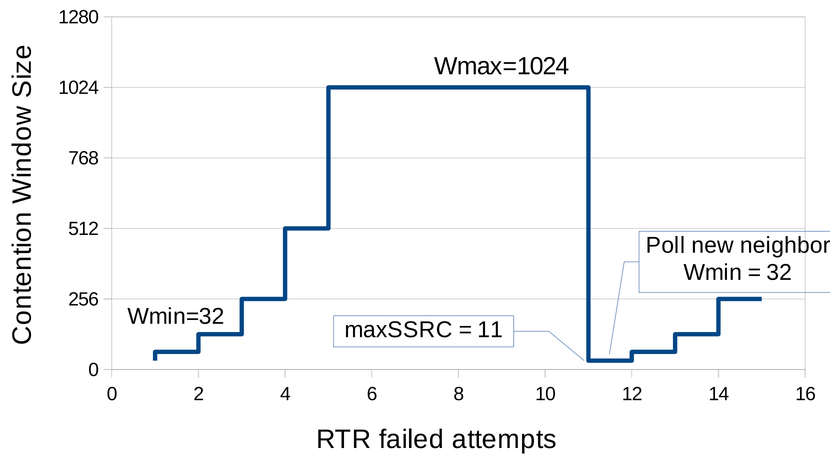


Figure 4.1: Example of backoff contention window growth.

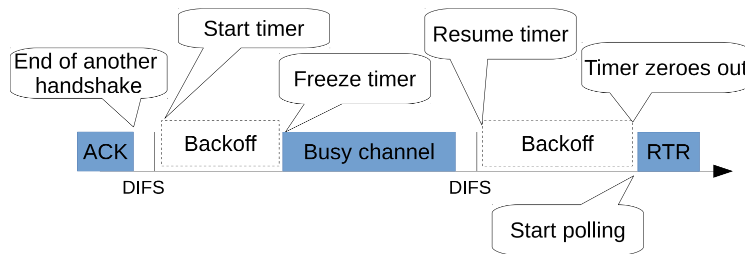


Figure 4.2: Example of backoff timer decrement.

4.2 Initialization

When the station is turned on, it immediately starts backing off before accessing the channel, as illustrated in Figure 4.3. As soon as the first backoff stage ends, and the station is allowed to transmit, the station checks whether there is a packet at the head of its queue. If there is any, and it is a *broadcast* MAC frame, it is dequeued and transmitted, since it may carry important information regarding this node, such as routing control messages. Otherwise, if it is a *unicast* MAC frame, it *stays in queue*, and the node starts the polling process for packets from its neighbors (the packet left at head of the queue is a packet that needs to be polled by someone else). Therefore, the node starts its own polling process with the transmission of an RTR frame.

At first, while a given node does not know any MAC address of its neighbors, the RTR frame uses a *broadcast* MAC address as destination, and whoever hears the RTR broadcast message does not reply, in order to avoid frame collisions. This procedure acts as an initial "hello" frame to neighboring nodes, so they can add the *source* MAC address in this frame to their *neighborhood tables*. As usual, the neighborhood table is a list of neighbors' MAC addresses, with an expiration time associated to each of its entries. An entry is removed from the table if no frame is heard from that particular node before time is up. It is expected that, as time goes by, all node's neighbors also initiate their backoff algorithm and start sending MAC frames, which will allow this node to fill out its neighborhood table, too. A node can start sending unicast RTR frames to specific nodes after acquiring the address of at least one neighbor. In fact, in order to quickly populate the neighborhood table and keep it updated, every MAC frame heard by a node is used to fill out its neighborhood table.

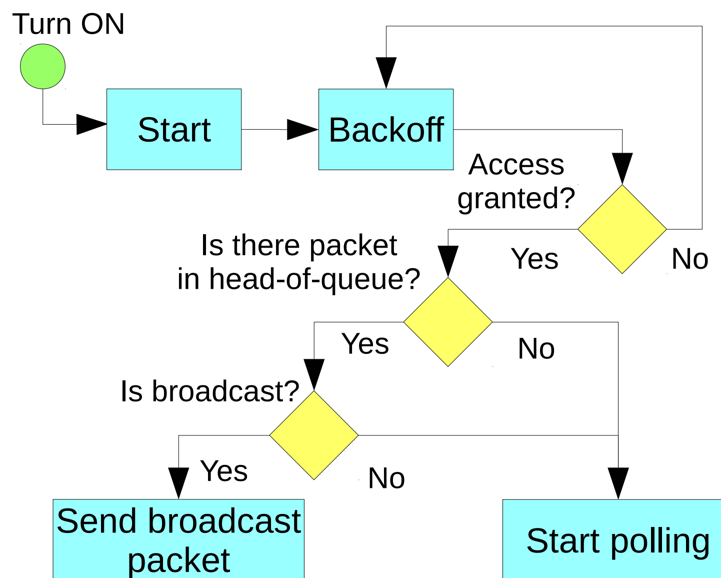


Figure 4.3: Receiver Initiated MAC initialization flowchart.

4.3 Frame Reordering

The DATA frames in a MAC transmission queue are usually destined to various nodes. And the network throughput and delay may be reduced tremendously if the sender sends those frames with the original order strictly [20], as one can verify in the following example. We assume:

- There are two DATA frames in node A's queue. And their destinations are as follows: node B and node C.
- Node B's probability of successful polling is lower than node C's (node B may be more distant from node A than node C are, thus node B needs more RTR retransmissions before polling successfully), i.e., node C accesses the channel more often than node B according to BEB algorithm.
- Node A will send the frames according to the FIFO rule.

Now, the polling attempt is from node C to node A. Although there is a DATA frame for node C in the queue, node A will not let it out because it is waiting for the polling from node B. Upon receiving the expected polling, node A starts to transmit the pending DATA frame to node B. However, due to bad channel conditions, the transmission may fail, leading to node B backs off while increasing its contention window. After a longer waiting, node A receives another polling from node B and, this time, the DATA frame transmission is successful (or the DATA frame is discarded because it exceed the maximum number of retransmissions). In the meantime, many polling attempts from node C were ignored by node A. But now, as the frame destined to node B has already been dequeued, node A can transmit the node C's DATA frame upon receiving the next polling from node C. The complete process is shown in Figure 4.4a. In this example, node C's DATA frame in node A's queue suffered a delay caused by node C's and node B's backoffs, and it was severely aggravated by the node B's RTR retransmissions.

In fact, the efficiency could be much higher if only node A adjusts the transmission order according to the newcomer polling. Specifically in the example, if node A could transmit node C's DATA frame at the first moment it was polled, this frame would suffer delay only caused by node C's backoff, reducing significantly the delay with respect to the previous example, as depicted in Figure 4.4b. Moreover, comparing to the sender-initiated paradigm using FIFO queue, node B's DATA frame would still be retransmitted many times before a successful transmission or its discard, thus node C's DATA frame delay would also be affected in this case. Therefore, a frame reordering technique is implemented in the receiver-initiated protocol as a necessary way of increasing its efficiency.

Assuming that the neighborhood table already contains at least one entry, the node may choose a neighbor to poll (including the last neighbor polled) according to a given polling discipline. The chosen destination MAC address is included in the RTR frame header, and the frame transmission is initiated in the end of a backoff stage (i.e., every time the backoff timer zeroes out). When the intended data source receives the RTR, it checks for the existence of *any* DATA frame addressed to the RTR sender in its *whole* queue. In other words, not only the frame at the head of the queue

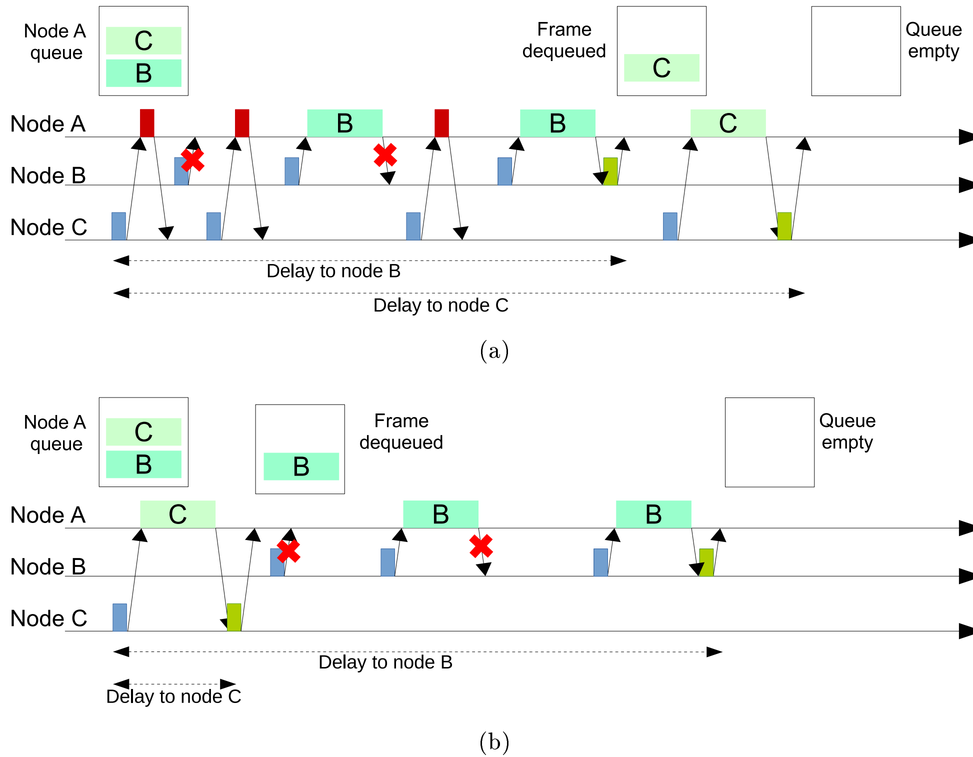


Figure 4.4: Example of frame reordering technique compared to FIFO queue. (a) FIFO operation. (b) Frame Reordering (FR). Average delay of the DATA frames is reduced with FR technique. Blue frames are polling RTR frames, red frames are NTS, green frames are ACK.

is checked, but also *all* frames stored in the MAC queue. This is to allow faster response to polling nodes. If positive, the first DATA frame found in the queue is transmitted to the sender of the RTR. After receiving a successful DATA frame, the polling node acknowledges it with the sending of an ACK frame after a SIFS time interval, as depicted in Figure 4.5. Otherwise, a *nothing-to-send* (NTS) control frame is transmitted. The NTS control frame serves the purpose of telling the polling node that there is no DATA frame addressed to it from this polled node. Such a situation is depicted in Figure 4.6. It must be stressed that the dequeued frame is not necessarily the first in queue, but it is the first frame addressed to the polling node. Thus, a frame may not need to wait for the transmission of all frames ahead of it in the queue. Consequently, the average DATA frame delay may be reduced. This idea is similar to the frame reordering concept in [20].

Given that a DATA frame may remain in queue indefinitely if its destination address is a neighbor that no longer is within reach of the node (and, therefore, it may never poll this DATA frame again either because it has left the network or it has moved away from this node), a *maximum queue delay* is set for every DATA frame in the MAC queue. Hence, if this frame is not requested by its destination node within this maximum delay, the DATA frame is dropped from the queue. Notice that, because of the frame reordering technique, this is no longer a FIFO queue, and the distribution of frames in the queue will depend on how frames arrive (or are generated) at this node, and which neighbors poll this node along the time.

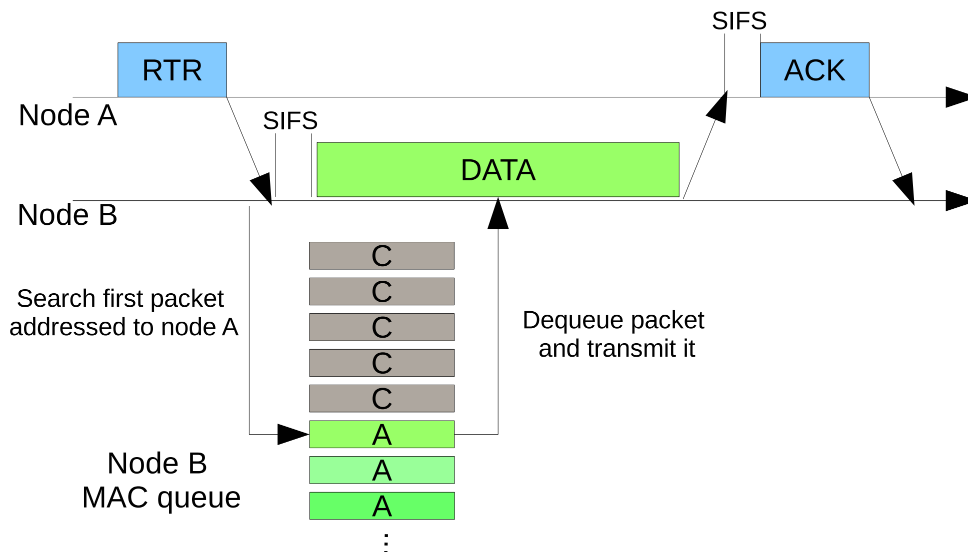


Figure 4.5: Frame reordering in MAC queue. The first packet addressed to RTR source is transmitted.

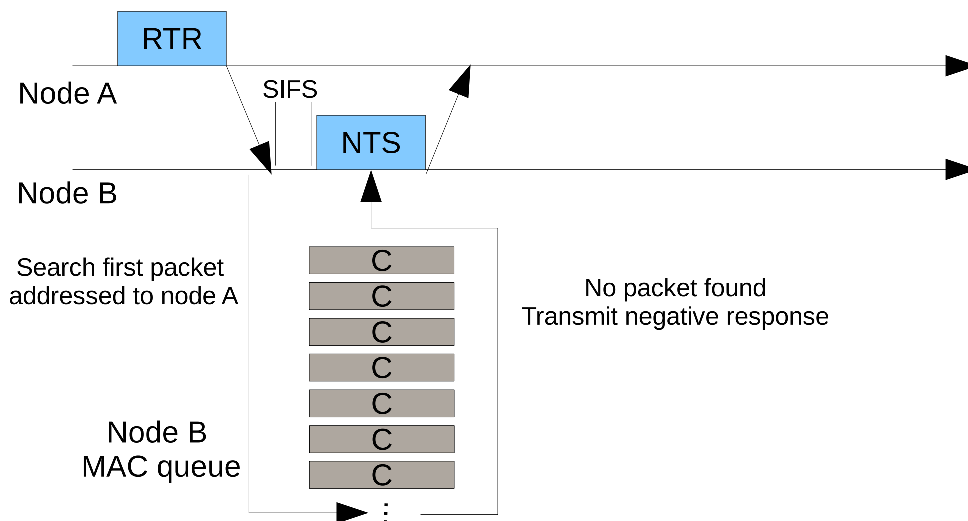


Figure 4.6: NTS transmission. Since there is no packet addressed to RTR source, a negative response is transmitted.

4.4 Virtual Carrier Sensing

Virtual carrier sensing is a mechanism adopted in CSMA/CA-like protocols. It consists in predicting the traffic on the channel depending on the time value of the duration field in control frames, in order to avoid transmission while the indicated time is up. Similar to the IEEE 802.11 DCF, a virtual carrier sensing mechanism is also adopted for the sake of collision avoidance. Hence, like the CTS frame in the IEEE 802.11 DCF, the sending of an RTR frame serves the purpose of reserving the channel around the receiver before reception of a DATA frame. All frames (RTR, NTS, DATA, ACK) carry timing information to update the NAV (Network Allocation Vector) of

neighboring nodes. The computation of the channel time to be reserved by an RTR frame needs to take into account the *maximum length* of a DATA frame plus the ACK frame and all interframe intervals (DIFS, and SIFS), as depicted in Figure 4.7. Notice that, because the polling node has no knowledge about the length of the DATA frame to be sent by the polled node, it needs to reserve the channel for the *worst case* scenario. The channel reservation time announced by the DATA frame considers the time for an ACK plus a DIFS. And the channel reservation time announced by the ACK frame is zero (if the DATA frame has not been fragmented). This procedure for the ACK is similar to what the IEEE 802.11 DCF implements. Likewise, the estimated time announced in the NTS header is equal to zero, since an NTS frame signifies that the handshake is over due to the lack of a DATA frame addressed to the sender of the RTR. As far as NAV updates are concerned, the channel reservation time conveyed by a frame is used only if the advertised time finishes at a time instant superior to the time instant corresponding to the end of a previous RTR time allocation. This is to avoid that other frames on the channel interfere with an ongoing virtual carrier sensing.

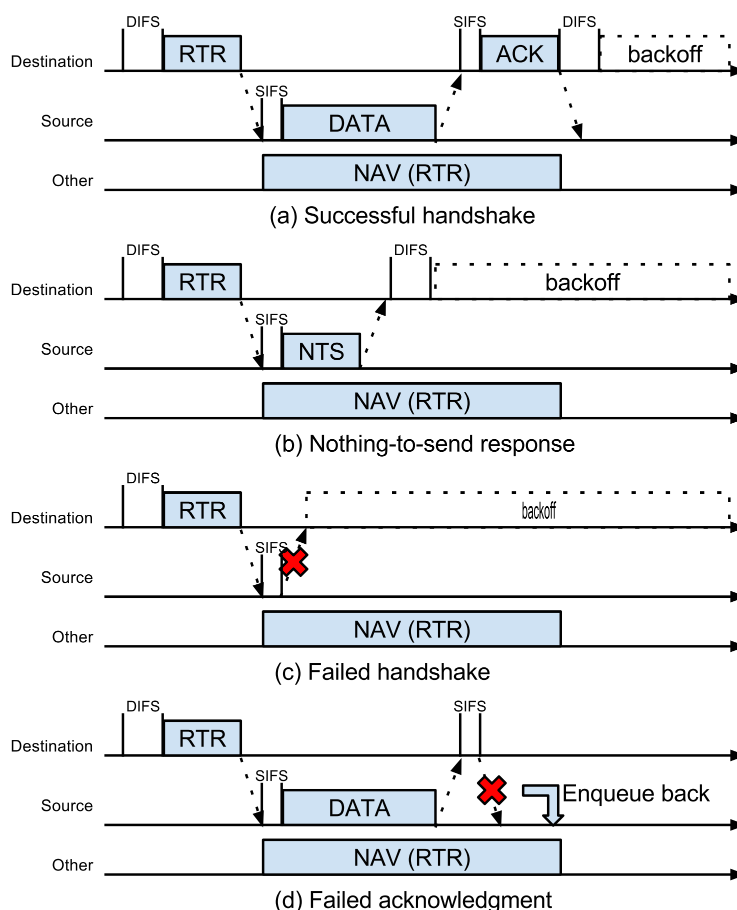


Figure 4.7: Receiver-Initiated handshake cases. (a) Polling with positive DATA response and acknowledgement. (b) Polling with negative response (backoff starts earlier). (c) Polling with no response. (d) DATA transmission with failed acknowledgment.

4.5 Data Acknowledgment and Retransmission

After receiving a successful DATA frame, the node acknowledges it with the sending of an ACK frame after a SIFS time interval, as described earlier. When the ACK frame is received successfully at the polled node, the handshake is finished, as illustrated in Figure 4.7(a). If the data receiver does not get any response (DATA or NTS) within a given time interval, as illustrated in Figure 4.7(c), it retries the sending of an RTR to the same destination address (after a random backoff period) for a maximum number `maxSSRC` of attempts before it decides to poll another neighbor. On the transmitter side, if the node does not get any ACK, as illustrated in Figure 4.7(d), it enqueues the frame back in the first position, and retransmits it (when requested) for a maximum number of attempts `maxSLRC` (maximum station large retry count) before the packet be discarded. The packet is placed in the first position (queue's head), as illustrated in Figure 4.8, because we have to guarantee that, when requested, the packet will be dequeued before the newer ones, since we do the frame reordering operation and the receiver must receive the packets in correct order. If the maximum number of attempts to transmit a given packet is reached, the packet must be discarded as the receiver node may have moved away or left the network [1]. Finally, when the handshake is finished (by ACK or NTS), the node is able to choose a new neighbor to poll. Otherwise, if the handshake fails, the node will retry to send an RTR to the same neighbor. Then, in order to restart the polling process, the node should wait before sending the next RTR frame according to the BEB algorithm explained earlier. The algorithms at both DATA receiver (polling node) and DATA transmitter (polled node) are summarized in Algorithms 1 and 2, and in the flowchart of Figure 4.9 next.

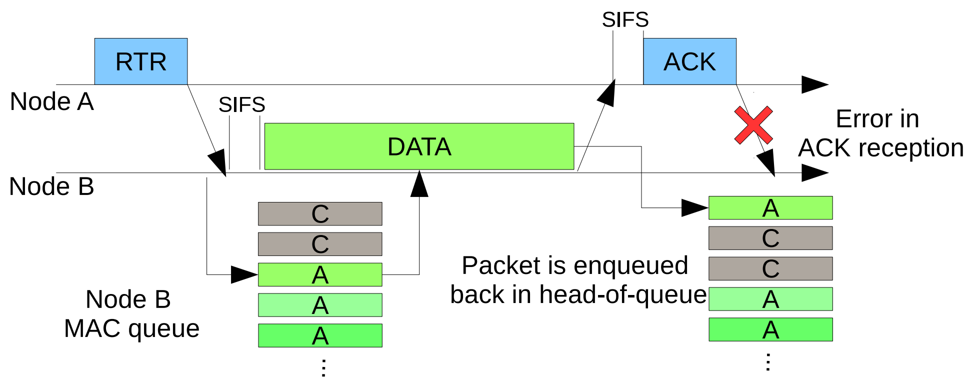


Figure 4.8: Example of ACK timeout. Data packet is enqueued back in MAC queue head-of-line.

[1] Along with the data packet time expiration in the MAC queue, the maximum number of attempts to retransmit a data packet is used to indicate that the packet must be discarded. In fact, only the packet expiration could be used for this purpose, and the `maxSLRC` would be redundant. Thus, since there are a maximum number of polling retries, after this number, the data packet would not be consulted anymore and it will eventually be discarded. However, we decided to maintain the operation of data packet retransmission because it is already implemented in the original sender-initiated IEEE 802.11 protocol code, which we used as basis for implementing the receiver-initiated protocol.

Algorithm 1 DATA Receiver

```
1: procedure DoStart
2:     Reset contention window
3:     Start backoff
4: procedure NotifyAccessGranted
5:     Check packet in head of queue pkt
6:     if pkt is broadcast then
7:         Dequeue pkt
8:         Start broadcast transmission pkt
9:     else
10:        Start polling RTR
11: procedure StartPolling
12:     if NeighborhoodTable is empty then
13:         dest ← broadcastMACAddr
14:     else
15:         dest ← GetAddrFromPollingDiscipline
16:     Send RTR with destination dest
17: procedure ReceiveOk
18:     if Received NTS then
19:         Reset RTR retry counter ssrc
20:         Notify handshake failed
21:         Update contention window failed
22:         Start backoff
23:     else if Received DATA then
24:         Reset RTR retry counter ssrc
25:         Notify handshake success
26:         Send ACK
27:         Reset contention window
28:         Start backoff
29: procedure DATA timeout
30:     Increment RTR retry counter ssrc
31:     if Max RTR retry maxSSRC reached then
32:         Reset contention window
33:     else
34:         Update contention window failed
    Start backoff
```

Algorithm 2 DATA Transmitter

```
1: procedure ReceiveOk
2:     if Received RTR then
3:         Check if DATA frame to polling node in queue
4:         if There is DATA to polling node then
5:             Send DATA
6:         else
7:             Send NTS
8:     else if Received ACK then
9:         Reset DATA retry counter slrc
10:        Notify transmission success
11: procedure ACK timeout
12:     Increment DATA retry counter slrc
13:     if Max DATA retry maxSLRC reached then
14:         Drop DATA
15:     else
16:         Push DATA to head of queue
```

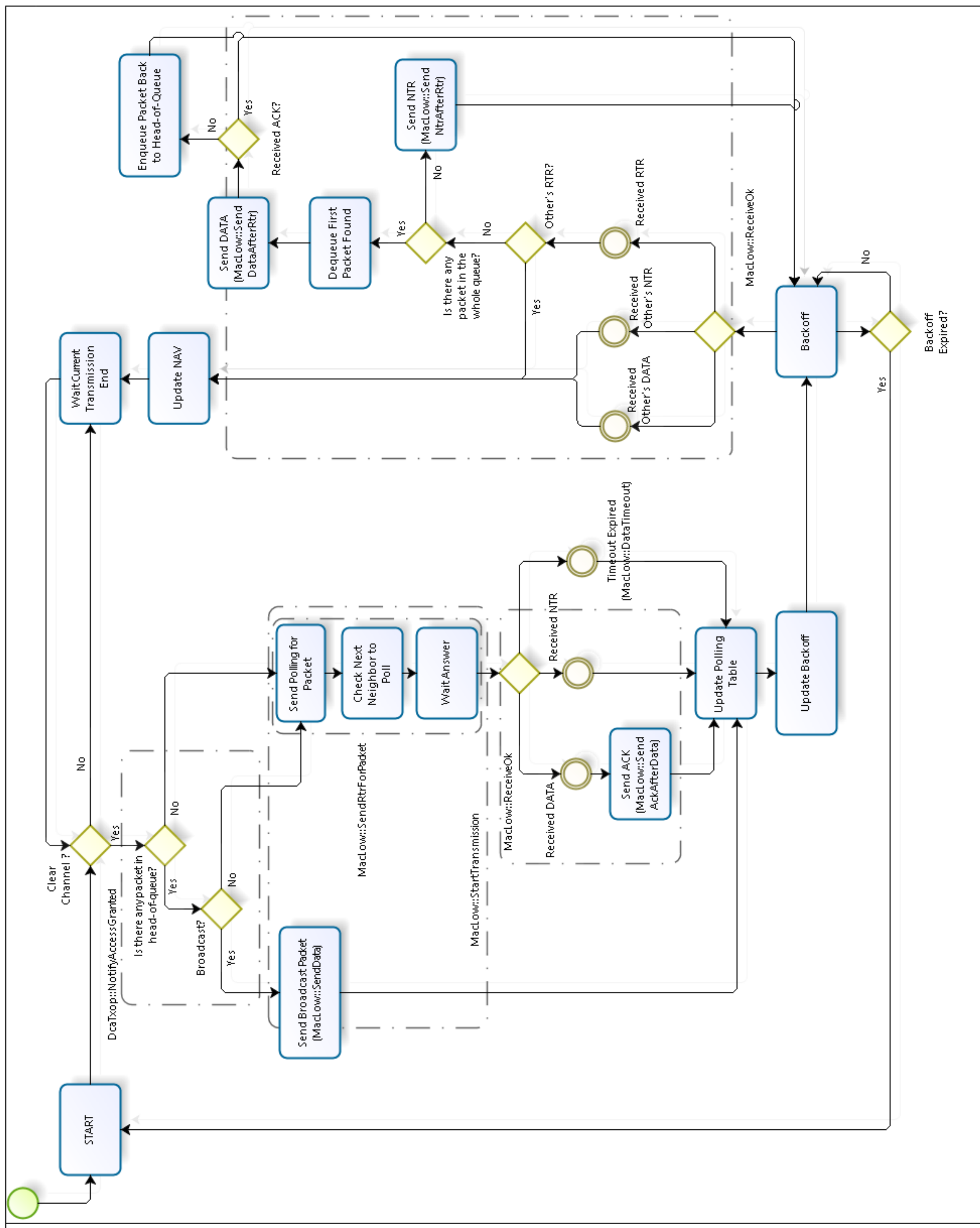


Figure 4.9: Receiver-Initiated MAC flowchart.

4.6 Summary

In this chapter, we described how the Receiver-Initiated MAC protocol works. First, we explained how the BEB algorithm controls the polling rate. Next, we presented the initial operation of the node when it is turned on, the creation of the neighborhood table, and how a node starts to

poll its neighbors. Then, we explained how the Frame Reordering feature is useful for diminishing the packet delay, as the frame does not need to reach the head of the queue to be transmitted. Furthermore, we presented the virtual carrier sensing mechanism adopted for the sake of collision avoidance, and presented the role of the timing information to update the NAV, analogous to the IEEE 802.11 DCF NAV operation. Finally, we explained the data acknowledgement and the retransmission scheme of polling and data frames. Now, we still must define the operation of deciding which neighbor a node will poll. This operation is called Polling Discipline, and, in the next chapter, we will describe the three polling disciplines evaluated in this work, and how each discipline chooses the next neighbor to poll.

This Receiver-Initiated MAC protocol is implemented over the IEEE 802.11b implementation in the *ns-3*, by reverting the handshake paradigm and the binary exponential backoff algorithm, and the created and modified *ns-3* classes are presented in Appendix IV. We considered the utilization of the 802.11b version because, at first, we wanted to simplify the implementation process. Moreover, we want to evaluate the polling disciplines under the receiver-initiated protocol in MAC-level without concerning the data rate achieved by the physical layer, and without concerning Quality-of-Service. Thus, the IEEE 802.11b standard fulfills our requirements with its basic configurations.

Chapter 5

Polling Disciplines

The way how a node chooses the next neighbor to poll can change the performance of the protocol in different criteria. For instance, a discipline that treats all neighbors equally, i.e., that polls each neighbor in the neighborhood table sequentially, without priorities, may actually poll nodes that do not have any DATA frame destined to the polling node. As a result, precious polling time may be wasted, resulting in lower overall throughput performance. Alternatively, if a discipline prioritizes the polling of nodes that have experienced lower average throughput (as an attempt to boost their performance), the protocol may achieve higher fairness, but lower overall throughput. On the other hand, if a discipline prioritizes the nodes with which there are higher probabilities of successful transmission, the protocol will not waste time polling nodes with bad channel conditions (or no DATA frames to it), resulting in higher throughput, but less fairness, since the nodes have different opportunities to transmit. Therefore, the choice of a polling discipline for a receiver-initiated MAC protocol implies on a trade-off between different performance metrics, such as fairness, throughput, and/or delay. Consequently, depending on the target network application, one polling discipline may serve better than others. In this chapter, we describe three polling disciplines that embody different types of prioritization and embody the same BEB algorithm.

5.1 Round-Robin Discipline

The Round-Robin discipline [32] is the simplest of all disciplines and, because of that, it is commonly adopted in many studies. It consists in performing a cyclic poll of all nodes registered in the neighborhood table. The main goal of the round-robin discipline is to make sure that all nodes registered in the neighborhood table are treated equally, in the sense that there is no prioritization in the polling process. In the case of our specific BEB-based MAC protocol, round robin is implemented by making the polling node to switch to the next neighbor in the list only after the end of current polling. This will happen either because *i*) a successful handshake has taken place; *ii*) an NTS frame is received, or *iii*) the retry limit for current polling has been reached. Once all nodes in the neighborhood table are polled, the polling node simply returns to the top of the list to pick up the next node to poll, as depicted in Figure 5.1.

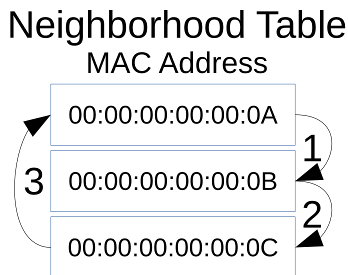


Figure 5.1: Example of Round-Robin discipline in neighborhood table.

5.2 Proportional Fair Discipline

The Proportional Fair discipline targets a minimal level of service to all neighbors in the neighborhood table. This is accomplished by assigning a scheduling priority that is inversely proportional to the historical average throughput of the given flow between the receiver and the potential transmitter [33]. In the prioritization scheme, a node is scheduled when its priority function assumes a value that is the maximum among all nodes. The priority function is given by

$$P = \frac{T^{\alpha}}{R^{\beta}}, \tag{5.1}$$

where T denotes the data rate achievable by the transmit node at the present time, R is the historical average throughput of the transmit node, and α and β tunes the “fairness" of the scheduler. By tuning the parameters α and β we can adjust the ratio with which the “best nodes” (with best channel conditions) are served with respect to the “worst nodes” (the ones under worst channel conditions), in a way that costly nodes are served often enough to have an acceptable level of service. In the extreme case where $\alpha = 0$ and $\beta = 0$, the scheduler acts as a uniformly random scheduler, where all nodes have equal priority (i.e., uniformly distributed scheduling). If $\alpha = 1$ and $\beta = 0$, the scheduler always serves the nodes with best channel conditions, which will maximize their throughput. On the other hand, the nodes with bad channel conditions may suffer from throughput starvation. If $\alpha = \beta = 1$, we have the proportional fair scheduler. Thus, the next neighbor to poll is the one with the largest scheduling priority function, given by

$$P = \frac{T}{R}. \tag{5.2}$$

This priority function is similar to the one used for proportional fair scheduling in 4G networks. Thereby, the node which has the poorest data rate, at some moment, has the highest polling priority, in order to enhance its throughput, as illustrated in Figure 5.2. Hence, this discipline has the potential to deliver high fairness in an ad hoc network, as the nodes will not suffer from starvation. For use in the BEB-based MAC protocol, the computation of R is given by the historical average throughput computed for a particular node. This historical average throughput is computed at a given node for each neighbor registered in the neighborhood table. Each entry has a field that adds the number of bytes received from that particular neighbor since its last entry update, and divides it by the corresponding time length. Each entry has a neighbor expiration time T_{exp} so that the historical average throughput is computed for a maximum period of time T_{exp}.

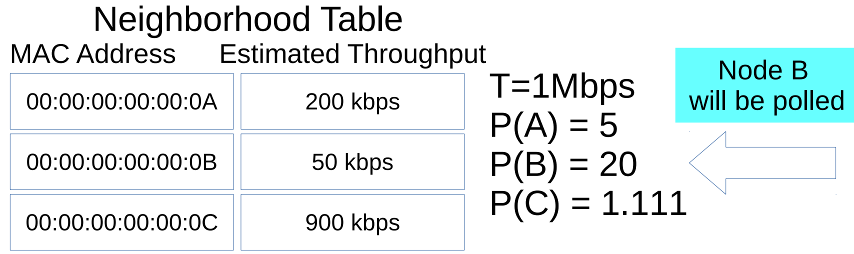


Figure 5.2: Example of Proportional Fair discipline in neighborhood table.

5.3 Likelihood of Successful Handshake (LSH) Discipline

Bonfim and Carvalho [23] have proposed a discipline that assigns polling probabilities to every neighbor registered in the neighborhood table, according to the likelihood of successful handshake, as illustrated in Figure 5.3. The entries in this table are derived from an adaptive *estimation* of the probability of a successful handshake. Thus, each neighbor is polled according to the probability assigned to it. The motivation for this discipline consists in prioritizing the polling of nodes with whom there is a high probability of successful handshake. In fact, a successful handshake depends on both link quality (channel contention and signal propagation conditions) and DATA frame availability at the polled node. Otherwise, significant time may be wasted if a node insists on polling a neighbor that rarely has a DATA frame addressed to it (or that experiences bad channel conditions). As a side effect, one should expect some level of unfairness due to the prioritization of "good" neighbors. It is important to mention that it has been advocated in the literature [34] that the performance of receiver-initiated protocols would achieve its best performance if the *distribution of traffic* at nodes could be known beforehand. This is exactly what the proposed discipline is trying to accomplish indirectly, since it is trying to learn which nodes have DATA frames that can be delivered to it *successfully*.

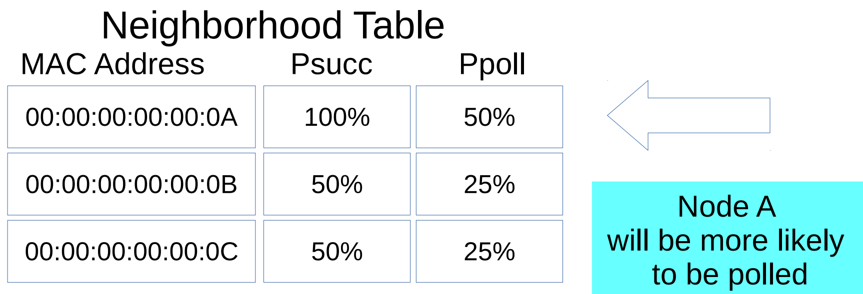


Figure 5.3: Example of Likelihood of Successful Handshake discipline in neighborhood table.

The operation of this discipline consists in the execution of the following steps: 1) estimate the probability P^{succ} of having a successful handshake with each neighbor registered in neighborhood table; 2) compute the probability P^{poll} of polling each neighbor in the neighborhood table; 3) pick one neighbor according to the probability distribution just defined (the node with the highest P^{poll} has the highest chance of being picked); 4) assign the MAC address of the chosen neighbor to the header of the RTR control frame.

Bonfim and Carvalho [23] have proposed an idea for estimating the probability of successful handshake based on an iterative computation. In this work, we improve their idea by using an exponentially-weighted moving average (EWMA) estimator. First, however, we review Bonfim's original work in order to compare it with ours. According to Bonfim [23], the estimation of the successful handshake probability must be updated every time a handshake is attempted with a given neighbor. More specifically, at the end of node j's k-th attempt to initiate a handshake with node i, the estimated probability $P_{ji}^{succ}(k)$ of having a successful handshake with i must be updated as

$$P_{ji}^{succ}(k) = \frac{(k-1) \times P_{ji}^{succ}(k-1) + \eta}{k}, \tag{5.3}$$

where it is assumed that $P_{ji}^{succ}(0) = 1 \ \ \forall i$, and η is an indicator function for the occurrence of a successful handshake in this last attempt (i.e., $\eta = 1$ if success, and $\eta = 0$ if failure). As it can be observed from Eq. (5.3), the effect of η on updating the value of the estimated probability $P_{ji}^{succ}(k)$ diminishes as k increases. This will not be a problem if the network reaches some sort of steady state (as it is assumed in Bonfim's analytical model). However, in practical scenarios, topology and traffic may change dramatically, and Bonfim has suggested to allow the parameter k to assume a maximum value $k_{\max}$, after which it should assume the last value computed for η (0 or 1), and start over the estimation. Evidently, this truncated approach may cause inaccurate estimations of the successful handshake probability. Because of that, we propose a modification based on an *exponentially-weighted moving average* (EWMA) estimation. Instead of using Eq. (5.3), we use the EWMA probability estimation given by

$$P_{ji}^{succ}(k) = (1-\alpha)P_{ji}^{succ}(k-1) + \alpha \times \eta, \tag{5.4}$$

where α is a weight coefficient given to the information regarding the outcome of last attempted handshake. The higher the value of α, the higher is the importance given to the outcome of recent handshakes. On the other hand, the lower the value of α, the higher is the importance given to the average value computed over past handshakes (i.e., past history). In the latter case, one achieves slower convergence and smooth probability estimation.

Every time a new estimation is computed for the probability of successful handshake $P_{ji}^{succ}(k)$, the node has to update the *polling probability* associated with every node in its neighborhood table. The polling probability P_{ji}^{poll} with which node j will poll node i will be given by

$$P_{ji}^{poll} = \lambda P_{ji}^{succ}, \tag{5.5}$$

where λ is obtained from the normalization condition $\sum_{\forall i \in V_j} \lambda P_{ji}^{succ} = 1$, where V_j is the set of nodes in the neighborhood table of node j.

Figure 5.4 depicts Monte Carlo simulations for estimation of the probability of successful handshake according to Eqs. (5.3) and (5.4). The goal of these simulations is to show how each estimator behaves if the values of η are equally distributed, i.e., $P[\eta = 0] = P[\eta = 1] = 0.5$, which means that half of the time there is a successful handshake ("real" $P^{succ} = 0.5$). The parameter values are $k_{max} = 100$ for Eq. (5.3) and $\alpha = 0.02$ for Eq. (5.4). As we can see, the estimation computed according to Eq. (5.3) (blue line) suffers abrupt changes (to 0 or 1) every period of 100 iterations.

The estimation according to Eq. (5.4) (red line) reaches a value close to 0.5 before 100 iterations, and stays around that value without abrupt changes.

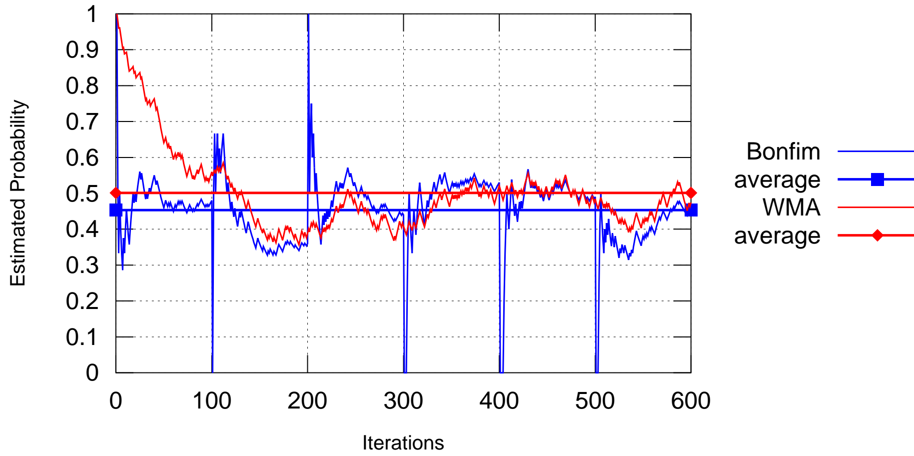


Figure 5.4: Monte Carlo simulations for the estimated probability of successful handshake computed by using both approaches (Eq. (5.3) and Eq. (5.4)).

5.4 Summary

In this chapter, we presented three different polling disciplines with different approaches, each one emboding one type of prioritization. First, we described the simple Round-robin discipline, where a node polls its neighbors in cyclic sequence. Next, we described the Proportional Fair discipline, where a node polls the neighbor with less throughput, in order to try to equalize the throughput levels across all the neighbors. And then, we described the Likelihood of Successful Handshake discipline and its modification with respect to the original proposition, where a node gives more priority to neighbors that have higher probability of successful transmission. Now, understanding the protocol operation and the disciplines, in the next chapter, we will evaluate the performance of the receiver-initiated MAC protocol under the three disciplines and compare to the sender-initiated IEEE 802.11 standard with respect to the MAC control overhead, point-to-point packet delay, network fairness, and flow throughput.

Chapter 6

Performance Evaluation

In this chapter, we evaluate the performance of the BEB-based receiver-initiated unicast MAC protocol under the three polling disciplines, and compare their performance to the sender-initiated IEEE 802.11b DCF[1]. Each version of the BEB-based receiver-initiated MAC protocol is named differently depending on the polling discipline in use: *receiver-initiated round-robin* (RIRR), *receiver-initiated proportional fair* (RIPF), and *receiver-initiated binary exponential backoff* (RIBB) (the original one, with the likelihood of successful handshake (LSH) discipline). The goal of this evaluation is to understand how the BEB-based MAC protocol performs with each polling discipline. Differently from some previous works, the scenarios under investigation consider large-scale channel propagation effects, and network topologies are varied to allow various degrees of spatial sparsity. Regarding traffic conditions, all nodes are saturated (i.e., they always have DATA frames addressed to someone at any time), and traffic is *generated at every node*. This way, we evaluate the worst-case MAC-level performance of each protocol. The destination of every DATA frame is always an immediate neighbor: we focus on MAC-level performance and, therefore, all metrics concern link performance only, without routing activities (all topologies are static, no mobility). In order to investigate the effects of the polling disciplines, we consider two traffic scenarios with respect to the destination of DATA frames in every queue:

- **Scenario A** – the application at each node generates data packets to *all* of its neighbors (according to an exponential distribution, as described next). Thus, every poll for a given node may be potentially answered back with a DATA frame, if the corresponding RTR is received successfully.
- **Scenario B** – the application at each node generates data packets addressed to a third of the neighbors. Thus, on average, only a third [2] of a node's neighbors have DATA frames addressed to it. As a result, many polls may result on the reception of NTS frames, representing a more realistic scenario.

[1] We cannot compare to other receiver-initiated MAC protocols because there is no implementation on the Network Simulator 3.

[2] This is an arbitrary choice, since we observed that the nodes have at least two or three neighbors, it should be at least one neighbor with data packets available.

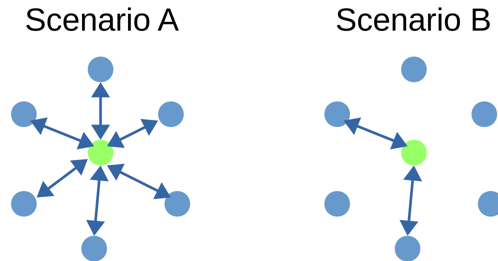


Figure 6.1: Average traffic distribution per node in the neighborhood. In Scenario A, all neighbors have data available. In Scenario B, only one third of the neighbors have data available.

Five metrics are considered for performance evaluation: *control overhead*, *average point-to-point delay per DATA frame*, *fairness*, *average flow throughput*, and *average aggregate throughput*. The *control overhead* measures the average number of control (CTL) frames needed to transmit one successful DATA frame, and it is computed by dividing the total number of control frames (RTR, NTS, RTS, CTS, ACK) transmitted, by the total number of DATA frames transmitted successfully. The *point-to-point delay per DATA frame* measures how long it takes for each DATA frame to reach the other end of the link, i.e., the time from the instant the local application generates a data packet and places it in the transmit MAC queue, to the instant when the packet is received at the other side of the link, after the MAC delivers it to the network layer. The *average delay* is computed by dividing the sum of the delays of all successfully received data packets (from all flows in the network) by the total number of successfully received packets. The *fairness* metric determines whether data flows receive a fair share of the medium access, and it is computed by using Jain's fairness index [35]

$$J(x_1, ..., x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2}, \tag{6.1}$$

where x_i is the throughput of the i-th flow, and n is the number of flows in the network. When the index equals 1, it means that the data flow throughputs are equally distributed. On the other side, when the index equals $1/n$, it means that only one data flow has the maximum throughput. By flow we mean the set of DATA frames generated by a given node and addressed to a specific neighbor (traffic generation is explained shortly). The *average flow throughput* is computed by dividing the total number of received data bits in the network, by the total time it takes to receive that amount of bits and by the total number of data flows. Finnaly, the *network aggregate throughput* is the sum of all flow throughputs in the network, and it measures the transmission capacity of the whole network. The metrics are computed after a period of network warm up, when nodes have already initialized and stabilized their operations.

6.1 Simulation Setup

In order to observe the impact of network contention and spatial reuse, we use topologies with 6 dispersion levels: from a fully-connected network to more sparse topologies. The description of the topologies generation process is in Appendix II. All topologies contain 50 nodes distributed in a terrain of 800 × 800 m. We classify the topologies into six types, each one with a different

average number of neighboring nodes (a neighbor is someone within the transmission range of the node, according to the channel propagation model). Table 6.1 contains a description of the types of topologies and corresponding average number of neighboring nodes for each node. In order to classify the topologies, we define a ratio called *number of hops*, which estimates the average number of hops in the topology (sort of “diameter” of a graph). It is defined as total number of nodes (minus 1) divided by the average number of neighbors of each node. Type-0 topologies are fully-connected networks, whereas type-5 topologies are the ones with the highest number of concurrent transmissions (highest sparsity). Figure 6.2 illustrates examples of topologies used in simulations.

Type of topology	0	1	2	3	4	5
Average number of neighbors	49.0	24.5	12.2	8.1	6.1	4.9
Average number of hops	1	2	4	6	8	10

Table 6.1: Types of topologies used in simulations are classified according to the average number of neighbors per node and average number of hops in the topology.

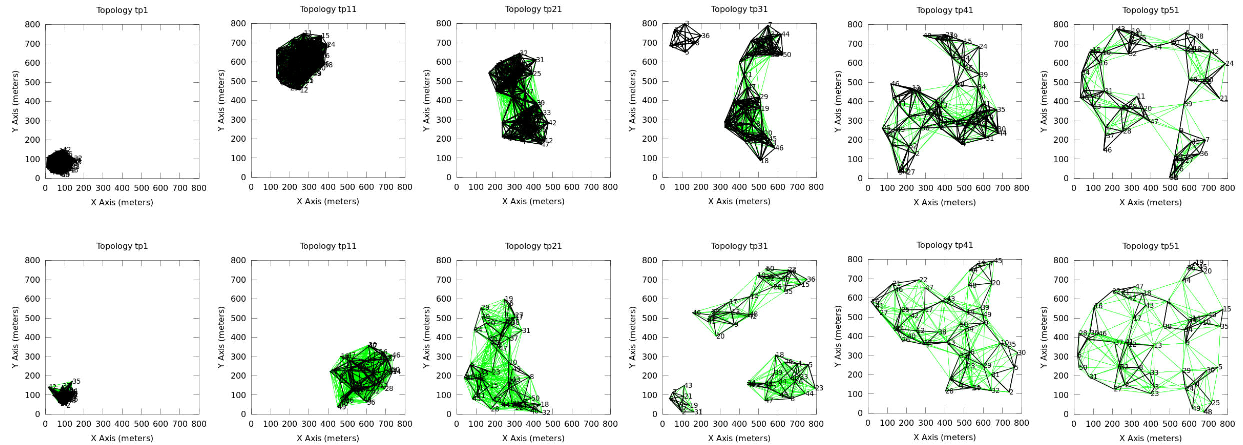


Figure 6.2: Topologies with different sparsity levels used in simulations. Green lines indicate nodes within carrier sensing range of each other, and black lines indicate transmit/receive pairs. Top row shows topologies from scenario A. Bottom row shows topologies from scenario B.

For data traffic generation, the application layer at each node utilizes an “on-off” data source, where data for a single flow is active (on) during an exponentially-distributed random period with an average of 0.3 s, and inactive (off) during an exponentially-distributed random period with an average of 0.9 s. Each “on” period corresponds to the generation of data packets addressed to a *specific* neighbor (filling in the transmit MAC queue). Then, the next “on” period is dedicated for generation of packets addressed to another neighbor (according to scenarios A or B). Hence, at any time, a given node will have a *mixed distribution* of packets addressed to different neighbors. Figure 6.3 shows a snapshot of the packet distribution (by destination address) at the MAC transmit queue of a node labelled “5” during simulations. The figure clearly shows that there is a reasonable distribution of DATA frames to practically all neighbors of node 5.

As far as the polling disciplines are concerned, we set $\alpha = 0.02$ for the weight of the moving

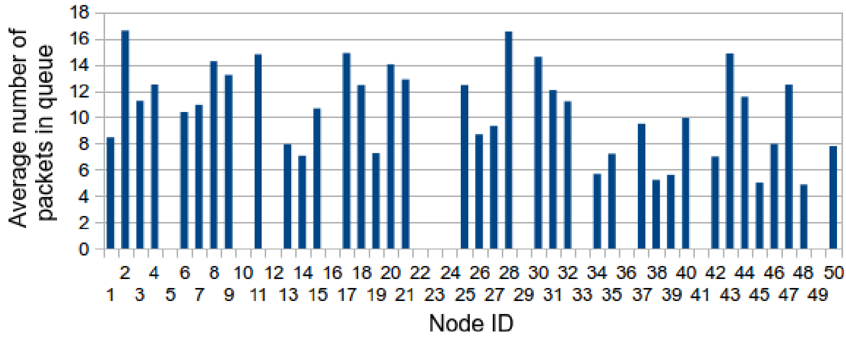


Figure 6.3: Average number of packets in the MAC queue of node 5.

average of the LSH discipline. The value of α depends on how dynamically the topology changes. A high value of α implies that the recent handshake outcomes gain more weight in the successful handshake probability, which is adequated to topologies of high mobility, for instance. On the other hand, a low value of α is more suitable for static topologies or with low mobility. Therefore, a value of α that does not suit the kind of topology may cause an incorrect estimation of the successful handshake probability, which may imply failures on handshake attempts and then degration of performance. In this work, we set a very low α because we are using static topologies. For the Proportional Fair discipline, we set the neighbor expiration time as 0.5 seconds. Hence, the historical average data rate is computed over the number of received bytes in every 0.5 seconds, and the average data rate is estimated over the most recent period. For the BEB algorithm, the maximum number of RTR retransmission attempts (`maxSSRC`) is 7, whereas the maximum number of DATA frame retransmissions (`maxSLRC`) is 7, as well. These values are chosen to be the same as the ones used in the IEEE 802.11b DCF standard in order to have a fair comparison in this work. The length of the RTR frame is equal to the length of an RTS, which is 44 bytes, and the length of the NTS is equal to the length of a CTS, which is 38 bytes. By default, in NS-3, the maximum size of the MAC queue is 400 packets, and the maximum waiting time for any frame in the queue is 10 seconds, after which it is discarded from the queue (if not polled by any neighbor). The maximum waiting time setting is important because, without this parameter, frames addressed to nodes that are no longer neighbors will be kept in queue indefinitely, taking the space of other new incoming packets. Also, when the queue reaches its maximum size, any new incoming packet is dropped. Simulation results correspond to an average computed over four instances of topologies of the same type and four simulation seeds for each topology (16 simulation runs), and error bars in the graphs indicate the computed standard deviation. The rest of simulation parameters are shown in Tables 6.2, 6.3, 6.4, and 6.5.

6.2 Control Overhead

6.2.1 Scenario A

Figure 6.4a illustrates the MAC overhead computed as the average number of control (CTL) frames per DATA frame transmitted successfully. The results are obtained for the receiver-initiated

Table 6.2: Physical layer parameters

Transmission rate	1 Mbps
Transmission power	10 dBm
Transmission range	150 m
Clear Channel Assessment range	225 m
Antenna height	1.2 m
Transmission gain	0 dB
Reception gain	0 dB
Noise figure	10 dB
Propagation model	Two Ray Ground
Modulation	DBPSK
Carrier frequency	2.407 GHz

Table 6.3: MAC layer parameters

P^{succ} mobile average weight (α)	0.02
Neighbor expiration time	0.5 seconds
Maximum Station Short Retry Count (ssrc)	7
Maximum Station Long Retry Count (slrc)	7
RTR frame size	44 bytes
NTR frame size	38 bytes
Maximum Transportation Unit	1500 bytes
ACK frame size	38 bytes
MAC Queue maximum size	400 packets
MAC Queue maximum delay	10 seconds

MAC protocol operating according to each polling discipline, across the six topology groups. Also, the results for the sender-initiated IEEE 802.11b are also shown for comparison purposes. It is noticeable that RIRR and RIPF disciplines present high overhead in less sparse topologies, about 9.5 CTL frames per DATA frame transmitted in fully-connected topologies, on average. Figure 6.4b illustrates the percentage gain of the average overhead with respect to IEEE 802.11b, and one can see that RIRR and RIPF requires 172% of control frames more than IEEE 802.11b in the fully-connected topologies, and 102% more in the second less sparse topology. This is because there are more polling attempts that are not successful due to the high contention levels present in those kind of topologies. Meanwhile, RIBB overhead is less sensitive to topology sparsity, as RIBB prioritizes the neighbors with higher likelihood of successful handshake. Thus, RIBB overhead performance is closer to the 802.11 performance, whose average overhead is about 3.5 CTL frames per DATA across all topologies and it is somewhat constant, and RIBB overhead goes from 4.9 CTL/DATA in fully-connected topologies to 2.5 CTL/DATA in more sparse topologies. In general, receiver-initiated overhead decreases as more sparsed is the topology, becoming even lower than 802.11 overhead in more sparse topology. This is expected because 802.11 standard requires at

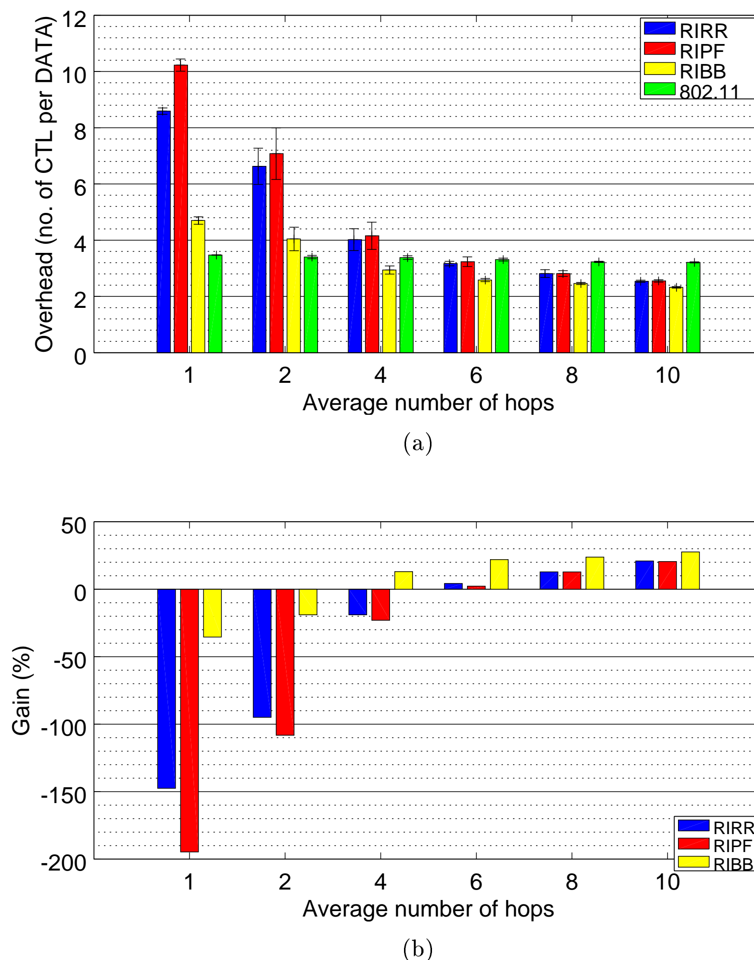
Table 6.4: Application layer parameters

Application	UDP
Packet size	1412 bytes
Data rate	1 Mbps
ON time	Exponential Random Value (mean 0.3)
OFF time	Exponential Random Value (mean 0.9)

Table 6.5: Simulation Parameters

Simulation time	120 seconds
Warm up time	20 seconds
Cool down time	1 second
Nodes	50
Terrain	800 m × 800 m

least three control frames to transmit one DATA frame (RTS, CTS, ACK), while receiver-initiated protocol requires at least two (RTR, ACK).



(a)

(b)

Figure 6.4: (a) Control overhead for each polling discipline compared to IEEE 802.11b, across different groups of topology sparsity in Scenario A. (b) Control overhead gain over IEEE 802.11b.

The high overhead of the Round Robin and the Proportional Fair disciplines in more connected topologies impacts the data flow throughput, as shown further. This result shows that the expected gains of receiver-initiated MAC protocols over sender-initiated ones reported previously in the literature [14] are not feasible without an appropriate polling discipline, despite the reduced number of control frames in a successful handshake.

6.2.2 Scenario B

Figure 6.5a shows the MAC overhead computed as the average number of CTL per DATA frame transmitted successfully for all polling disciplines and the IEEE 802.11b when about one third of a node's neighbors have DATA frames addressed to it. Figure 6.5b shows the percentage gain of the average overhead with respect to IEEE 802.11b. In this scenario, RIRR and RIPF overheads are higher (9.7 CTL/DATA, on average) than in Scenario A (4.8 CTL/DATA), while RIBB and 802.11 overhead do not change that much. This is because, when there are less potential transmitter in the neighborhood, RIRR and RIPF disciplines are more susceptible to get more negative responses (NTS). This is different from Scenario A where all neighbors are potential transmitters. As a result, RIRR and RIPF generates more control traffic before transmitting a DATA frame successfully. The

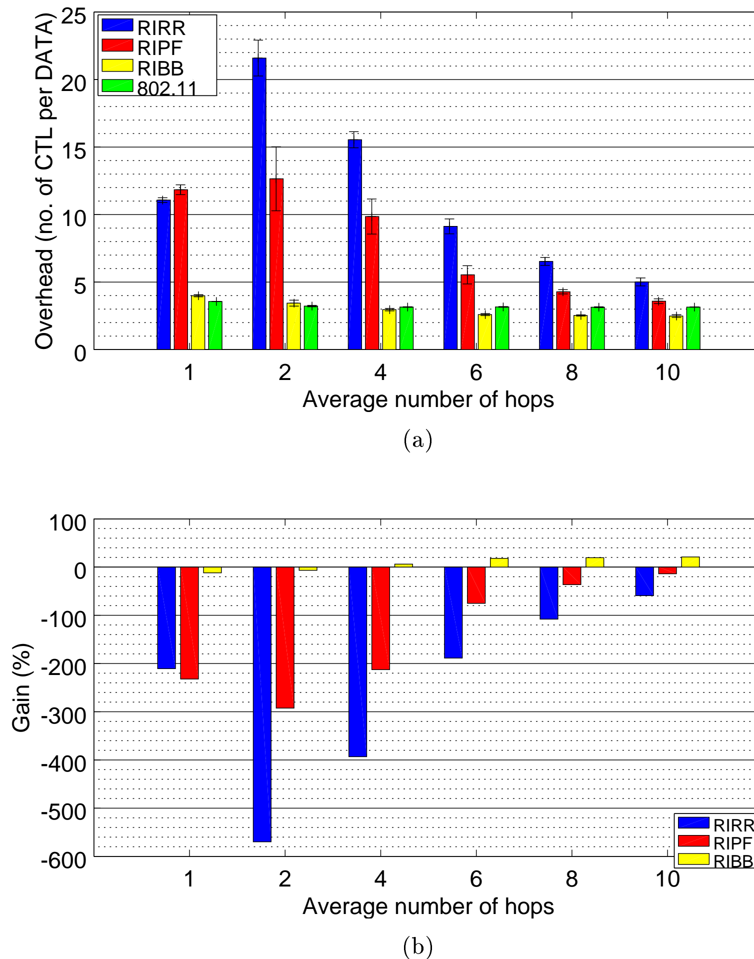


Figure 6.5: (a) Control overhead for each polling discipline compared to IEEE 802.11b, across different groups of topology sparsity in Scenario B. (b) Control overhead gain over IEEE 802.11b.

difference of RIBB and 802.11 performance between Scenario A and B is almost imperceptible (3.2 CTL/DATA on average in Scenario A, and 3.0 CTL/DATA in Scenario B). It shows that RIBB discipline may properly predict which neighbor to poll in both scenarios. RIBB can estimate which neighbors have DATA available to the polling node, and it will not waste control frames polling other neighbors. Thus, RIBB performance can match 802.11 overhead, and even ouperform it in

more sparse topologies due to the reduced number of control frames in a given handshake.

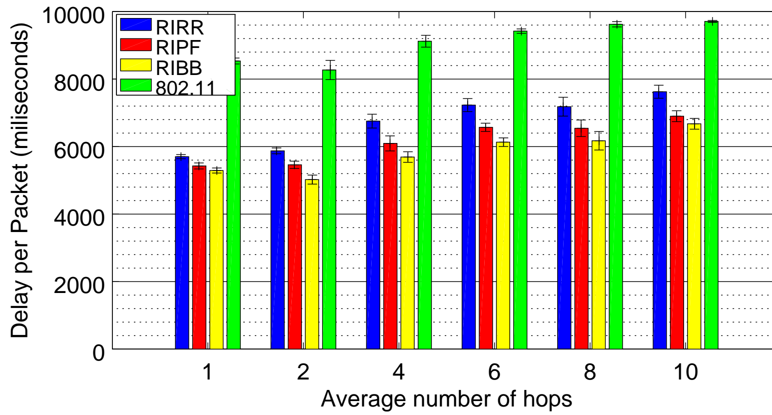
6.3 Average Delay per DATA Frame

6.3.1 Scenario A

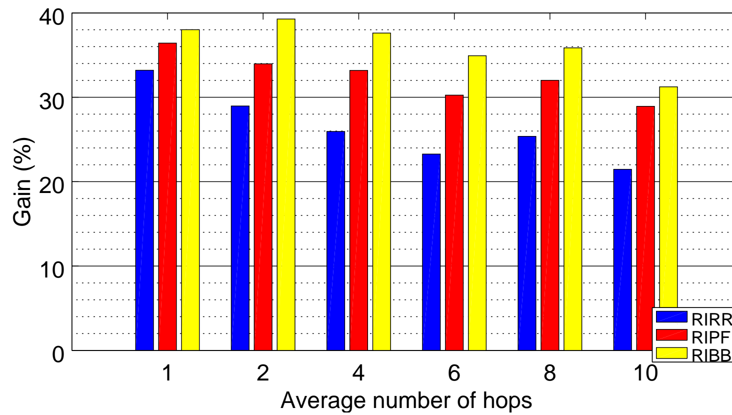
Figure 6.6a depicts the average point-to-point (link) delay per packet. As one can see, the average delay in all receiver-initiated protocols is better than the delay in sender-initiated IEEE 802.11b networks across all topologies and disciplines, with an average gain of 31.6% with respect to IEEE 802.11. Figure 6.6b depicts the percentage gain of the average delay with respect to IEEE 802.11b. Basically, this result is mostly due to the frame reordering mechanism adopted in the MAC queue. In the receiver-initiated protocol, when a node receives a request for a DATA frame, it pulls out the first DATA frame addressed to the polling node, regardless of its position in the queue. Thus, a DATA frame does not need to wait for its arrival at the head of the queue in order to be transmitted, as it is traditionally done in first-in first-out (FIFO) queue implementations of the IEEE 802.11 DCF MAC.

Another important observation to make is the fact that, under scenario A, the average point-to-point delay increases as topologies become more sparse. This result is somewhat contradictory, since a decrease in channel contention should lead to a decrease in average delay. However, under scenario A, all nodes are not only saturated, but they also generate DATA frames to all of its neighbors. Consequently, regardless of the polling discipline, there is always a high chance of consulting a node that has DATA frames addressed to the polling node. This is also true for IEEE 802.11 networks, where the sender finds DATA frames in its queue addressed to all of its neighbors. As a result, given that nodes are more distant from each other under sparse topologies, i.e., the nodes become more distant from each other, signal reception becomes more susceptible to errors due to weaker signal powers. Thus, not only every polled node will likely have DATA frames to deliver, but also each DATA frame will require a higher number of retransmissions in order to be successfully received at the target destination. Such retransmissions incur higher delays, which certainly diminishes the gains of the frame reordering technique adopted in the receiver-initiated protocol. As topologies become more sparse, the gain in performance with respect to IEEE 802.11 decreases by about 8.6% on average, as it can be seen in Figure 6.6b.

As far as the polling disciplines are concerned, we can observe that RIBB has achieved the best performance across all topology groups. On average, the performance gain with respect to RIPF and RIRR is about 9.5%, and with respect to IEEE 802.11b is 36.1%. RIBB discipline prioritizes neighbors with higher successful handshake probabilities, which means that a node prefers to poll neighbors who seem to experience better channel conditions (or are located closer to the polling node), in general. Because, under scenario A, all nodes mostly have (in their MAC queues) DATA frames addressed to all of its neighbors, it is fair to assume that, under this scenario, RIBB mostly prioritizes nodes under better channel conditions (as opposed to the more rare and temporal case when a node does not have DATA frames addressed to it). In spite of being fairer (as we will see shortly), RIPF and RIRR waste more time polling nodes under bad channel conditions or less



(a)



(b)

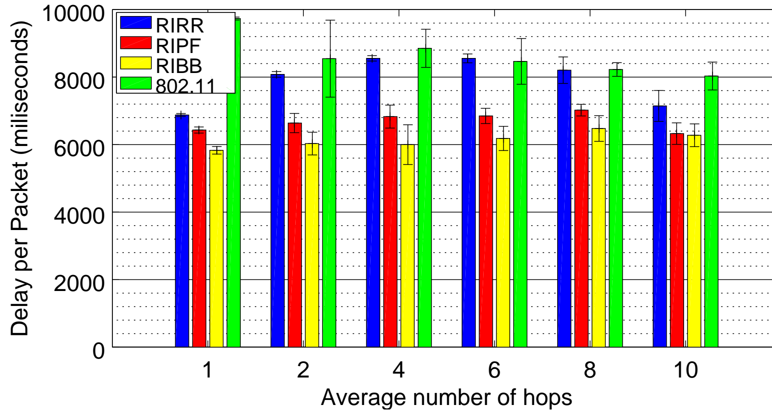
Figure 6.6: (a) Average point-to-point (or link) delay for each polling discipline compared to IEEE 802.11b, across different groups of topology sparsity in Scenario A. (b) Gain on average point-to-point delay over IEEE 802.11b.

temporal availability of DATA frames. This aspect will become more evident in Scenario B, where the difference in performance between RIBB and the others is more striking.

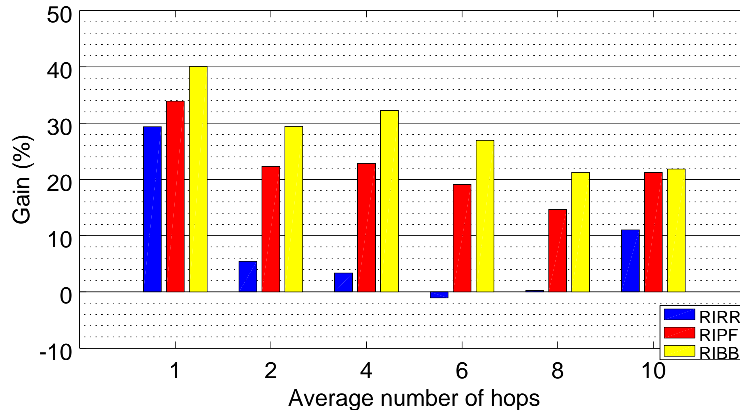
6.3.2 Scenario B

Figure 6.7a depicts the average point-to-point delay per packet computed for all polling disciplines and the IEEE 802.11b when about one third of a node's neighbors have DATA frames addressed to it. Figure 6.7b depicts the percentage gains with respect to the IEEE 802.11b performance. In this scenario, the absolute delay values are a little higher, about 5.4% more on average compared to the case where all nodes have DATA frames addressed to all neighbors. The receiver-initiated delay values increase because the node wastes time polling nodes that do not have DATA frames to it. Still, receiver-initiated delay is lower than IEEE 802.11 delay. Different from Scenario A, however, the average delay tends to decrease as topologies become more sparse. This can be explained as a direct consequence of the use of the NTS control frame: polled nodes

with no DATA frames to the sender of the RTR immediately responds with an NTS to allow the polling of other nodes. Thus, given that only a third of a node's neighbors (on average) have DATA frames addressed to it, the polling process is sped up by the reception of NTS frames.



(a)



(b)

Figure 6.7: (a) Average point-to-point (or link) delay for each polling discipline compared to IEEE 802.11b, across different groups of topology sparsity in Scenario B. (b) Gain on average point-to-point delay over IEEE 802.11b.

As mentioned in Scenario A, RIBB outperforms RIPF and RIRR and this characteristic is better observed in this scenario, where a polling node will not always receive a DATA frame in response because the polled node may not have a DATA frame addressed to it. In addition, RIBB is 28.6% better than IEEE 802.11b, on average. In fact, RIBB delay is not only the lowest, but it also presents small variations among all topology groups. Basically, the delay seems to be ruled by node saturation, and RIBB seems to do a good job in polling only the nodes of interest.

6.4 Fairness

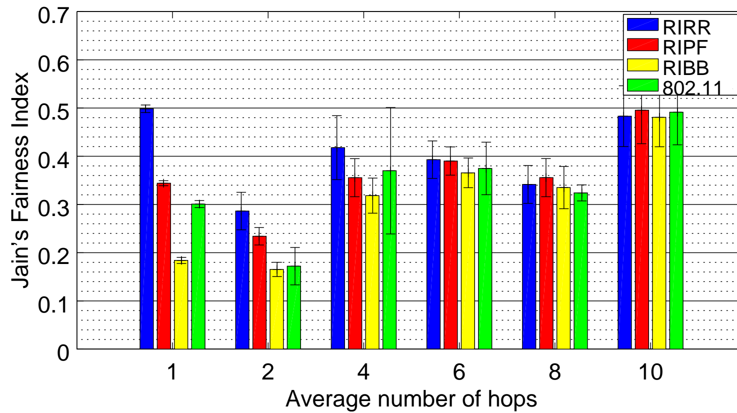
6.4.1 Scenario A

Figure 6.8a contains the results for the fairness in throughput achieved by each protocol according to Jain's fairness index. Figure 6.8b illustrates the percentage gains with respect to IEEE 802.11b. It is well-known that IEEE 802.11 networks present poor fairness performance due to the use of the binary exponential backoff (BEB) algorithm [17]. Under BEB operation, the nodes that last acquired the channel are the ones more likely to acquire it again (since they will start off with lower contention window sizes). Such unfair behavior may be exacerbated if a node's queue is filled with a stream of successive frames addressed to the same destination. On the other hand, the use of the BEB algorithm on a receiver-initiated protocol does not necessarily lead to this biased behavior. This is because, although some nodes may dominate channel access due to lower contention window sizes (as a result of BEB), their handshakes are not necessarily biased towards the same destination. For instance, RIRR selects a different neighbor every time it performs a new BEB cycle. Thus, depending on the polling discipline, the receiver-initiated protocol may distribute channel access fairerly among nodes.

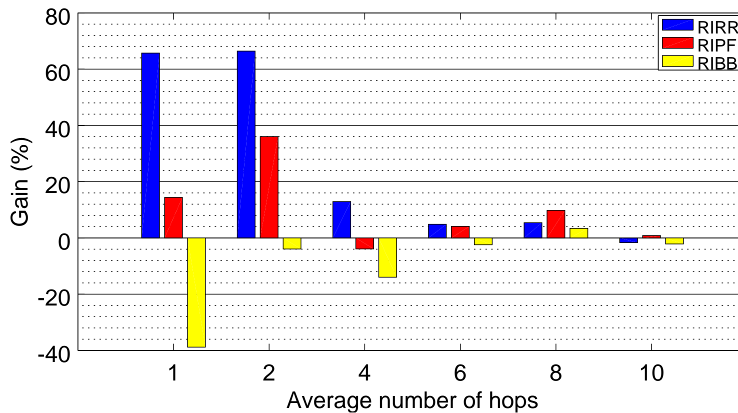
The Proportional Fair and Round Robin disciplines prevailed in this evaluation, as expected, according to their main objective. The average gains of RIPF and RIRR with respect to IEEE 802.11b are 10.2% and 25.6%, respectively. Compared to RIBB, their performance gains are 17.6% and 30.8%, respectively. Under Scenario A, all neighbors usually have DATA frames in their queues addressed to the polling node. Because of that, under low sparsity, RIRR delivers the best performance, since all neighbors are treated equally. But, as topologies become more sparse, RIPF surpasses RIRR because it tries to compensate for those nodes that are more likely to undergo channel errors due to their distance from the polling node.

RIBB has the worst fairness performance among polling disciplines in Scenario A. This is expected, since RIBB prioritizes nodes with whom there are higher chances of successful handshakes. In fact, under Scenario A, RIBB is worse than IEEE 802.11 when topologies are less sparse. This unfair behavior is possibly aggravated due to the confluence of two factors: under Scenario A, all neighbors are likely to have DATA frames queued for the polling node, but only a subset of them end up being prioritized by RIBB. This is due to a "reinforcement effect" in the iterative probability computation of Eq. (5.4), which tends to favor the first nodes with which a successful handshake has occurred. Secondly, the nodes who last acquired the channel are the ones more likely to access it again. Therefore, the reinforcement effect is also confined to a small number of polling nodes in less sparse topologies. However, as topologies become more sparse, channel errors come to play, and neighbors are no longer "similar" to each other. Hence, under more sparse topologies, RIBB learns about the best neighbors with which it can have a successful handshake. Because of that, RIBB performance improves, and it becomes closer to the IEEE 802.11 performance.

From Figure 6.8a, one can realize that the more dispersed the nodes are, the lower are the differences in fairness among polling disciplines. This is because, as there are fewer neighbors around, there will be fewer options left to the polling discipline to choose, and the same nodes



(a)



(b)

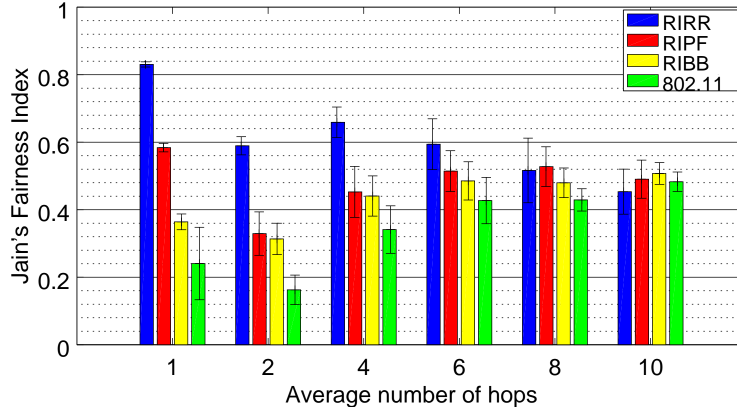
Figure 6.8: (a) Jain's fairness index for different polling disciplines and different topologies in Scenario A, according to their average number of hops. All disciplines are compared to sender-initiated IEEE 802.11 DCF MAC. (b) Gain of fairness with respect to IEEE 802.11b.

are likely to be polled regardless of the discipline. Furthermore, the gains over IEEE 802.11b decrease as topologies become more sparse, for the same previous reasons, ranging from 13.7% in fully-connected networks (on average), to -1%, in more sparse networks. Exceptionally, RIBB gains increase with the dispersion of topologies.

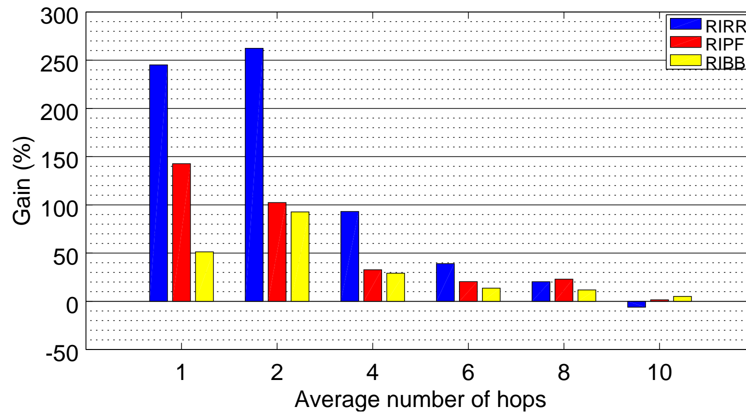
6.4.2 Scenario B

Figure 6.9a illustrates the fairness in throughput measured by Jain's index, and Figure 6.9b illustrates the percentage gains in relation to IEEE 802.11b. In this scenario, RIBB becomes better than 802.11b, especially in less sparse topology (as opposed to Scenario A). The fact that RIBB learns which neighbors have DATA for the polling node contributes to this result. Moreover, RIBB is consistently better than 802.11b in all topology groups, and becomes better than RIRR in the more sparse topology group. The fairness in throughput of the IEEE 802.11b is very bad in fully-connected scenarios and in less sparse topologies, about 0.24 and 0.16 respectively, while the

receiver-initiated protocol achieves gains in relation to IEEE 802.11b from 72.0% (RIBB) to 122.5% (RIPF) and 253.7% (RIRR) in these topology groups. The fairness index of RIPF and RIRR are higher in these first groups because they serve each node more equally (especially RIRR), despite spending time on polling unnecessary nodes (see Figures 6.6 and 6.7). For this same reason (i.e., spending time on polling unnecessary nodes), RIRR becomes worse than 802.11b in more sparse topologies, about 6.0% of loss in relation to 802.11b for the more sparse topologies.



(a)



(b)

Figure 6.9: (a) Jain's fairness index for different polling disciplines and different topologies in Scenario B, according to their average number of hops. All disciplines are compared to sender-initiated IEEE 802.11 DCF MAC. (b) Gain of fairness with respect to IEEE 802.11b.

6.5 Average Throughput

6.5.1 Scenario A

Figure 6.10a shows the average flow throughput, i.e., for each flow of data between a specific pair of nodes. Figure 6.10b shows the percentage gains in relation to IEEE 802.11b. As it can be observed in less sparse topologies, the receiver-initiated protocols present lower throughput

performance than IEEE 802.11. However, absolute values are generally very close, and they all follow an exponential-like increase as topologies become more sparse. Performance losses with respect to IEEE 802.11 are mostly under 23.5% with RIRR, 38.1% with RIPF, and less than 3.5% with RIBB, for less sparse topologies. This is a good result, considering the 31.6% average gain in delay achieved by receiver-initiated protocols under Scenario A. In more sparse topologies, receiver-initiated protocols present gains with respect to IEEE 802.11 from 0.6% to 4.24%, with an average of 2.8%. Such a behavior has already been predicted by Bonfim and Carvalho [23], where the 2.8% gain is very close to the one predicted by the analytical model (3%). This is basically due to the reduction in the number of control frames in RIBB (see Figure 6.4b).

Given that all nodes generate traffic to all of its neighbors in Scenario A, the difference in performance between receiver-initiated protocols is not very different. Also, because of this traffic distribution, average throughput of the flows are much lower than the case where traffic is generated from only a third of the neighbors (which is analyzed in next). For completeness, Figure 6.10c depicts the average aggregate throughput, i.e., the average sum of every flow throughput within the network. This metric gives an idea of the increase in network throughput as a result of spatial reuse. Given that the percentage gains are similar to the average flow throughput, the same previous observations apply for the average aggregate throughput.

6.5.2 Scenario B

Figure 6.11a shows the average flow throughput when DATA frames are generated from a third of a given node's neighbors. Given the smaller number of different flows, throughput values are much higher across all topologies compared to Scenario A. In this scenario, it is apparent the throughput degradation observed in both RIRR and RIPF. Both disciplines unnecessarily poll neighbors with no DATA frames (or initiate handshakes under bad channel conditions) more often than RIBB does, since RIBB prioritizes nodes with whom it can actually communicate or contains DATA frames. As a result, not only RIBB improves its performance in fully-connected scenarios (3.4% gain) but it also keeps up with the throughput increase of the IEEE 802.11. Moreover, under Scenario B, RIBB is always fairer and incur less delay than IEEE 802.11 across all topologies, in addition to a 7.6% average gain in overhead. Figure 6.11b shows the gain of the average flow throughput in relation to the IEEE 802.11b performance and Figure 6.11c depicts the average aggregate throughput.

6.6 Conclusions

This chapter presented a performance analysis of a random-access receiver-initiated MAC protocol that utilizes a reversed version of the binary exponential backoff (BEB) algorithm of the IEEE 802.11 DCF as a means to self-regulate and control the rate at which a node polls its neighbors. The use of the BEB algorithm indirectly takes into account the perceived level of contention, channel state, and DATA frame availability at polled nodes. The proposed receiver-initiated MAC protocol is also enhanced by allowing frame reordering at transmit queues, and the incorporation

of the *nothing-to-send* (NTS) control frame, which helps on speeding up polling rounds (i.e., a node that receives an NTS from a polled node may immediately switch to the next neighbor in its neighborhood table). To suplement the polling rate control mechanism, we also introduced an enhanced version of an adaptive polling discipline that prioritizes the polling of nodes according to the likelihood of successful handshake. In addition to this polling discipline, we also investigated the traditional round-robin scheme, and a variant of the proportional fair scheduling mechanism typical of 4G networks. The performance of the receiver initiated MAC protocol with each of the polling mechanisms (RIBB, RIRR, and RIPF) was compared to the performance of the sender-initiated IEEE 802.11 DCF with respect to MAC-level control overhead, delay, fairness, and throughput. Using a discrete-event simulator, we compared the performance of all protocols under two traffic scenarios in networks topologies with different sparsity characteristics.

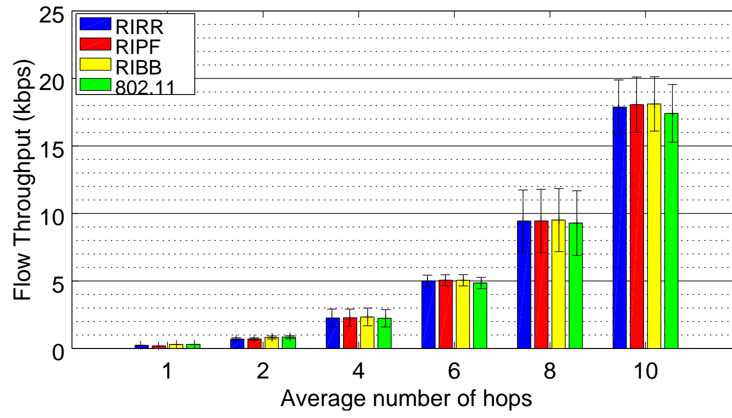
Regarding MAC overhead, we have observed that the receiver-initiated protocol has a lower control overhead due to the reduced number of control frames in a handshake. However, one can see that RIRR and RIPF require up to four times more control frames to transmit one DATA frame than IEEE 802.11, in some cases. On the other hand, RIBB can keep up with IEEE 802.11 performance, and even achieving better performance in more sparse topologies, as the network contention becomes less influent. This is a consequence of the priorization of neighbors with higher likelihood of successful handshake, as the polling node potentially do not waste the usage of control frames to poll neighbors with bad channel conditions or with no data frames to respond.

In general, we could observe that, as far as delay is concerned, the receiver-initiated protocols (RIRR, RIPF, and RIBB) performed better than IEEE 802.11 across all traffic scenarios and topologies. This is a direct consequence of the frame reordering technique and the introduction of the NTS control frame. In particular, RIBB delivers the best performance among all. When traffic is not homogeneously distributed among neighbors, RIBB learns the neighbors that actually have DATA frames addressed to it, and prioritizes the ones which are also under relatively good channel conditions. This does not happen with RIRR and RIPF, which either treats all nodes equally (RIRR) or keeps trying to boost the performance of nodes under unfavorable conditions (RIPF).

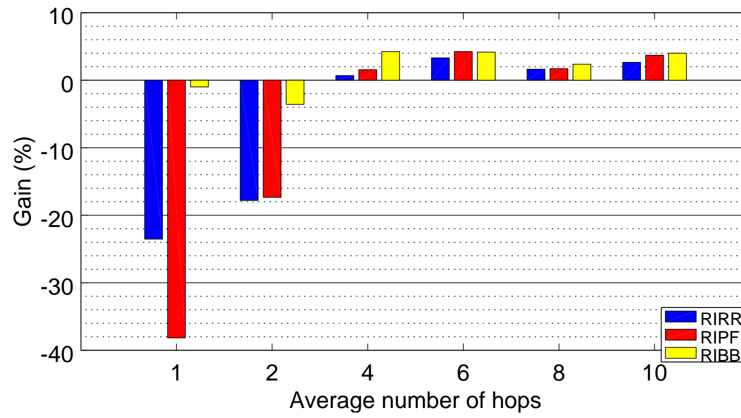
Regarding fairness, RIPF and RIRR prevailed in this category, as expected, because of their inherent properties. Under fully-connected scenarios (or low sparsity), IEEE 802.11 is very unfair, especially when traffic is not homogenously distributed among nodes. Because of the prioritization scheme incorporated into RIBB, it does not perform as well as RIRR and RIPF in homogeneous traffic scenarios (although it outperforms IEEE 802.11 in more sparse networks), but it is certainly better than IEEE 802.11 in every topology when traffic is not equally distributed among nodes. Here, it is interesting to notice that the well-known fairness issues of the IEEE 802.11 BEB algorithm are less pronounced in its reversed version because, even though a node may still dominate channel acquisition more than its neighbors, polled nodes may vary completely, depending on the polling discipline. This is in direct contrast to sender-initiated MAC protocols with FIFO queue discipline, where a node not only may dominate channel acquisition, but it may also “lock” on a specific receiver due to a stream of successive same-destination DATA frames in its transmit queue. This phenomenon is well illustrated by the low fairness values obtained by IEEE 802.11 DCF in

the studied scenarios, especially under less sparse topologies.

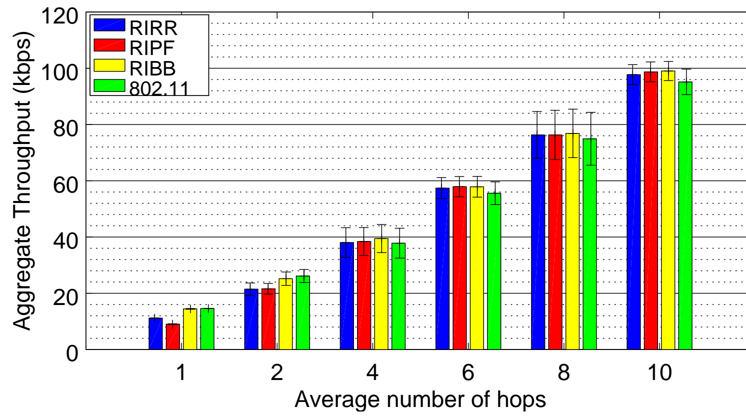
Finally, regarding throughput, we could observe that the losses in performance of receiver-initiated MAC protocols with respect to IEEE 802.11 were not very high, especially for RIBB, which could closely follow IEEE 802.11 in both types of traffic scenarios. RIRR and RIPF outperform IEEE 802.11 in more sparsed scenarios, and when all neighbors are potential transmitters (Scenario A), as RIBB as well. However, in Scenario B, where only one third of neighbors have data to transmit, RIRR and RIPF performance degrade considerably, in contrast to RIBB outperforming IEEE 802.11 in some cases, which is a combined effect of a lower number of control frames, the use of NTS and frame reordering, and the adaptive learning of DATA frame availability at neighboring nodes (queues are finite, and they may not contain packets to some nodes, occasionally). Also, from the results, it is clear that we should seek the design of a polling discipline that balances the features of RIPF and RIBB according to the dynamics of the network topology and traffic. This approach is studied in Chapter 7.



(a)

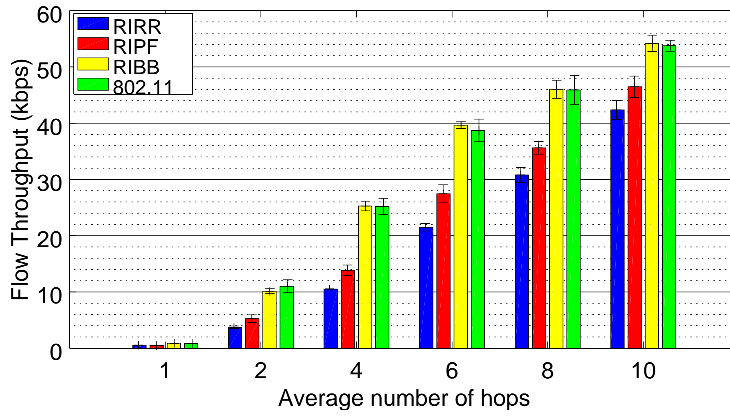


(b)

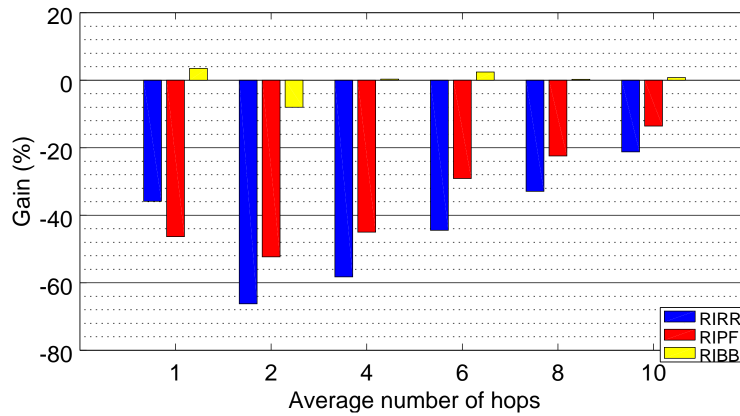


(c)

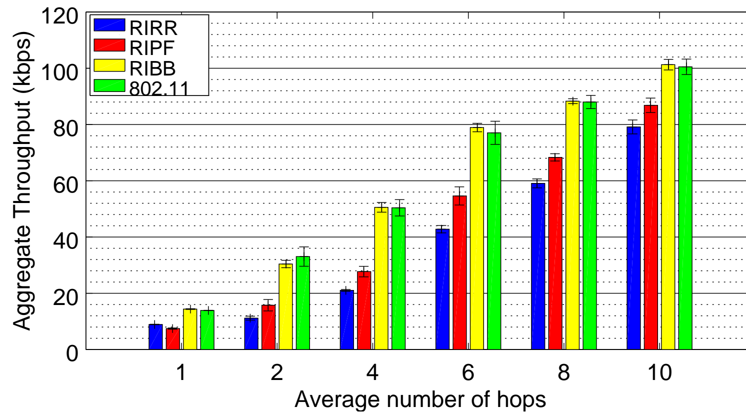
Figure 6.10: (a) Average throughput for different polling disciplines and different topologies in Scenario A, according to their average number of hops. All disciplines are compared to sender-initiated IEEE 802.11 DCF MAC. (b) Gain of average throughput with respect to IEEE 802.11b. (c) Average aggregate throughput.



(a)



(b)



(c)

Figure 6.11: (a) Average throughput for different polling disciplines and different topologies in Scenario B, according to their average number of hops. All disciplines are compared to sender-initiated IEEE 802.11 DCF MAC. (b) Gain of average throughput with respect to IEEE 802.11b. (c) Average aggregate throughput.

Chapter 7

RIMAP: Receiver-Initiated MAC Protocol with Adaptive Polling Discipline

From the results of the polling disciplines evaluation in the previous chapter, the utilization of the disciplines Likelihood of Successful Transmission and the Proportional Fair suggests a trade-off situation: as the LSH discipline prioritizes the neighbors with best probability of successful handshake, in other words, with potentially better channel conditions or better chances to have positive data response, the LSH improves data throughput at the expense of network fairness, since it tends to select fewer neighbors to communicate with. Conversely, PF gives proportional chances to neighbors with low data throughput in order to try to equalize the throughput of all neighbors, and thus improving overall network fairness without concerning to achieve throughput performance. But, in a wireless ad hoc network (and its applications, such as in vehicular networks), there may exist scenarios where one discipline may fit better than the other. For instance, a node that has too many neighbors—but perceives unequal and heterogeneous channel conditions to each of them—will probably be better served if it uses the LSH discipline, which prioritizes the best neighbors only. Likewise, a node that has few neighbors, and homogeneous link quality to each of them, may be better served if it uses the PF discipline, because it may allow a fair share of channel access without jeopardizing individual throughput. Hence, instead of using a specific polling discipline, a receiver-initiated MAC protocol could implement an adaptive polling discipline according to neighborhood conditions at any time.

In this chapter, we present the Receiver-Initiated MAC with Adaptive Polling Discipline (RIMAP), a unicast MAC protocol that dynamically selects a polling discipline (LSH or PF) according to channel contention and link quality homogeneity to all neighbors. The adaptive behavior is controlled by two switching parameters that can be tuned to trade off fairness with throughput-delay performance. To control a node's *polling rate*, RIMAP utilizes the same reversed version of the binary exponential backoff (BEB) algorithm of the IEEE 802.11 DCF (as presented before). Similarly to RIRR, RIPF and RIBB protocols, it also implements the frame reordering

(FR) technique, and the *Nothing-To-Send* (NTS) control frame. RIMAP targets networks where every node acts as both transmitter and receiver (i.e., there are no special nodes, such as "sinks", etc.), and without any node hierarchy (e.g., no master/slave, etc). RIMAP performance is evaluated with discrete-event simulations under topologies that present hidden terminals, concurrent transmissions, and saturated traffic. Also, its performance is compared with the same BEB-based MAC protocol under fixed polling disciplines (LSH or PF only), as well as with the IEEE 802.11 DCF MAC, a representative of the sender-initiated paradigm.

7.1 Adaptive Polling Discipline

The RIMAP protocol operates as described in Chapter 4 only differing on how a given node chooses the next neighbor to poll. Previously, a given node utilizes a single polling discipline along time to select a neighbor to potentially receive a data frame from. Thus, the network could achieve different performances according to the discipline adopted. On the other hand, RIMAP implements a mechanism that switches the polling discipline utilized when it is appropriate.

RIMAP seeks to find a balance between the two previous polling disciplines by taking advantage of their strengths in an adaptive manner, according to the scenario perceived by each node. Hence, every node makes use of two parameters to decide whether to switch to one discipline or the other. A first criterion for switching is based on how *homogeneous* the quality of the links are with respect to every neighbor in the neighborhood table. For that, an average signal-to-noise ratio (SNR) is estimated each time a frame is received from each neighbor, and a historical average SNR is associated to every neighbor. The *variance* of the estimated SNR values across all neighbors is used as an indication of quality homogeneity: the smaller the variance, the more homogeneous the quality of the links.

The other switch criterion is based on the *number of neighbors* registered in the neighborhood table, at the moment of polling. When there are many neighbors, it may become too hard to serve every neighbor fairly, especially if link quality is not homogeneous. Too much time may be spent trying to poll a portion of the neighbors under low link quality, which can severely damage the average throughput of nodes that could be better served otherwise. But, when there are few neighbors, it may be worth it to pursue a more fair distribution of throughput among neighbors, without compromising too much individual throughput, especially if all neighbors experience similar link quality. Based on these observations, two threshold parameters are defined: `snrVarThresh` and `nNeighThresh`, which control the SNR variance and the number of neighbors, respectively. Hence, the following discipline for switching between disciplines is proposed, represented by the Algorithm 3: before polling a given node, if both estimated SNR variance and number of neighbors are higher than their respective thresholds, switch to LSH discipline. Otherwise, switch to PF discipline. Thus, the node switches to LSH discipline when both SNR variance and number of neighbors are higher than their respective thresholds. Otherwise, the node switches to PF discipline.

The adaptive mechanism of RIMAP protocol acts at the moment the polling node sets the

RTR destination address from the neighborhood table, where it is stored the average SNR of the frames received by each neighbor. From this, the polling node computes the SNR variance among the neighbors and the number of neighbors in the table, as well. Then, the polling node compare these computed values with the given thresholds in order to decide which discipline will be used for setting the RTR destination address, according to the Algorithm 3. Thus, the polling discipline (LSH or PF) will choose the best neighbor to poll according to the current network condition, parameterized by the SNR variance and the number of neighbors.

Algorithm 3 Discipline switching mode

```
1: procedure RIMAP
2:     if snrVar > snrVarThresh and nNeigh > nNeighThresh then
3:         use LSH
4:     else
5:         use PF
```

7.2 Performance Evaluation of RIMAP for Ad Hoc Networks

Now, we evaluate the performance of RIMAP for different threshold values, and compare it with fixed polling disciplines under the same BEB-based MAC protocol (RIPF and RIBB protocols). Adittionally, we compare RIMAP with the standard IEEE 802.11 DCF MAC. Three performance metrics are considered: *average point-to-point delay per DATA frame*, *fairness*, and *average flow throughput*.

The scenarios under investigation consider topologies that are *not* fully-connected (i.e., no single-hop, topologies of Type 4, described in Table 6.1 in Chapter 6). Therefore, hidden terminals may occur, as well as concurrent transmissions between distinct pair of nodes. All topologies contain 50 nodes distributed on a terrain of 800×800 m, as depicted in Figures 7.1a and 7.1b. Only large-scale propagation effects are considered (no small-scale fading), and traffic is saturated at all nodes, i.e., every node always has a DATA frame addressed to someone, at the head of their queues, at any time. The destination of every DATA frame is always an immediate neighbor: we focus on worst-case MAC-level performance only, and, therefore, all metrics express link performance, without taking into account routing activities (all topologies are static, no mobility). Hence, data is generated at each node, and data packets are addressed to only a third of the neighbors, so that, on average, only a third of a node's neighbors have DATA frames addressed to it (notice that, each node generates data flows to a number of neighbors, as the Scenario B in the performance evaluation in Chapter 6). This is to allow the occurrence of NTS frames in simulations: a fraction of the polling attempts will trigger the transmission of NTS frames. The simulation parameters are summarized in Tables 6.2, 6.3, 6.4, and 6.5. The NS-3 simulator [24] is used for simulations, and the performance results correspond to average values computed over four instances of topologies, each with three different simulation seeds. In the following graphs, error bars indicate standard deviation, and each point indicates RIMAP's performance for specific values of both SNR variance (sT) and number of neighbors (nN) thresholds. The three horizontal lines in the graphs indicate the performance of RIPF, RIBB, and IEEE 802.11, for comparison purposes. We evaluate the RIMAP

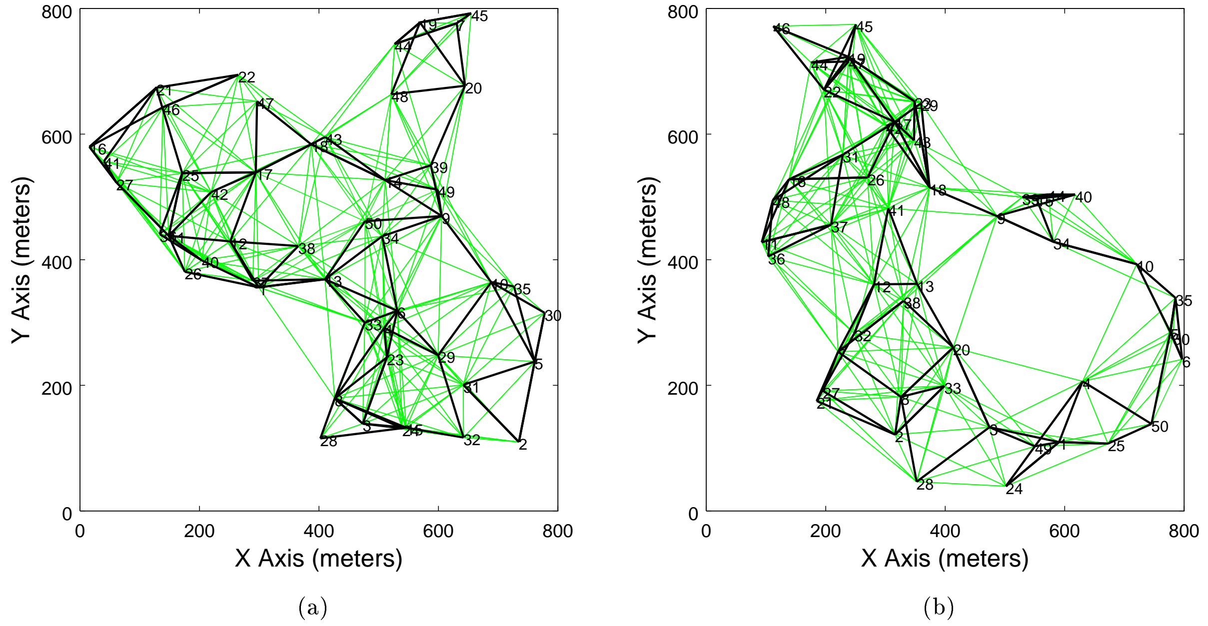


Figure 7.1: Sample topologies: green lines indicate nodes within carrier sensing range of each other, and black lines indicate transmit/receive pairs.

performance under the topologies of Types 0, 2, and 4 (the types correspond to a given sparsity level defined in Table 6.1) in traffic Scenario A and B, but we will present only the results for the topologies of Type 4 of Scenario B. The results for topologies of Type 0 and Type 2 (less sparse) of Scenario B, and topologies of Types 0, 2, and 4 of Scenario A, as well, are shown in Appendix I. We omit the results for Scenario A topologies because, in this scenario, the performances of RIBB and RIPF with respect to flow throughput and fairness do not differ much. Thus, regardless of the values of the switching thresholds, RIMAP performance remains constant, closely following RIBB and RIPF performances. The results for topologies of Type 0 and 2 of Scenario B are also omitted because in this more sparse scenarios, the RIMAP swithching mechanism needs higher values of the number of neighbors threshold to actually switch from RIBB to RIPF, since the more connected the topology is, the more the number of neighbors a node will have, which increases the minimum number of neighbors in the topology. While the threshold is lower than the minimum number of neighbors, i.e., every node has more neighbors than the threshold, all nodes will use the RIBB polling discipline. Therefore, a higher value for the number of neighbors threshold is needed to switch the polling discipline, in order to perceive any change in network performance. Basically, the graphics for the topologies of Type 0 and 2 of Scenario B just look like they have been "shifted" to right. And we should evaluate the performance utilizing higher thresholds in order to visualize the changes in the performance.

7.2.1 Average Point-to-Point Delay per DATA frame

Figure 7.2 depicts the average point-to-point delay per DATA frame obtained for all MAC protocols. As we can see, the average delay of all receiver-initiated MAC protocols is lower than the delay of the sender-initiated IEEE 802.11 DCF MAC (14.6% gain for RIPF, and 21.2% for RIBB). As far as RIMAP threshold values are concerned, the higher the value of nN, the closer RIMAP gets to RIPF's performance (as a result of more nodes executing the PF polling discipline in the network). Conversely, the lower the value of nN, the closer RIMAP gets to RIBB's performance, since more nodes switch to the LSH discipline in the network. In fact, with the combined use of the sT and nN thresholds, RIMAP achieves the lowest delay among all protocols, when $nN = 0$ and sT ranges from 1 to 1,000 (0 dB to 30 dB). Regarding the impact of sT on delay, it is clear that it is not very significant, since the performance corresponding to different threshold values do not differ considerably (for a fixed value of nN). The only difference being the case when $sT = 10{,}000$ (40 dB), where few nodes switch to the LSH discipline and, thus, RIMAP performs very close to RIPF for all values of the nN threshold.

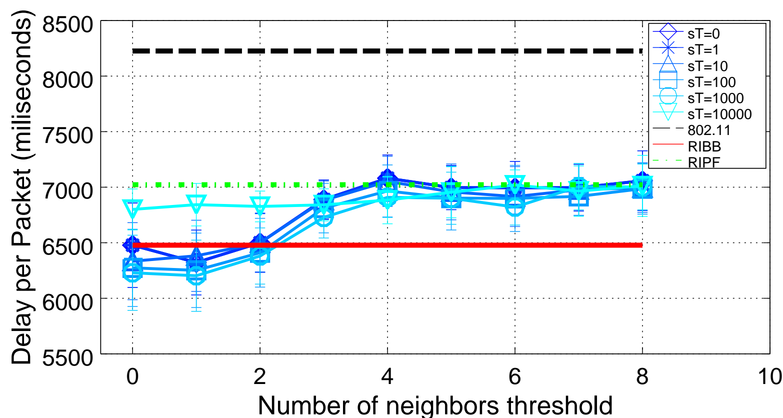


Figure 7.2: Average delay per data frame.

7.2.2 Average Throughput per Flow

Figure 7.3 shows the average flow throughput, which is computed across all data flows established between all pair of nodes (recall that each node generates data packets addressed to a third of their neighbors). The effect of the nN threshold on average throughput is similar to the effect on delay: the higher its value, the closer RIMAP gets to RIPF's performance, and vice-versa. However, differently from the case of delay, the sT threshold has a significant impact on throughput, especially when nN is low. In this case, as sT increases, a large number of nodes use the PF discipline (only a few can switch to LSH). Hence, in spite of perceiving a high SNR variance among neighbors, many nodes insist on polling their neighbors in a fairly manner, which induces a higher rate of failed polling attempts to distant neighbors, decreasing throughput. But, as sT decreases (and nN is kept low), a larger fraction of the nodes switch to LSH, which prioritizes the polling of nodes with better channel conditions, resulting in higher average throughput. Notice that, in such cases, RIMAP is similar to both RIBB and IEEE 802.11. Finally, as nN increases, few nodes

may switch to LSH, and irrespective of SNR variance, they are all forced to proceed with the PF discipline (which explains the convergence to RIPF performance).

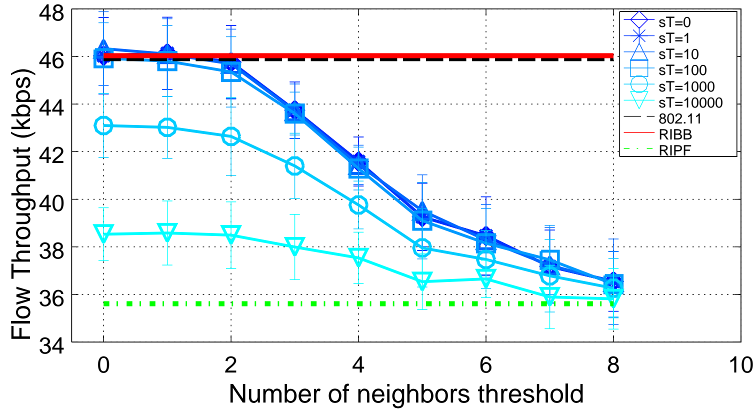


Figure 7.3: Average flow throughput.

7.2.3 Fairness

Figure 7.4 contains the results for fairness. We can see that RIPF performs better than RIBB and IEEE 802.11, whereas RIMAP can deliver the highest fairness, provided that appropriate threshold values are used. In the studied scenarios, RIMAP achieves its best performance when $nN = 4$, and sT ranges from 0 to 100 (i.e., SNR values differ by up to 10 dB from average SNR). In this case, nodes with up to four neighbors and low- to mid-range SNR variance use PF, whereas nodes with higher number of neighbors, and SNR variance, switch to LSH, which guarantees the prioritization of nodes with better channel conditions. In fact, under the BEB-based MAC protocol, the sooner a node releases the channel, the better, because it can start polling other neighbors, and neighboring nodes may resume their polling activity, as well. The pairing of these two factors explain the highest fairness achieved by RIMAP on the selected threshold values.

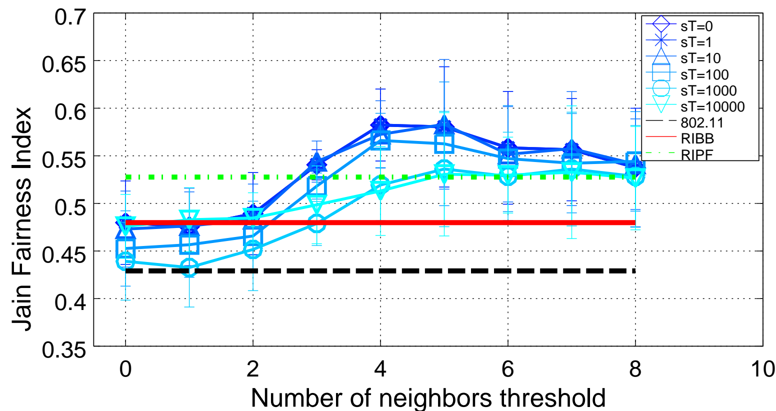


Figure 7.4: Jain Fairness Index.

7.3 Conclusion

This chapter introduced the Receiver-Initiated MAC with Adaptive Polling Discipline (RIMAP), a unicast MAC protocol that dynamically selects a polling discipline according to channel contention and link quality homogeneity to all neighbors. For that, two polling disciplines were considered: one that prioritizes nodes according to the *likelihood of successful handshake* (LSH), and the other that targets throughput fairness among nodes, the *proportional fair* (PF) discipline. The adaptive behavior is controlled by two switching parameters that can be tuned to trade off fairness with throughput-delay performance: number of neighbors, and perceived SNR variance among neighbors. Simulation results with topologies under hidden terminals, concurrent transmissions, and saturated traffic have showed that RIMAP can deliver lower delays and higher fairness than the polling disciplines by themselves (i.e., LSH and PF only, with gains up to 9.4% higher than PF), as well as the IEEE 802.11 DCF (up to 31.8% higher), at the expense of relatively small losses in throughput (up to 8.6% lower than LSH).

Simulation results show that, for our scenario, tuning the number of neighbor threshold to about 4 neighbors and the SNR variance threshold to the lower values (0 to 10), it is possible to reach a middle ground performance between RIPF and RIBB results in delay, but with a higher performance in fairness, and without losing too much in throughput. However, these threshold values are strictly related to the topology characteristics such as the node sparsity and channel quality.

Chapter 8

Conclusions

The first part of this dissertation presented a performance analysis of a random-access receiver-initiated MAC protocol that utilizes a reversed version of the binary exponential backoff (BEB) algorithm of the IEEE 802.11 DCF as a means to self-regulate and control the rate at which a node polls its neighbors. The use of the BEB algorithm indirectly takes into account the perceived level of contention, channel state, and DATA frame availability at polled nodes. The proposed receiver-initiated MAC protocol is also enhanced by allowing frame reordering at transmit queues, and the incorporation of the *nothing-to-send* (NTS) control frame, which helps on speeding up polling rounds (i.e., a node that receives an NTS from a polled node may immediately switch to the next neighbor in its neighborhood table). To suplement the polling rate control mechanism, we also introduced an enhanced version of a polling discipline that prioritizes the polling of nodes according to the likelihood of successful handshake. In addition to this polling discipline, we also investigated the traditional round-robin scheme, and a variant of the proportional fair scheduling mechanism typical of 4G networks. The performance of the receiver initiated MAC protocol with each of the polling mechanisms (which is named RIBB, RIRR, and RIPF) was compared to the performance of the sender-initiated IEEE 802.11 DCF with respect to MAC-level control overhead, delay, fairness, and throughput. Using a discrete-event simulator, we compared the performance of all protocols under two traffic scenarios in network topologies with different sparsity characteristics.

Regarding MAC overhead, we have observed that the receiver-initiated protocol has a lower control overhead due to the reduced number of control frames in a handshake. However, one can see that RIRR and RIPF require up to four times more control frames to transmit one DATA frame than IEEE 802.11, in some cases. On the other hand, RIBB can keep up with IEEE 802.11 performance, and even achieving better performance in more sparse topologies, as the network contention becomes less influent. This is a consequence of the priorization of neighbors with higher likelihood of successful handshake, as the polling node potentially do not waste the usage of control frames to poll neighbors with bad channel conditions or with no data frames to respond.

In general, we could observe that, as far as delay is concerned, the receiver-initiated protocols (RIRR, RIPF, and RIBB) performed better than IEEE 802.11 across all traffic scenarios and topologies. This is a direct consequence of the frame reordering technique and the introduction

of the NTS control frame. In particular, RIBB delivers the best performance among all. When traffic is not homogeneously distributed among neighbors, RIBB learns the neighbors that actually have DATA frames addressed to it, and prioritizes the ones which are also under relatively good channel conditions. This does not happen with RIRR and RIPF, which either treats all nodes equally (RIRR) or keeps trying to boost the performance of nodes under unfavorable conditions (RIPF).

Regarding fairness, RIPF and RIRR prevailed in this category, as expected, because of their inherent properties. Under fully-connected scenarios (or low sparsity), IEEE 802.11 is very unfair, especially when traffic is not homogenously distributed among nodes. Because of the prioritization scheme incorporated into RIBB, it does not perform as well as RIRR and RIPF in homogeneous traffic scenarios (although it outperforms IEEE 802.11 in more sparse networks), but it is certainly better than IEEE 802.11 in every topology when traffic is not equally distributed among nodes. Here, it is interesting to notice that the well-known fairness issues of the IEEE 802.11 BEB algorithm are less pronounced in its reversed version because, even though a node may still dominate channel acquisition more than its neighbors, polled nodes may vary completely, depending on the polling discipline. This is in direct contrast to sender-initiated MAC protocols with FIFO queue discipline, where a node not only may dominate channel acquisition, but it may also "lock" on a specific receiver due to a stream of successive same-destination DATA frames in its transmit queue. This phenomenon is well illustrated by the low fairness values obtained by IEEE 802.11 DCF in the studied scenarios, especially under less sparse topologies.

Finally, regarding throughput, we could observe that the losses in performance of receiver-initiated MAC protocols with respect to IEEE 802.11 were not very high, especially for RIBB, which could closely follow IEEE 802.11 in both types of traffic scenarios. RIRR and RIPF outperform IEEE 802.11 in more sparsed scenarios, and when all neighbors are potential transmitters (Scenario A), as RIBB as well. However, in Scenario B, where only one third of neighbors have data to transmit, RIRR and RIPF performance degrade considerably, in contrast to RIBB outperforming IEEE 802.11 in some cases, which is a combined effect of a lower number of control frames, the use of NTS and frame reordering, and the adaptive learning of DATA frame availability at neighboring nodes (queues are finite, and they may not contain packets to some nodes, occasionally). Motivated by these results, we sought the design of a polling discipline that balances the features of RIPF and RIBB according to the dynamics of the network topology and traffic, adaptively using two polling disciplines that can be tuned to trade off fairness with throughput-delay performance.

The second part introduced the Receiver-Initiated MAC with Adaptive Polling Discipline (RIMAP), a unicast MAC protocol that dynamically selects a polling discipline according to channel contention and link quality homogeneity to all neighbors. For that, two polling disciplines were considered: one that prioritizes nodes according to the *likelihood of successful handshake* (LSH), and the other that targets throughput fairness among nodes, the *proportional fair* (PF) discipline. The adaptive behavior is controlled by two switching parameters that can be tuned to trade off fairness with throughput-delay performance: the number of neighbors, and the perceived SNR variance among neighbors. If both estimated SNR variance and number of neighbors are higher than their respective thresholds, the node switches to LSH discipline. Otherwise, the node switches

to PF discipline.

This adaptive scheme allows a given node to utilize the most approriate polling discipline according to the current network condition perceived by itself. Since each node has a unique perception of the network, the individual gains obtained by using different polling disciplines over the time and over the network are summed to achieve an overall network gain. Simulation results with topologies under hidden terminals, concurrent transmissions, and saturated traffic have showed that RIMAP can deliver lower delays and higher fairness than the polling disciplines by themselves, as well as the IEEE 802.11 DCF, at the expense of relatively small losses in throughput.

Most of the works concerning receiver-initiated MAC protocols are targeted to wireless sensor networks (WSN), and there is a great effort on providing energy efficiency on this type of network. Although our protocol is not specific in addressing this issue, some features may indirectly alleviate the energy consumption: the NTS frame that allows a polled node to immediately switch to the next neighbor; the frame reordering technique that allows a frame to be transmitted at the time it is polled (instead of waiting until it reaches the head of the queue), and the prioritization of polling nodes in the LSH discipline. All these features may avoid the polled node to waste energy on useless handshakes.

This work presented a receiver-initiated paradigm for MAC protocol that have competitive qualities which directly confronts the sender-initiated paradigm. The sender-initiated paradigm is widely adopted by its success in the last decades. However, this paradigm jeopardizes the throughput fairness of the network due to its own backoff mechanism, which favors the node that recently had a successful handshake, while the others are fated to have longer waiting before they can access the channel. And also because there is this favored node in a sender-initiated handshake, the channel is accessed by a single data flow that will likely gain the right to access the channel again, if it has more packets in the data stream. In the other hand, in the receiver-initiated handshake, even if the same node wins the channel access every time, it may communicate with different neighbors at each time, giving the opportunity to diversify the data flows over the channel. Thus, the receiver-initiated paradigm diminishes the unfairness effect of the backoff algorithm by reverting it. Furthermore, the polling discipline distributes the sharing of the channel among the neighbor nodes. Therefore, the use of the receiver-initiated paradigm is crucial for wireless ad hoc networks whose applications demand a high accesibility (more users are able to send their data with a minimum service level, instead of a few users with high data throughput), provided mainly by the MAC protocol. Besides, the minimum service level can be incresed by improving the data rate of the link in the physical layer, whose capacities are enhancing over the years.

8.1 Future Work

For future work, we will investigate the impact of RIBB, RIRR, RIPF, and RIMAP on routing protocols and under mobility. The creation of routes across multiple hops may affect polling priorities significantly, and the adaptive learning of RIBB must be able to keep track of changes under mobility. Besides, this approach may require a cross-layer design, since the polling discipline

may gather the routing information in order to weight the priorization scheme of the polling neighbor decision. Furthermore, we will investigate the impact on the energy efficiency, since there are ad hoc network applications working over nodes powered by batteries with limited life time. And, concerning the wide utilization of receiver-initiated MAC protocols in WSNs, we will seek to adapt the proposed protocol to sensor networks, including the mechanisms of duty cycles, energy harvesting, and sleeping cycles, for instance.

Regarding the polling discipline, the Quality of Service (QoS) issue has an important weight in the decision of which neighbor the node should poll. Different priorities of services will be available across the neighbors, and the polling node must handle the polling prioritizations according to the transmitters traffic demand. Since the polling node, a priori, does not have the information of the QoS in the potential transmitters, the node must have a mechanism that predicts or discovers the QoS information. Analogously, there is the sender-initated IEEE 802.11e standard with the Enhanced Distributed Channel Access (EDCA) method, where high-priority traffic has a higher chance of being sent than low-priority traffic: a station with high priority traffic waits a little less before it sends its packet, on average, than a station with low priority traffic. Reverting the paradigm, neighbors with high-priority traffic should have a higher chance of being polled than neighbors with low-priority traffic. The IEEE 802.11e standard is considered of critical importance for delay-sensitive applications, such as Voice over Wireless LAN and streaming multimedia, which are very popular applications. Therefore, the development of reversed version of the EDCA is crucial to help the receiver-initiated paradigm to be widely adopted.

As far as polling rate is concerned, we should seek alternative methods to optimize the polling rate control, methods that better reflect the channel contention conditions, and also considers QoS. In sender-initiated paradigm, there is an effort to improve the access delay and fairness of the BEB algorithm [36, 37, 38]. In general, the approach is to modify the backoff window size growth rate, in order to balance between throughput and delay performance, given by the fact that, with a faster growth rate, the network is better capable of absorbing the mounting contention. On the other hand, it may lead to a more severe delay jitter due to a larger difference of backoff window sizes between a fresh packet and a deeply backlogged one [39]. In receiver-initiated paradigm, we must investigate the effect of modifying the window size growth rate, since there is the problem of a polling rate that is too slow, which leads to higher delay levels, and if it is too fast, which leads to higher chances of occurring a collision of polling packets.

This work is an intermediate step for studying the Multi-Packet Reception (MPR) in ad hoc networks. The MPR is a technique that allows a node to receive multiple packets simultaneously from different sources. For this, it may be employed the use of Code Division Multiple Access (CDMA) [40] or Multiple-In Multiple-Out (MIMO) antenas schemes [25]. In this last case, the possibility of ad hoc networks becoming more scalable would not be limited by the multiple access interference (MAI), but by the complexity of transmitters and receivers. They had proven that the channel utilization in IEEE 802.11 networks could be significantly improved by the MPR mechanisms, in which the basics behind are the utilization of the receiver-initiated protocol as a way to locally syncronize the nodes involved in the multiple transmission/reception of packets. In this context, Bonfim and Carvalho proposed a Receiver-Initiated Multi-Packet MAC protocol,

denominated RIMP-MAC [41]. Although the syncronization issue has been addressed, their work also addressed the lack of an approriate polling disciplines and a realistic mechanism of polling rate control, whose descriptions we presented in this dissertation. Bonfim and Carvalho developed an analytical model in order to evaluate the RIMP-MAC performance, and our future work is to implement this protocol in a discrete-event network simulator based on the implementation already done in this work. Therefore, we could evaluate the RIMP-MAC performance in more complex scenarios with the presence of hidden and exposed terminals, concurrent transmission, mobility, and routing.

Further, we could extend the receiver-initiated MAC protocol for the utilization of operation dynamic spectrum access and multiple channel scenarios. Today, there is a large demand of data services using 3G and LTE, and other technologies using ISM band such as WiFi and Bluetooth. However, there is a bad allocation of band resources, which makes the spectrum looks scarce from the user point of view. This spectrum sub-utilization leads to the necessity of planning protocols of Dynamic Spectrum Access (DSA). One approach of DSA protocols is the Opportunistic Spectrum Access (OSA), where it is imposed restrictions of *when* and *where* the users could transmit. This approach focuses mainly in the idle spaces (spatial and temporal) of the spectrum, allowing opportunistic users to identify and to explore the available spaces in the spectrum in a non-intrusive manner. In this context, Oliveira and Carvalho proposed the Opportunistic Channel Aggregation MAC protocol (OCA-MAC) [42] for wireless ad hoc networks. OCA-MAC allows opportunistic MAC-level channel aggregation per frame transmission avoiding the use of an extra control channel for coordination among nodes. The extension of the receiver-initiated paradigm is given by the fact that the receiver is the best positioned to know when and where the spectrum space will be available for itself. Therefore, the receiver node should be the first to announce this information to the potential transmitters. For multiple channel scenarios, the LSH discipline can be extended to, not only prioritize specific neighbors, but prioritize the best channels available too. Simmilarly to the estimation of the probability of successful handshake of a given neighbor, the information of the estimated probability of successful transmission in a given channel could be added to weight the decision of which channel will be use for polling.

This MAC protocol could be useful in applications such as vehicular ad hoc networks (VANETs), where the vehicular nodes must communicate with each other with minimum delay levels since the messages bring crucial information about traffic safety, for instance. Also, vehicular applications may not need to send long messages with a high data rate, but it may need that all vehicular nodes have a fair use of the channel, in order to allow more nodes to send their short and frequent messages. Other interesting application for the receiver-initiated MAC protocol is the networks of robots, where autonomous machines cooperates with each other in order to accomplish a task in a distributed manner. Each robotic node transmits its current status or commands, for instance, in short messages. At the same time, the nodes are frequently asking for new commands and the status of the other robotic nodes, in order to perform their own operations.

REFERENCES

[1] HOEBEKE, J.; MOERMAN, I.; DHOEDT, B.; DEMEESTER, P. An overview of mobile ad hoc networks: applications and challenges. *JOURNAL OF THE COMMUNICATIONS NETWORK*, v. 3, n. 3, p. 60–66, 2004.

[2] BHUSHAN, N.; LI, J.; MALLADI, D.; GILMORE, R.; BRENNER, D.; DAMNJANOVIC, A.; SUKHAVASI, R.; PATEL, C.; GEIRHOFER, S. Network densification: the dominant theme for wireless evolution into 5g. *IEEE Communications Magazine*, v. 52, n. 2, p. 82–89, 2014.

[3] TEHRANI, M. N.; UYSAL, M.; YANIKOMEROGLU, H. Device-to-device communication in 5g cellular networks: challenges, solutions, and future directions. *Communications Magazine, IEEE*, IEEE, v. 52, n. 5, p. 86–92, 2014.

[4] CHOUDHARY, M.; SHARMA, P.; SANGHI, D. Secure multicast model for ad-hoc military networks. In: IEEE. *Networks, 2004.(ICON 2004). Proceedings. 12th IEEE International Conference on.* [S.l.], 2004. v. 2, p. 683–688.

[5] TAKAHASHI, T.; KITAMURA, Y.; MIWA, H. Organizing rescue agents using ad-hoc networks. In: *Highlights on Practical Applications of Agents and Multi-Agent Systems*. [S.l.]: Springer, 2012. p. 139–146.

[6] HARTENSTEIN, H.; LABERTEAUX, K. P. A tutorial survey on vehicular ad hoc networks. *Communications Magazine*, IEEE, v. 46, n. 6, p. 164–171, 2008.

[7] YICK, J.; MUKHERJEE, B.; GHOSAL, D. Wireless sensor network survey. *Computer networks*, Elsevier, v. 52, n. 12, p. 2292–2330, 2008.

[8] KAISER, A.; ACHIR, N.; BOUSSETTA, K. Multiplayer games over wireless ad hoc networks: energy and delay analysis. In: IEEE. *Ultra Modern Telecommunications & Workshops, 2009. ICUMT'09. International Conference on.* [S.l.], 2009. p. 1–7.

[9] HEUVEN, D. *Opportunistic sensing & Aggregation network Using Smartphones and Sensor nodes.* Dissertation (Master) — University of Twente, 2014.

[10] CHIN, A.; ZHANG, D. *Mobile Social Networking.* [S.l.]: Springer, 2014.

[11] BONTA, J. D.; CALCEV, G.; JR, B. J. F.; MANGALVEDHE, N. R.; SMITH, N. J. *System for enabling mobile coverage extension and peer-to-peer communications in an ad hoc network and method of operation therefor.* [S.l.]: Google Patents, nov. 29 2011. US Patent 8,068,454.

[12] KALEJAIYE, G. B.; RONDINA, J. A.; ALBUQUERQUE, L. V.; PEREIRA, T. L.; CAMPOS, L. F.; MELO, R. A.; MASCARENHAS, D. S.; CARVALHO, M. M. Mobile offloading in wireless ad hoc networks: The tightness strategy. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 44, n. 3, p. 96–102, jul. 2014. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/2656877.2656891>.

[13] PELUSI, L.; PASSARELLA, A.; CONTI, M. Opportunistic networking: data forwarding in disconnected mobile ad hoc networks. *Communications Magazine, IEEE*, IEEE, v. 44, n. 11, p. 134–141, 2006.

[14] TALUCCI, F.; GERLA, M. MACA-BI (MACA by invitation). a wireless MAC protocol for high speed ad hoc networking. In: IEEE. *Universal Personal Communications Record, 1997. Conference Record., 1997 IEEE 6th International Conference on.* [S.l.], 1997. v. 2, p. 913–917.

[15] GARCIA-LUNA-ACEVES, J. J.; TZAMALOUKAS, A. Reversing the collision-avoidance handshake in wireless networks. In: ACM. *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking.* [S.l.], 1999. p. 120–131.

[16] MORAES, R. M. D.; GARCIA-LUNA-ACEVES, J. Receiver-initiated collision avoidance in multi-hip ad hoc networks. In: CITESEER. *Communications in Computing.* [S.l.], 2004. p. 220–226.

[17] WANG, Y.; GARCIA-LUNA-ACEVES, J. J. A hybrid collision avoidance scheme for ad hoc networks. *Wireless Networks*, Springer, v. 10, n. 4, p. 439–446, 2004.

[18] SUN, Y.; GUREWITZ, O.; JOHNSON, D. B. RI-MAC: a receiver-initiated asynchronous duty cycle MAC protocol for dynamic traffic loads in wireless sensor networks. In: ACM. *Proceedings of the 6th ACM conference on Embedded network sensor systems.* [S.l.], 2008. p. 1–14.

[19] EU, Z. A.; TAN, H.-P. Probabilistic polling for multi-hop energy harvesting wireless sensor networks. In: IEEE. *Communications (ICC), 2012 IEEE International Conference on.* [S.l.], 2012. p. 271–275.

[20] HU, Q.; TIAN, Q.; TANG, Z. RP-MAC: A passive MAC protocol with frame reordering for wireless sensor networks. *International journal of wireless information networks*, Springer, v. 20, n. 1, p. 74–80, 2013.

[21] RAMANATHAN, S. A unified framework and algorithm for channel assignment in wireless networks. *Wireless Networks*, Springer-Verlag New York, Inc., v. 5, n. 2, p. 81–94, 1999.

[22] RAJENDRAN, V.; OBRACZKA, K.; GARCIA-LUNA-ACEVES, J. J. Energy-efficient, collision-free medium access control for wireless sensor networks. *Wireless Networks*, Springer-Verlag New York, Inc., v. 12, n. 1, p. 63–78, 2006.

[23] BONFIM, T. S.; CARVALHO, M. M. Reversing the IEEE 802.11 backoff algorithm for receiver-initiated MAC protocols. In: IEEE. *Wireless Communications and Mobile Computing Conference (IWCMC), 2012 8th International.* [S.l.], 2012. p. 269–274.

[24] THE ns-3 Network Simulator. `http://www.nsnam.org`.

[25] LAL, D.; TOSHNIWAL, R.; RADHAKRISHNAN, R.; AGRAWAL, D.; JR, J. C. A novel MAC layer protocol for space division multiple access in wireless ad hoc networks. In: IEEE. *Computer Communications and Networks, 2002. Proceedings. Eleventh International Conference on*. [S.l.], 2002. p. 614–619.

[26] SU, Y.-S.; SU, S.-L.; LI, J.-S. Receiver-initiated multiple access protocols for spread spectrum mobile ad hoc networks. *Computer Communications*, v. 28, n. 10, p. 1251 – 1265, 2005. ISSN 0140-3664. Performance issues of Wireless LANs, PANs and ad hoc networks.

[27] TAKATA, M.; BANDAI, M.; WATANABE, T. A receiver-initiated directional MAC protocol for handling deafness in ad hoc networks. In: *Communications, 2006. ICC'06. IEEE International Conference on*. [S.l.: s.n.], 2006. v. 9, p. 4089–4095.

[28] DUTTA, P.; DAWSON-HAGGERTY, S.; CHEN, Y.; LIANG, C.; TERZIS, A. Design and evaluation of a versatile and efficient receiver-initiated link layer for low-power wireless. In: ACM. *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*. [S.l.], 2010. p. 1–14.

[29] LIANG, H.; ZHUANG, W. Double-loop receiver-initiated MAC for cooperative data dissemination via roadside WLANs. *Communications, IEEE Transactions on*, v. 60, n. 9, p. 2644–2656, September 2012. ISSN 0090-6778.

[30] LEONARDI, A.; PALAZZO, S.; RAMETTA, C.; KNIGHTLY, E. W. A new adaptive receiver-initiated scheme for mitigating starvation in wireless networks. *Ad Hoc Networks*, Elsevier, v. 11, n. 2, p. 625–638, 2013.

[31] PU, L.; LUO, Y.; PENG, Z.; MO, H.; CUI, J.-H. Traffic estimation based receiver initiated MAC for underwater acoustic networks. In: ACM. *Proceedings of the Eighth ACM International Conference on Underwater Networks and Systems*. [S.l.], 2013. p. 7.

[32] HAVERKORT, B. R. *Performance of Computer Communication Systems: A Model-Based Approach*. [S.l.]: John Wiley & Sons, 1998.

[33] KUSHNER, H.; WHITING, P. Convergence of proportional-fair sharing algorithms under general conditions. *Wireless Communications, IEEE Transactions on*, v. 3, n. 4, p. 1250–1259, July 2004. ISSN 1536-1276.

[34] TOH, C.-K.; VASSILIOU, V.; GUICHAL, G.; SHIH, C.-H. MARCH: a medium access control protocol for multihop wireless ad hoc networks. In: IEEE. *MILCOM 2000. 21st Century Military Communications Conference Proceedings*. [S.l.], 2000. v. 1, p. 512–516.

[35] JAIN, R.; CHIU, D.-M.; HAWE, W. R. *A quantitative measure of fairness and discrimination for resource allocation in shared computer system*. [S.l.]: Eastern Research Laboratory, Digital Equipment Corporation Hudson, MA, 1984.

[36] KUPTSOV, D.; NECHAEV, B.; LUKYANENKO, A.; GURTOV, A. How penalty leads to improvement: A measurement study of wireless backoff in ieee 802.11 networks. *Computer Networks*, Elsevier, 2014.

[37] CHEN, W.-T.; JIAN, B.-B.; LO, S.-C. An adaptive retransmission scheme with qos support for the ieee 802.11 mac enhancement. In: IEEE. *Vehicular Technology Conference, 2002. VTC Spring 2002. IEEE 55th.* [S.l.], 2002. v. 1, p. 70–74.

[38] NASIR, Q.; ALBALT, M. History based adaptive backoff (hbab) ieee 802.11 mac protocol. In: IEEE. *Communication Networks and Services Research Conference, 2008. CNSR 2008. 6th Annual.* [S.l.], 2008. p. 533–538.

[39] SUN, X.; DAI, L. Backoff design for ieee 802.11 dcf networks: Fundamental tradeoff and design criterion. *IEEE Trans. Networking.*

[40] MERGEN, G.; TONG, L. Receiver controlled medium access in multihop ad hoc networks with multipacket reception. In: IEEE. *Military Communications Conference, 2001. MILCOM 2001. Communications for Network-Centric Operations: Creating the Information Force. IEEE.* [S.l.], 2001. v. 2, p. 1014–1018.

[41] BONFIM, T. d. S. *RIMP: protocolo de controle de acesso ao meio com múltipla recepção de pacotes para redes ad hoc.* Dissertation (Master) — Universidade de Brasília, 2013.

[42] OLIVEIRA, L. M. E. d. *OCA-MAC: protocolo de controle de acesso ao meio com agregação oportunista de canal.* Dissertation (Master) — Universidade de Brasília, 2014.

APPENDIX

I. PERFORMANCE OF THE DYNAMIC POLLING DISCIPLINE IN RIMAP PROTOCOL FOR AD HOC NETWORKS

I.1 Average Point-to-Point Delay per DATA frame

I.1.1 Scenario A

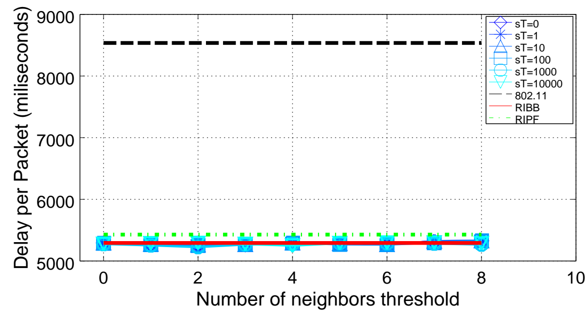


Figure I.1: Average delay per data frame in Scenario A topology Type 0 (fully-connected).

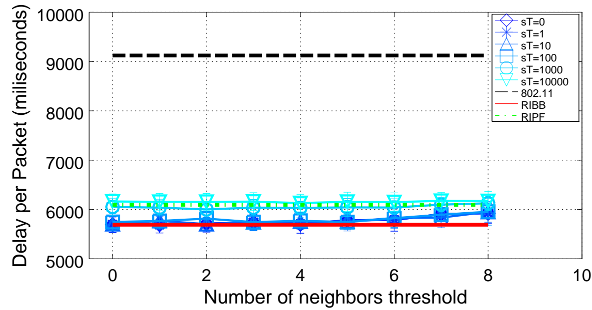


Figure I.2: Average delay per data frame in Scenario A topology Type 2.

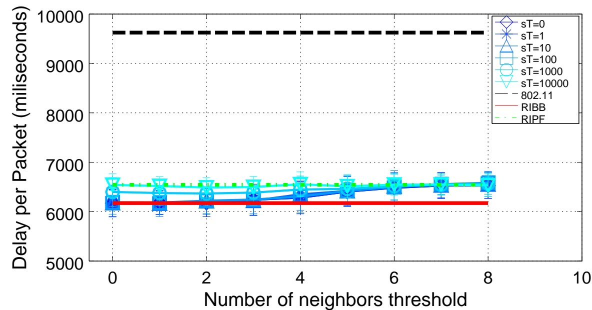


Figure I.3: Average delay per data frame in Scenario A topology Type 4.

I.1.2 Scenario B

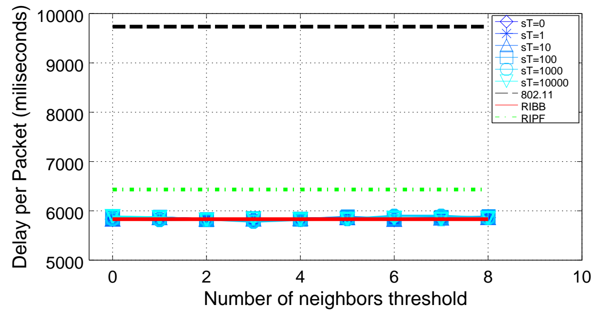


Figure I.4: Average delay per data frame in Scenario B topology Type 0 (fully-connected).

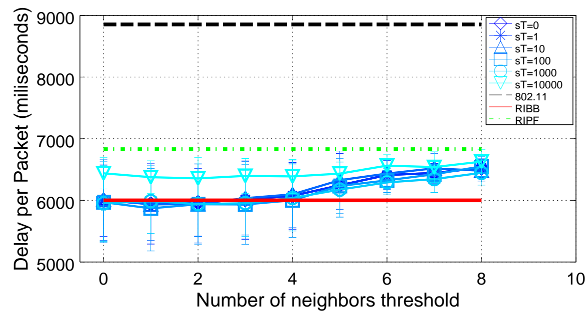


Figure I.5: Average delay per data frame in Scenario B topology Type 2.

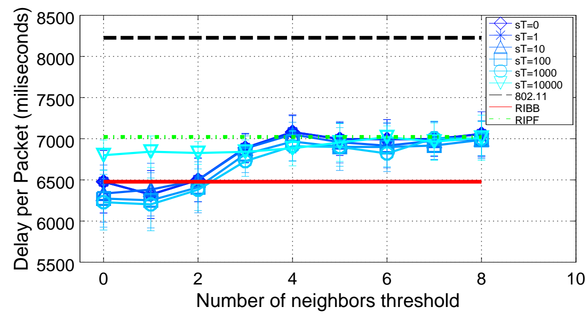


Figure I.6: Average delay per data frame in Scenario B topology Type 4.

I.2 Average Throughput per Flow

I.2.1 Scenario A

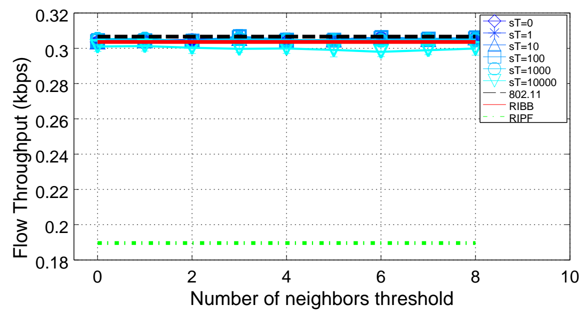


Figure I.7: Average throughput per flow in Scenario A topology Type 0 (fully-connected).

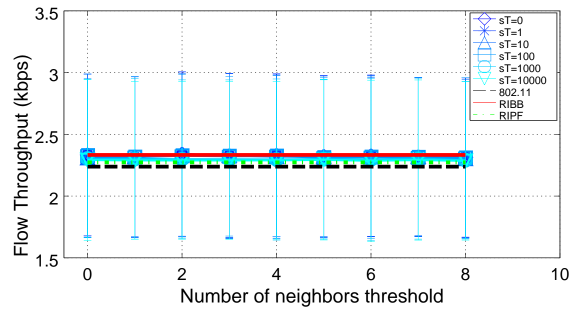


Figure I.8: Average throughput per flow in Scenario A topology Type 2.

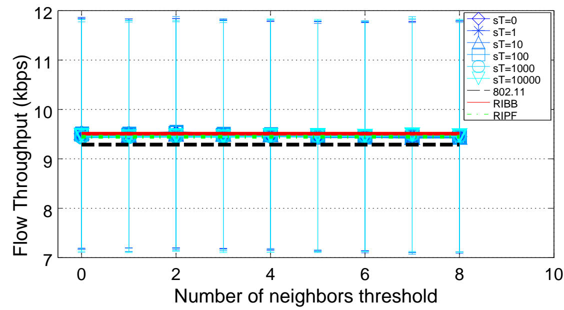


Figure I.9: Average throughput per flow in Scenario A topology Type 4.

I.2.2 Scenario B

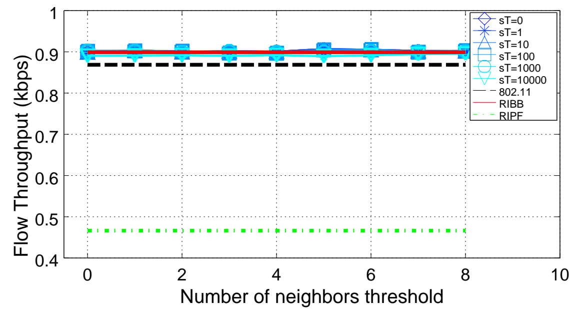


Figure I.10: Average throughput per flow in Scenario B topology Type 0 (fully-connected).

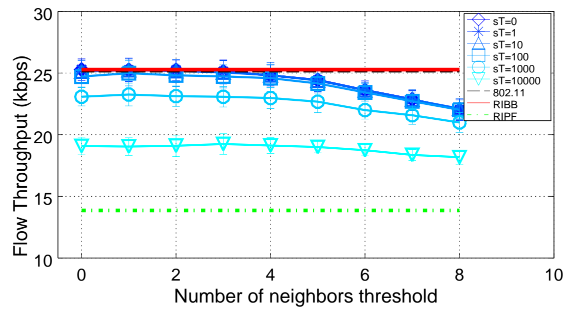


Figure I.11: Average throughput per flow in Scenario B topology Type 2.

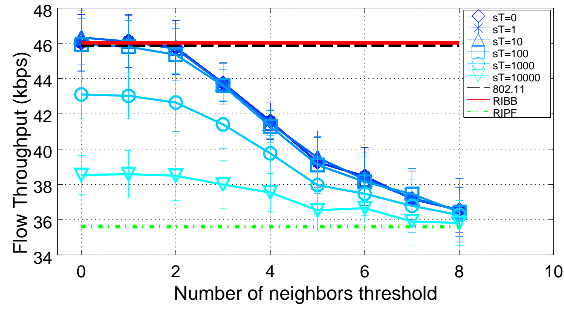


Figure I.12: Average throughput per flow in Scenario B topology Type 4.

I.3 Fairness

I.3.1 Scenario A

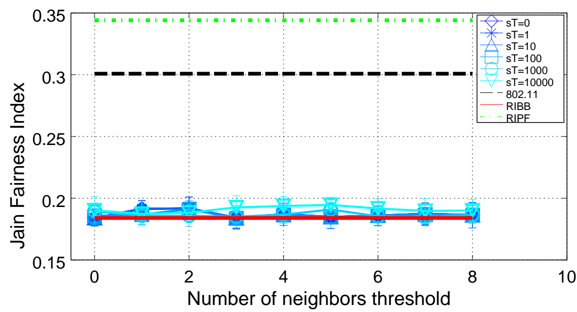


Figure I.13: Jain Fairness Index in Scenario A topology Type 0 (fully-connected).

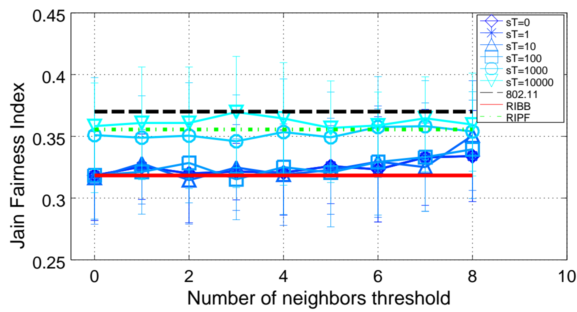


Figure I.14: Jain Fairness Index in Scenario A topology Type 2.

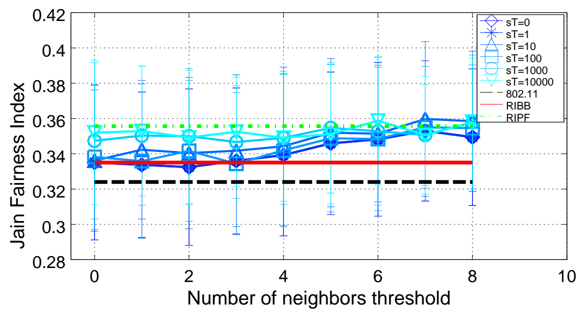


Figure I.15: Jain Fairness Index in Scenario A topology Type 4.

I.3.2 Scenario B

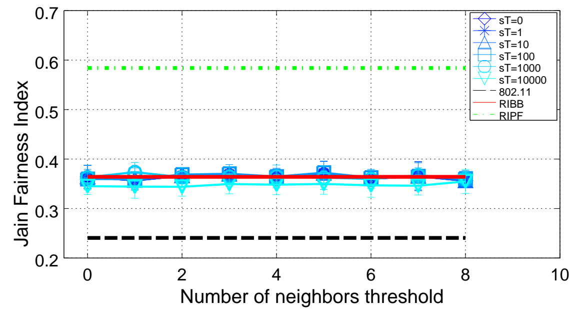


Figure I.16: Jain Fairness Index in Scenario B topology Type 0 (fully-connected).

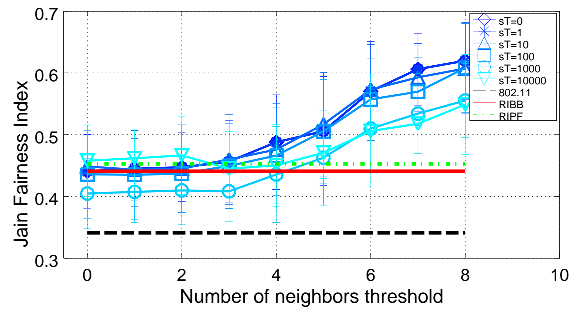


Figure I.17: Jain Fairness Index in Scenario B topology Type 2.

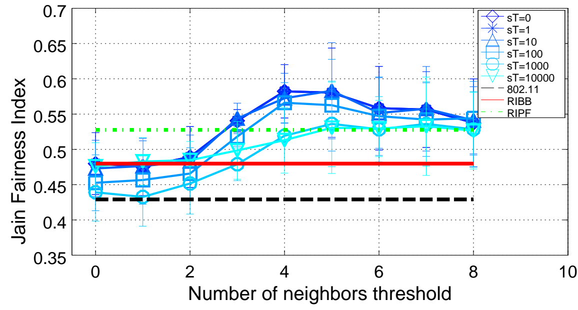


Figure I.18: Jain Fairness Index in Scenario B topology Type 4.

II. TOPOLOGY GENERATION

The topologies are generated considering 50 nodes distributed in a flat terrain of 800×800 m, with a transmission range of 150 m. The generation process allows to achieve different levels of sparsity, from fully-connected networks to more sparsed topologies. In order to achieve the different levels, we set two parameters to tune the sparsity of the network: the minimum number of neighbors $nmin$, and the number of divisions sec of the total area. It is expected that the higher is the minimum number of neighbors of a given topology, more connected is the network, i.e., a given node will have at least $nmin$ neighbors, and if this value is high, more neighbors the node will have. On the other hand, the lower is $nmin$, more sparse the network will be. The motivation of the number of divisions comes from the fact that the two-dimensional uniform distribution concentrates the nodes in the center of the area, leading to more connected networks. To circumvent this problem, we divide the total area in equal sectors, and uniformly distribute an equal number of nodes in each sector, to achieve higher sparsity of the network in the total area. Thus, tuning the number of sectors sec within the total area and the minimum number of neighbors $nmin$, we can generate topologies with different levels of sparsity. For instance, setting $nmin = 49$ (49 neighbors in a 50 node topology) and $sec = 1$ (one distribution in the whole area) will generate a fully-connected network. On the other side, setting $nmin = 2$ (few neighbors per node) and $sec = 16$ (dividing the total area in 16 equal parts) will generate a more sparse topology since it will distribute less nodes in a given region.

However, this process is not a precise function. Therefore, the topologies are generated by trial-and-error until the characteristics of the defined categories are achieved. We classify the topologies into six types, each one with a different average number of neighboring nodes, defined by the ratio called *number of hops*, which estimates the average number of hops in the topology. This ratio is defined as total number of nodes (minus 1) divided by the average number of neighbors of each node. Table II.1 shows the parameters utilized in each topology according to its category.

Type	**Average Number of Neighbors**	**Average Number of Hops**	**Minimum Neighbors**	**Sectors Defined**
0	49.0	1	49	1
1	24.5	2	11	1
2	12.2	4	6	1
3	8.1	6	4	4
4	6.1	8	3	25
5	4.9	10	2	25

Table II.1: Parameters for topology generation.

The distribution of the nodes is given in three steps:

1. Draw the (x,y) coordenates for every node, each node is uniformly distributed within one

sector after the other until all nodes are positioned.

```
int sx=0, sy=0;
for(int j=0; j<nodes; j++)
{
  x[j]= (uv.GetValue(sx*sec, (sx+1)*sec));
  sx = (sx + 1) % sectors;
  if (sx == (sectors-1)) sy = (sy + 1) % sectors;
  y[j]= (uv.GetValue(sy*sec, (sy+1)*sec));
}
```

2. Calculate the distances between all the nodes and verify if there is any node that does not have the minimum number of neighbors, i.e., if there is the minimum of nodes inside a given transmission range.

```
bool somaProbOk = false;
int it=0;
while (!somaProbOk)
{
  cout << "try " << it++ << endl;
  int somaProb = 0;
  for(int j=0; j<nodes; j++)
  {
    aux = 0;

    for(int k=0; k<nodes; k++) if(j != k)
    {
      dist[j][k] = sqrt(pow((x[j]-x[k]),2) + pow((y[j]-y[k]),2));
      if(dist[j][k] <= txrange) aux++;
    }

    if(aux < nmin)
    {
      prob[j] = 1;
      cout << "\t" << j << " fail" << endl;
    }
    else {prob[j] = 0;}
  }
```

3. Replace the nodes that did not present the minimum number of neighbors, distributing uniformly within the total area, until they obtain the minimum number of neighbors or more.

```
for(int j=0; j<nodes; j++)
  {
    aux=0;
    if(prob[j] == 1)
    {
      while(aux < nmin)
      {
x[j]= (uv.GetValue(0, limit));
y[j]= (uv.GetValue(0, limit));

for(int k=0; k<nodes; k++) if(j != k)
{
  dist[j][k] = sqrt(pow((x[j]-x[k]),2) + pow((y[j]-y[k]),2));
  if(dist[j][k] <= txrange) aux++;
}
      }
    }
  }

  for(int j=0; j<nodes; j++)
  {
    somaProb += prob[j];
  }
  if (somaProb == 0)
  {
    cout << "\tall ok" << endl;
    somaProbOk = true;
  }
}
```

After defining the position of the nodes and their respective neighbors, we define the pairs of data communication. The communication flows between only immediate neighbors, because we

are evaluating concerning link performance only. Thus, if a neighbor is within the transmission range, we define the pair of communication with a probability given by the proportion *prop* of neighbors with available data. Therefore, we can configure different traffic scenarios according to this proportion. For instance, in Scenario A, we set the probability as 1.0, so that all neighbors have a data flow to the given node. And in Scenario B, we set the probability as 0.333, so that only one third of the neighbors have data available to the given node.

```
for(int j=0; j<nodes; j++)
{
  viz[j]=0;
  cca[j]=0;
  for (int k=0; k<nodes; k++)
  {
    dist[j][k] = sqrt(pow((x[j]-x[k]),2) + pow((y[j]-y[k]),2));
    if(dist[j][k] <= txrange && j!=k)
    {
      mediaDistViz += dist[j][k];
      v_pairs++;
      viz[j]++;
      if (uv.GetValue(0,1) < prop)
      {
route << k+1 << " " << j+1 << " 0" << endl;
flow_pairs++;
      }
    }
    if(dist[j][k] <= txrange*1.5 && dist[j][k] > txrange && j!=k)
    {
      mediaDistCca += dist[j][k];
      c_pairs++;
      cca[j]++;
    }
  }
}
```

III. NS-3 MAIN SCRIPT

```
/* -*-  Mode: C++; c-file-style: "gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2011-2013 NERds GPDS UnB
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Author: Fadhil Firyaguna <firyaguna@ieee.org>
 */

/* FEATURES
 *
 * ----------- SCRIPT INPUTS -----------------
 * All variable parameters are in a 'sim_Info' struct with instance 'info'
 * and they are accessible by command line.
 * See //--- COMMAND LINE ---// section.
 *
 * ----------- TOPOLOGY READING ---------------
 * It reads a topology file ("topo.dat") and a link file ("routes.dat"),
 * varying from 1 to 10, that must be in this directory: "topology/tpX/",
 * where X is the number of the topology.
 *
 * The files must be in this standard:
 *
 * "topo.dat"
 * 526.0783 743.5801
 * 397.6912 435.3065
 * 183.8507 674.9844
 * ...
 * First column for x-axis, second column for y-axis.
 *
 * "routes.dat"
 * 1 61 0
 * 2 70 0
 * 3  1 0
 * ...
 * First column for destination node, second column for source node.
 * At each RngRun, the routes array order will be shuffled.
 *
 * ----------- PCAP (Wireshark) OUTPUT -------------
 * Enable pcap tracing (-tracing=1) to create pcap files of all devices
 * to open in Wireshark. By default, it always generate a pcap file for
 * device number 1. These files are saved in "tracing/" folder.
 *
 * Input a prefix name for the pcap files (-out="prefixName").
 * Default: "teste".
 *
 * ----------- SIMULATION RESULTS OUTPUT -----------
 * Enable final report output to be saved in a file (-save=1). The output
 * will be written in the end of file, so it would not be overwritten and
 * will keep all report history. The same final report will be printed in
 * the terminal. This file is saved in "tracing/output" if enabled.
 *
 *
 * ----------- SIMULATION SCENARIO --------------
 * Wifi Ad Hoc scenario based on 802.11b standard
 * Wifi Channel
 *  - constant propagation delay
 *  - Friis or TwoRay propagation loss
 *  - Rician fading (Nakagami loss)
 * Wifi Physical Layer
 *  - energy detection threshold as a function of distance
 *  - cca threshold proportional to energy threshold
 *  - constant rate 1 mbps
```

```
 * Mobility
 *  - constant position
 *
 * IP Configuration
 *  - base "10.1.1.0"
 *  - mask "255.255.255.0"
 *  - ARP cache is pre-populated before simulation starts
 *
 * Udp Socket Application
 *  - each application defined by "routes.dat" starts after 0.05s the
 *    previous application
 *  - constant bit rate 1 mbps
 *  - on and off times are exponentially distributed with mean 0.5s
 */

#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/config-store-module.h"
#include "ns3/mobility-module.h"
#include "ns3/wifi-module.h"
#include "ns3/random-variable.h"
#include "ns3/flow-monitor-module.h"
#include "ns3/applications-module.h"

#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <cstdio>
#include <cstdlib>
#include <stdexcept>
#include <ctime>
#include <iomanip>

NS_LOG_COMPONENT_DEFINE ("WifiAdhocControle");

using namespace ns3;
using namespace std;


struct sim_Info
{
  // simulation parameters
  int run; // simulation run
  double stop; // simulation time
  double warmUp; // network warm up time
  int nos; // number of nodes
  int rotas; // number of routes
  int top;  // topology number
  char topType; // topology type
  bool tracing; // enable pcap tracing
  string outfile; // output file name
  bool verbose; // enable all log
  bool save; // enable simulation report

  // physical layer parameters
  string phyMode;
  double k;  // rician factor
  double distance; // transmission range [meters]
  double factorCca; // cca range [times distance]
  float height; // antenna height
  double txPowerDbm; // tx power
  bool useTwoRay; // enable tworay propagation
  double noiseFig; // noise figure

  // mac layer parameters
  int ssrc; // ssrc counter
  double alpha; // success tx probability update rate
  int pollingMode; // polling discipline mode
  bool rima; // rima
  int neighborsThresh; // number of neighbors threshold
  double snrVarThresh; // snr variation threshold
  bool moreData; // enable moreData flag
  int cwmin; // minimum contention window
  int cwmax; // maximum contention window

  // application layer parameters
  double onTime; // exponential mean for ON time
  double offTime; // exponential mean for OFF time
  uint32_t packetSize; // packet size [bytes]
```

```
  sim_Info ()
  {
      // simulation parameters
      run = 7; // simulation run
      stop = 120.0; // simulation time
      warmUp = 20; // network warm up time
      nos = 0; // number of nodes
      rotas = 0; // number of routes
      top = 0; // topology number
      topType = 'b'; // topology type
      tracing = false; // enable pcap tracing
      outfile = "teste"; // output file name
      verbose = false; // enable all log
      save = false; // enable simulation report

      // physical layer parameters
      phyMode = "DsssRate1Mbps";
      k = 100; // rician factor
      distance = 150; // transmission range [meters]
      factorCca = 1.5; // cca range [times distance]
      height = 1.2; // antenna height
      txPowerDbm = 10; // tx power
      useTwoRay = true; // enable tworay propagation
      noiseFig = 10; // noise figure

      // mac layer parameters
      ssrc = 7; // ssrc counter
      alpha = 0.02; // success tx probability update rate
      pollingMode = 1; // polling discipline mode
      rima = true; // rima
      neighborsThresh = 0; // number of neighbors threshold
      snrVarThresh = 0; // snr variation threshold
      moreData = false; // enable moreData flag
      cwmin = 31; // minimum contention window
      cwmax = 1023; // maximum contention window

      // application layer parameters
      onTime = 0.3; // exponential mean for ON time
      offTime = 0.9; // exponential mean for OFF time
      packetSize = 1412; // packet size [bytes]
  }
} info;

struct sim_Result
{
  uint64_t sumRxBytesByFlow;
  uint64_t sumRxBytesQuadByFlow;
  uint64_t sumLostPktsByFlow;
  uint64_t sumRxPktsByFlow;
  uint64_t sumTxPktsByFlow;
  uint64_t sumDelayFlow;
  uint64_t nFlows;

       /* Throughput Average by Flow (bps) = sumRxBytesByFlow * 8 / (nFlows * time)
       * Throughput Quadratic Average by Flow (bps) = sumRxBytesQuadByFlow * 64 / (nFlows * time * time)
       * Net Aggregated Throughput Average by Node (bps) = sumRxBytesByFlow * 8 / (nodes * time)
       * Fairness = sumRxBytesByFlow^2 / (nFlows * sumRxBytesQuadByFlow)
       * Delay per Packet (seconds/packet) = sumDelayFlow / sumRxPktsByFlow
       * Lost Ratio (%) = 100 * sumLostPktsByFlow / sumTxPktsByFlow
       */
  double thrpAvgByFlow;
  double thrpAvgQuadByFlow;
  double thrpVarByFlow;
  double netThrpAvgByNode;
  double fairness;
  double delayByPkt;
  double lostRatio;

  sim_Result ()
  {
    sumRxBytesByFlow = 0;
    sumRxBytesQuadByFlow = 0;
    sumLostPktsByFlow = 0;
    sumRxPktsByFlow = 0;
    sumTxPktsByFlow = 0;
    sumDelayFlow = 0;
    nFlows = 0;
  }
} data;
```

```
void PopulateArpCache (void);
Vector GetPosition (Ptr<Node> node);
void LerTopologia (char *topo, char *routes);
double rxPowerDbm (double distance, double height, double txPowerDbm, bool useTwoRay);
void ComputeResults (void);

//node coordinates and routes
float *x;
float *y;
ifstream in;
char ch;
int *from;
int *to;

const double PI = 3.14159265358979323846;
const double lambda = (3.0e8 / 2.407e9);
const double freq = 2.407e9;

int
main (int argc, char *argv[])
{
//   LogComponentEnable ("AdhocWifiMac", LOG_LEVEL_FUNCTION);
//   LogComponentEnable ("MacLow", LOG_LEVEL_DEBUG);
//   LogComponentEnable ("DcaTxop", LOG_LEVEL_DEBUG);
//   LogComponentEnable ("DcaTxop", LOG_LEVEL_FUNCTION);
//   LogComponentEnable ("WifiRemoteStationManager", LOG_LEVEL_DEBUG);
//   LogComponentEnable ("DcfManager", LOG_LEVEL_ALL);
//   LogComponentEnable ("RegularWifiMac", LOG_LEVEL_DEBUG);

  //----------------------------COMMAND LINE-----------------------------------------//
    CommandLine cmd;
    // simulation parameters
    cmd.AddValue ("run", "seed", info.run);
    cmd.AddValue ("stop", "stop simulation at this time in seconds", info.stop);
    cmd.AddValue ("warmup", "start monitoring network after this time in seconds", info.warmUp);
    cmd.AddValue ("top", "input topology file name", info.top);
    cmd.AddValue ("topType", "input topology type", info.topType);
    cmd.AddValue ("out", "output file name", info.outfile);
    cmd.AddValue ("tracing", "enable pcap tracing", info.tracing);
    cmd.AddValue ("verbose", "turn on all WifiNetDevice log components", info.verbose);
    cmd.AddValue ("save", "enable final report file output", info.save);
    cmd.AddValue ("rima", "rima", info.rima);

    // physical layer parameters
    cmd.AddValue ("phyMode", "Wifi Phy mode", info.phyMode);
    cmd.AddValue ("ricianK", "LOS and NLOS ratio", info.k);
    cmd.AddValue ("cca", "multiplier factor for CCA distance", info.factorCca);
    cmd.AddValue ("noise", "noise figure loss in dB", info.noiseFig);

    // mac layer parameters
    cmd.AddValue ("ssrc", "max ssrc", info.ssrc);
    cmd.AddValue ("alpha", "success tx probability update rate", info.alpha);
    cmd.AddValue ("poll", "polling discipline mode", info.pollingMode);
    cmd.AddValue ("neighborsThresh", "number of neighbors threshold", info.neighborsThresh);
    cmd.AddValue ("snrVarThresh", "snr variation threshold", info.snrVarThresh);
    cmd.AddValue ("moreData", "enable moreData flag", info.moreData);
    cmd.AddValue ("cwmin", "minimum contention window", info.cwmin);
    cmd.AddValue ("cwmax", "maximum contention window", info.cwmax);

    // application layer parameters
    cmd.AddValue ("onTime", "exponential mean for ON time", info.onTime);
    cmd.AddValue ("offTime", "exponential mean for OFF time", info.offTime);
    cmd.AddValue ("packetSize", "size of application packet sent", info.packetSize);
    cmd.Parse (argc, argv);
    //----------------------------fim-COMMAND LINE ------------------------------------//

    RngSeedManager::SetRun (info.run);

    char prefix[100];
    sprintf (prefix, "-top%d%crun%d-", info.top, info.topType, info.run);
    if (info.topType == 'a') info.outfile = "tracing/top-a/" + info.outfile;
    if (info.topType == 'b') info.outfile = "tracing/top-b/" + info.outfile;
    cout << "Tracing file: " << info.outfile << prefix << endl;

    char topofile[100]; //topology file "topo.dat" path name
    char routesfile[100]; //routes file "routes.dat" path name
    if (info.top == 0)
    {
      sprintf (topofile,"%s", "topology/teste/topo-teste.dat"); //topology file "topo.dat" path name
      sprintf (routesfile,"%s","topology/teste/routes-teste.dat"); //routes file "routes.dat" path name
    }
```

```
    else
    {
      sprintf (topofile,"%s%c%s%d%s","topology/top-",info.topType,"/tp",info.top,"/topo.dat");
      sprintf (routesfile,"%s%c%s%d%s","topology/top-",info.topType,"/tp",info.top,"/routes.dat");
    }
    LerTopologia (topofile, routesfile);

    // disable fragmentation for frames below 2200 bytes
    Config::SetDefault ("ns3::WifiRemoteStationManager::FragmentationThreshold", StringValue ("2200"));
    // turn off RTS/CTS for frames below 2200 bytes
    Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold", StringValue ("1000"));
    // Fix non-unicast data rate to be the same as that of unicast
    Config::SetDefault ("ns3::WifiRemoteStationManager::NonUnicastMode", StringValue (info.phyMode));
    if (info.rima)
    {
    // Change RTR and DATA retry counter
    Config::SetDefault ("ns3::WifiRemoteStationManager::MaxSsrc", UintegerValue (info.ssrc));
    Config::SetDefault ("ns3::WifiRemoteStationManager::MaxSlrc", UintegerValue (info.ssrc));
    // Change success tx probability update rate
    Config::SetDefault ("ns3::WifiRemoteStationManager::UpdateRate", DoubleValue (info.alpha));
    // Change polling discipline mode
      /* Polling modes:
      * 0 (default) = adaptive polling
      * 1 = alpha moving average (Tiago)
      * 2 = proportional fair
      * 3 = round robin
      */
    Config::SetDefault ("ns3::WifiRemoteStationManager::PollingMode", UintegerValue (info.pollingMode));
    Config::SetDefault ("ns3::WifiRemoteStationManager::nNeighborsThreshold", UintegerValue (info.neighborsThresh));
    Config::SetDefault ("ns3::WifiRemoteStationManager::SnrVarThreshold", DoubleValue (info.snrVarThresh));
    Config::SetDefault ("ns3::WifiRemoteStationManager::EnableMoreData", BooleanValue (info.moreData));
    }

    Config::SetDefault ("ns3::DcaTxop::CwMin", UintegerValue (info.cwmin - 1));
//     Config::SetDefault ("ns3::DcaTxop::MaxCw", UintegerValue (info.cwmax));


    NodeContainer c;
    c.Create (info.nos);

    // The below set of helpers will help us to put together the wifi NICs we want
    WifiHelper wifi;
    wifi.SetStandard (WIFI_PHY_STANDARD_80211b);

    YansWifiPhyHelper wifiPhy =  YansWifiPhyHelper::Default ();
    // ns-3 supports RadioTap and Prism tracing extensions for 802.11b
    wifiPhy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO);
    wifiPhy.Set ("RxNoiseFigure", DoubleValue (info.noiseFig));
    wifiPhy.SetErrorRateModel ("ns3::YansErrorRateModel");

    wifiPhy.Set ("TxGain", DoubleValue(0));
    wifiPhy.Set ("RxGain", DoubleValue(0));
    wifiPhy.Set ("TxPowerStart", DoubleValue(info.txPowerDbm));
    wifiPhy.Set ("TxPowerEnd", DoubleValue(info.txPowerDbm));
//     wifiPhy.Set ("EnergyDetectionThreshold", DoubleValue(-81.0));
//     wifiPhy.Set ("CcaMode1Threshold", DoubleValue(-91.0));
    wifiPhy.Set ("EnergyDetectionThreshold", DoubleValue(rxPowerDbm (info.distance, info.height, info.txPowerDbm, info.useTwoRay)));
    wifiPhy.Set ("CcaMode1Threshold", DoubleValue(rxPowerDbm (info.distance*info.factorCca, info.height, info.txPowerDbm, info.useTwoRay)));

    YansWifiChannelHelper wifiChannel ;
    wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");

    if(!info.useTwoRay){
    wifiChannel.AddPropagationLoss ("ns3::FriisPropagationLossModel",
     "Frequency", DoubleValue(freq));
    }else{
    wifiChannel.AddPropagationLoss ("ns3::TwoRayGroundPropagationLossModel",
     "Frequency", DoubleValue(freq));
    }

    /*/------------------------------Rician Fading-----------------------------//
    double m;
    m = (pow((info.k+1),2)) / (2*info.k+1); // "Wireless Communications" (Molisch)
    wifiChannel.AddPropagationLoss ("ns3::NakagamiPropagationLossModel",
"m0", DoubleValue (m),"m1", DoubleValue (m),"m2", DoubleValue (m));
    //------------------------------Rician Fading --------------------------/*/

    wifiPhy.SetChannel (wifiChannel.Create ());

    // Add a non-QoS upper mac, and disable rate control
    NqosWifiMacHelper wifiMac = NqosWifiMacHelper::Default ();
```

```
    wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager", "DataMode",StringValue(info.phyMode),
      "ControlMode",StringValue(info.phyMode));

    // Set it to adhoc mode
    wifiMac.SetType ("ns3::AdhocWifiMac");
    NetDeviceContainer devices = wifi.Install (wifiPhy, wifiMac, c);

    MobilityHelper mobility;
    Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator> ();
    for(int j=0;j<info.nos;j++) positionAlloc->Add (Vector (x[j],y[j],info.height)); //alocaÃğÃčo das posiÃğÃţes dos nÃşs
    mobility.SetPositionAllocator (positionAlloc);
    mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
    mobility.Install (c);

    InternetStackHelper internet;
    internet.Install (c);

    Ipv4AddressHelper ipv4;
    NS_LOG_INFO ("Assign IP Addresses.");
    ipv4.SetBase ("10.1.1.0", "255.255.255.0");
    Ipv4InterfaceContainer interface = ipv4.Assign (devices);

    PopulateArpCache ();

    //-----------------------Constant Rate application------------------------------------//
    double start;

    srand (RngSeedManager::GetRun ());
    int index[info.rotas];
    for (int i=0; i<info.rotas; i++) index[i] = i;
    random_shuffle (&index[0], &index[info.rotas]);

    char MeanValueOn[50], MeanValueOff[50];
    sprintf (MeanValueOn, "ns3::ExponentialRandomVariable[Mean=%.3f]", info.onTime);
    sprintf (MeanValueOff, "ns3::ExponentialRandomVariable[Mean=%.3f]", info.offTime);
    cout << MeanValueOn << endl;

    for (int j=0; j<info.rotas; j++)
    {
      int i = index[j];
//       cout << from[i] << " -> " << to[i] << endl;
      PacketSinkHelper sink("ns3::UdpSocketFactory",
      InetSocketAddress(interface.GetAddress(to[i]-1), 80));
      ApplicationContainer sinkApp = sink.Install(c.Get(to[i]-1));
      start = 0.0 + 0.05*i;
      sinkApp.Start(Seconds(start));
      sinkApp.Stop(Seconds(info.stop));

      OnOffHelper onOff ("ns3::UdpSocketFactory",
      InetSocketAddress(interface.GetAddress(from[i]-1), 80));
      onOff.SetConstantRate (DataRate("1000000"), info.packetSize);
      onOff.SetAttribute ("Remote", AddressValue(InetSocketAddress(interface.GetAddress(to[i]-1),80)));
      onOff.SetAttribute ("OnTime", StringValue (MeanValueOn));
      onOff.SetAttribute ("OffTime", StringValue (MeanValueOff));
      ApplicationContainer udpApp = onOff.Install(c.Get(from[i]-1));
      udpApp.Start(Seconds(start));
      udpApp.Stop(Seconds(info.stop));
    }
    //----------------------Constant Rate application-------------------------------------------/*/

    //--------------- Tracing----------------//
    if (info.tracing)
    {
//       wifiPhy.EnablePcap (info.outfile + prefix, devices);
      AsciiTraceHelper ascii;
      wifiPhy.EnableAsciiAll (ascii.CreateFileStream ("ascii-" + info.outfile + prefix + ".tr"));
    }
//     wifiPhy.EnablePcap (info.outfile + prefix, devices.Get (31));
    //---------------Tracing------------/*/

    //-------------------------------Flow Monitor----------------------/*/
    FlowMonitorHelper flowmon;
    Ptr<FlowMonitor> monitor = flowmon.InstallAll ();
    monitor->Start (Seconds (info.warmUp)); // start monitoring after network warm up
    monitor->Stop (Seconds (info.stop)); // stop monitoring

    Simulator::Stop (Seconds (info.stop+0.001));
    Simulator::Run ();
    monitor->CheckForLostPackets ();

    Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier> (flowmon.GetClassifier ());
```

```
    std::map<FlowId, FlowMonitor::FlowStats> stats = monitor->GetFlowStats ();

    for (std::map<FlowId, FlowMonitor::FlowStats>::const_iterator i = stats.begin (); i != stats.end (); ++i)
    {
      /* Throughput Average by Flow (bps) = sumRxBytesByFlow * 8 / (nFlows * time)
       * Throughput Quadratic Average by Flow (bps) = sumRxBytesQuadByFlow * 64 / (nFlows * time * time)
       * Net Aggregated Throughput Average by Node (bps) = sumRxBytesByFlow * 8 / (nodes * time)
       * Fairness = sumRxBytesByFlow^2 / (nFlows * sumRxBytesQuadByFlow)
       * Delay per Packet (seconds/packet) = sumDelayFlow / sumRxPktsByFlow
       * Lost Ratio (%) = 100 * sumLostPktsByFlow / sumTxPktsByFlow
       */
      data.nFlows++;
      data.sumRxBytesByFlow += i->second.rxBytes; // sum flows
      data.sumRxBytesQuadByFlow += i->second.rxBytes * i->second.rxBytes; // sum flowsÂš
      data.sumDelayFlow += i->second.delaySum.GetInteger (); // sum delays
      data.sumRxPktsByFlow += i->second.rxPackets; // sum rx pkts
      data.sumTxPktsByFlow += i->second.txPackets; // sum tx pkts
      data.sumLostPktsByFlow += i->second.lostPackets; // sum lost pkts
    }
    //---------------------------fim-Flow Monitor-----------------------/*/

    Simulator::Destroy ();

    ComputeResults ();

    return 0;
}

double
rxPowerDbm (double distance, double height, double txPowerDbm, bool useTwoRay)
{
    double lossPowerDbm;

    if (useTwoRay){
    double dCross = (4 * PI * height * height) / lambda;
    if (distance <= dCross){
      lossPowerDbm = 10 * log10( lambda*lambda / (16.0 * PI*PI * distance*distance));
    } else {
      lossPowerDbm = 10 * log10( (height*height*height*height) / (distance*distance*distance*distance) );
    }
    }
    else {
      lossPowerDbm = 10 * log10( lambda*lambda / (16.0 * PI*PI * distance*distance));
    }

return txPowerDbm + lossPowerDbm;
}

void
LerTopologia (char *topo, char *routes)
{
  in.open(topo); // counting number of lines

  if (!in){
    cerr << "Topology file not found" << endl;
    return;
  }else{
    while (in.get(ch)){
if (ch=='\n'){ info.nos++;} // number of lines is number of nodes
    }
  }
  in.close ();

  x = (float *)malloc(info.nos * sizeof(float));
  y = (float *)malloc(info.nos * sizeof(float));

  in.open(topo); // read coordinates
  if (!in){ cerr << "Topology file not found!" << endl; }
  else{
    while (in){
for(int i=0; i<info.nos; i++){ in >> x[i] >> y[i]; }
in.close ();
    }
  }
  in.close ();
  //-----------------------------------------------------------//
  in.open(routes); // count number of lines
  if (!in){
    cerr << "Routes file not found" << endl;
    return;
  }else{
```

```
    while (in.get(ch)){
if (ch=='\n'){ info.rotas++;} // number of lines is number of routes
    }
  }
  in.close ();

  from = (int *)malloc(info.rotas * sizeof(int));
  to = (int *)malloc(info.rotas * sizeof(int));

  in.open(routes); // read routes
  if (!in){ cerr << "Routes file not found!" << endl; }
  else{
    while (in){
for(int i=0; i<info.rotas; i++){ in >> from[i] >> to[i] >> ch; }
in.close ();
    }
  }
  in.close ();

//   for (int i=0; i<info.rotas; i++)
//   {
//     cout << from[i] << " -> " << to[i] << endl;
//   }
//     int opa;
//     cin >> opa;
}

Vector
GetPosition (Ptr<Node> node)
{
    Ptr<MobilityModel> mobility = node->GetObject<MobilityModel> ();
return mobility->GetPosition ();
}

void
ComputeResults (void)
{
  double deltaT = (info.stop - info.warmUp);
    // Throughput Average by Flow (bps)
  data.thrpAvgByFlow = (double) data.sumRxBytesByFlow * 8 / (data.nFlows * deltaT);
  // Throughput Quadratic Average by Flow (bpsÂš)
  data.thrpAvgQuadByFlow = (double) data.sumRxBytesQuadByFlow * 8*8 / (data.nFlows * deltaT*deltaT);
  // Throughput Variance by Flow (bpsÂš)
  data.thrpVarByFlow = data.thrpAvgQuadByFlow - data.thrpAvgByFlow * data.thrpAvgByFlow;
  // Network Aggregated Throughput Average by Node (bps)
  data.netThrpAvgByNode = (double) data.sumRxBytesByFlow * 8 / (info.nos * deltaT);
  // Fairness Jain's Index
  data.fairness = (double) data.sumRxBytesByFlow * data.sumRxBytesByFlow / (data.nFlows * data.sumRxBytesQuadByFlow);
  // Delay Mean by Packet (nanoseconds)
  data.delayByPkt = (double) data.sumDelayFlow / data.sumRxPktsByFlow;
  // Lost Ratio (%)
  data.lostRatio = (double) 100 * data.sumLostPktsByFlow / data.sumTxPktsByFlow;

  time_t now = time(0);
  char* dt = ctime(&now);
  cout << "=====================================================================" << endl
       << dt
       << "=====================================================================" << endl
       << "Simulation parameters:" << endl
       << "Run:      \t" << info.run << endl
       << "Time:     \t" << info.stop << " s" << endl
       << "Warm up:  \t" << info.warmUp << " s" << endl
       << "Topology: \t" << info.top << endl
       << "Nodes:    \t" << info.nos << endl
       << "Polling:  \t" << info.pollingMode << endl
       << endl
       << "=====================================================================" << endl
       << "Simulation results:" << endl
       << "Throughput Average by Flow (kbps):\t" << data.thrpAvgByFlow / 1024.0 << endl
       << "Throughput Deviation by Flow (kbps):\t" << sqrt (data.thrpVarByFlow) / 1024.0 << endl
       << "Network Aggregated Throughput Average by Node (kbps):\t" << data.netThrpAvgByNode / 1024.0 << endl
       << "Fairness Jain's Index:\t" << data.fairness << endl
       << "Delay Mean by Packet (seconds):\t" << data.delayByPkt / 1e9 << endl
       << "Packet Lost Ratio (%):\t" << data.lostRatio << endl << endl << endl;


  if (info.save)
  {
//     ofstream saida;
//     char filename[100];
//     strcpy (filename, info.outfile.c_str());
```

```
//     saida.open (filename, ios::app);
//     cout << "Output file: " << info.outfile << endl;
//     saida << "=====================================================================" << endl
//    << dt
//    << "=====================================================================" << endl
//    << "Simulation parameters:" << endl
//    << "Run:       \t" << info.run << endl
//    << "Time:      \t" << info.stop << " s" << endl
//    << "Warm up:   \t" << info.warmUp << " s" << endl
//    << "Topology: \t" << info.top << endl
//    << "Nodes:    \t" << info.nos << endl
//    << "Polling:  \t" << info.pollingMode << endl
//    << endl
//    << "Physical layer parameters:" << endl
//    << "Phy mode:       \t" << info.phyMode << endl
//    << "Rician fading: \t" << info.k << endl
//    << "Tx range:       \t" << info.distance << " m" << endl
//    << "Cca range:      \t" << info.distance*info.factorCca << " m" << endl
//    << "Tx power:       \t" << info.txPowerDbm << " dBm" << endl
//    << endl
//    << "Mac layer parameters:" << endl
//    << "Max ssrc: \t" << info.ssrc << endl
//    << endl
//    << "Application layer parameters:" << endl
//    << "On time: \t" << info.onTime << " s" << endl
//    << "Off time: \t" << info.offTime << " s" << endl
//    << "Packet size: \t" << info.packetSize << " bytes" << endl
//    << endl
//    << "=====================================================================" << endl
//    << "Simulation results:" << endl
//    << "Throughput Average by Flow (kbps): \t\t\t" << data.thrpAvgByFlow / 1024.0 << endl
//    << "Throughput Deviation by Flow (kbps): \t\t\t" << sqrt (data.thrpVarByFlow) / 1024.0 << endl
//    << "Network Aggregated Throughput Average by Node (kbps): \t" << data.netThrpAvgByNode / 1024.0 << endl
//    << "Fairness Jain's Index: \t\t\t\t\t" << data.fairness << endl
//    << "Delay Mean by Packet (miliseconds): \t\t\t" << data.delayByPkt / 1e6 << endl
//    << "Packet Lost Ratio (%): \t\t\t\t\t" << data.lostRatio << endl << endl << endl;
//     saida.close ();

    ofstream sheet;
    char filename2[100];
    strcpy (filename2 ,info.outfile.c_str());
    strcat (filename2, "_sh");
    sheet.open (filename2, ios::app);
    cout << "Output sheet file: " << filename2 << endl;
//     if (sheet.tellp () == sheet.eof ()) sheet << "Top"<<"\t"<< setw (9)<<"FlowThrp"<<"\t"<< setw (9)<<"NetThrp"<<"\t"<< setw (9)<<"Fairness"<<"\t"<< setw (9)<<"Dela
    sheet << info.pollingMode << "\t" << setw (9) << info.run << "\t" << setw (9) << info.top << "\t" << setw (9) << info.neighborsThresh << "\t" << setw (9) << info.s
    sheet.close ();

  }
}

void
PopulateArpCache (void)
{
  Ptr<ArpCache> arp = CreateObject<ArpCache> ();
  arp->SetAliveTimeout (Seconds(3600 * 24 * 365));
  for (NodeList::Iterator i = NodeList::Begin(); i != NodeList::End(); ++i)
  {
    Ptr<Ipv4L3Protocol> ip = (*i)->GetObject<Ipv4L3Protocol> ();
    NS_ASSERT(ip !=0);
    ObjectVectorValue interfaces;
    ip->GetAttribute("InterfaceList", interfaces);
    for(ObjectVectorValue::Iterator j = interfaces.Begin(); j !=
interfaces.End (); j ++)
    {
      Ptr<Ipv4Interface> ipIface = (j->second)->GetObject<Ipv4Interface> ();
      NS_ASSERT(ipIface != 0);
      Ptr<NetDevice> device = ipIface->GetDevice();
      NS_ASSERT(device != 0);
      Mac48Address addr = Mac48Address::ConvertFrom(device->GetAddress ());
      for(uint32_t k = 0; k < ipIface->GetNAddresses (); k ++)
      {
        Ipv4Address ipAddr = ipIface->GetAddress (k).GetLocal();
        if(ipAddr == Ipv4Address::GetLoopback())
          continue;
        ArpCache::Entry * entry = arp->Add(ipAddr);
        entry->MarkWaitReply(0);
        entry->MarkAlive(addr);
      }
    }
  }
```

```
  for (NodeList::Iterator i = NodeList::Begin(); i != NodeList::End(); ++i)
  {
    Ptr<Ipv4L3Protocol> ip = (*i)->GetObject<Ipv4L3Protocol> ();
    NS_ASSERT(ip !=0);
    ObjectVectorValue interfaces;
    ip->GetAttribute("InterfaceList", interfaces);
    for(ObjectVectorValue::Iterator j = interfaces.Begin(); j !=
interfaces.End (); j ++)
    {
      Ptr<Ipv4Interface> ipIface = (j->second)->GetObject<Ipv4Interface> ();
      ipIface->SetAttribute("ArpCache", PointerValue(arp));
    }
  }
}
```

IV. NS-3 CHANGELOG

IV.1 DcaTxop

IV.1.1 dca-txop.h

```
4d3
<  * Copyright (c) 2013,2014 NERds UnB
20,21d18
<  * Modifier: Fadhil Firyaguna <firyaguna@ieee.org>
<  * Modifier: Mateus Marcuzzo <mateusmarcuzzo@ieee.org>
202,205d198
<   /**
<    *
<    */
<   void DequeueByAddress (Mac48Address addr); // RIMA
209,226d201
<    * Event handler when a DATA is received.
<    *
<    * \param snr
<    * \param txMode
<    */
<   void GotData (double snr, WifiMode txMode); // RIMA
<   /**
<    * Event handler when a DATA timeout has occurred.
<    */
<   void MissedData (void); // RIMA
<   /**
<    * Event handler when a NTS is received.
<    *
<    * \param snr
<    * \param txMode
<    */
<   void GotNts (void); // RIMA
<   /**
280,283d254
<    *
<    */
<   bool NeedRtrRetransmission (void); // RIMA
<   /**
359,360c330
<   double m_ptxBroadcast; // RIMA
<   uint32_t m_initMinCw; // RIMA
---
>   uint32_t m_initMinCw;
```

IV.1.2 dca-txop.cc

```
4d3
<  * Copyright (c) 2013,2014 NERds UnB
20,21d18
<  * Modifier: Fadhil Firyaguna <firyaguna@ieee.org>
<  * Modifier: Mateus Marcuzzo <mateusmarcuzzo@ieee.org>
32,34d28
< #include "ns3/double.h"
< #include "ns3/random-variable.h"
<
96,112d89
<   virtual void DequeueByAddress (Mac48Address addr) // RIMA
<   {
<     return m_txop->DequeueByAddress (addr);
<   }
<   virtual void GotData (double snr, WifiMode txMode) // RIMA
<   {
<     m_txop->GotData (snr, txMode);
<   }
<   virtual void MissedData (void) // RIMA
<   {
<     m_txop->MissedData ();
<   }
<   virtual void GotNts (void) // RIMA
```

```
<   {
<     m_txop->GotNts ();
<   }
<
158,161d134
<     .AddAttribute ("ProbTxBroadcast", "Probability of transmit broadcast", // RIMA
<     DoubleValue (0.8),
<     MakeDoubleAccessor (&DcaTxop::m_ptxBroadcast),
<     MakeDoubleChecker<double> ())
166c139
<                    MakeUintegerChecker<uint32_t> ())
---
>                    MakeUintegerChecker<uint32_t> ())
221d193
<   m_low->SetTransmissionListener (m_transmissionListener); // RIMA
296,299d267
<   /* RIMA
<    * Ao receber um pacote da camada superior, apenas colocar na fila.
<    * NÃčo Ãľ necessÃąrio pedir acesso ao meio.
<    */
306c274
< //   StartAccessIfNeeded (); // RIMA
---
>   StartAccessIfNeeded ();
320,323d287
<   /* RIMA
<    * O acesso Ãľ reiniciado apÃşs uma tentativa bem sucedida
<    * (GotData) ou nÃčo (MissedData) de consulta.
<    */
325c289
<   if (/*(m_currentPacket != 0
---
>   if ((m_currentPacket != 0
327c291
<       && */!m_dcf->IsAccessRequested ())
---
>       && !m_dcf->IsAccessRequested ())
366c330
<   NS_LOG_DEBUG ("cwmin=" << m_dcf->GetCwMin () << " cwmax=" << m_dcf->GetCwMax ());
---
>   NS_LOG_DEBUG ("cwmin=" << m_dcf->GetCwMin ());
368,372d331
<   // request access for polling
<   /*
<    * A consulta por dados Ãľ iniciada jÃą ao ligar.
<    */
<   m_manager->RequestAccess (m_dcf); // RIMA
375,383d333
< bool // RIMA
< DcaTxop::NeedRtrRetransmission (void)
< {
<   /*
<    * Retorna TRUE se o nÃžmero de tentativas de consulta a um dado endereÃğo
<    * Ãľ menor que o nÃžmero mÃąximo permitido.
<    */
<   return m_stationManager->NeedRtrRetransmission (Low ()->GetCurrentPollAddr ());
< }
466,506d415
< void // RIMA
< DcaTxop::DequeueByAddress (Mac48Address addr)
< {
<   /*
<    * Procura na fila um pacote destinado ao endereÃğo 'addr'
<    * e retorna o primeiro que Ãľ encontrado.
<    */
<   if (m_queue->IsEmpty ())
<   {
<     NS_LOG_DEBUG ("queue empty, packet for no one");
<   }
<   else
<   {
<     NS_LOG_DEBUG ("search packet for " << addr);
<     Ptr<const Packet> packet = m_queue->PeekByAddress (&m_currentHdr, addr);
<     if (packet == 0)
<     {
<       NS_LOG_DEBUG ("no packet for " << addr << " found, sorry");
<     }
<     else
<     {
<       NS_LOG_DEBUG ("found it");
<       m_currentPacket = m_queue->DequeueByAddress (&m_currentHdr, addr);
```

```
<  NS_ASSERT (m_currentPacket != 0);
<  uint16_t sequence = m_txMiddle->GetNextSequenceNumberfor (&m_currentHdr);
<  m_currentHdr.SetSequenceNumber (sequence);
<  m_currentHdr.SetFragmentNumber (0);
<  m_currentHdr.SetNoMoreFragments ();
<  m_currentHdr.SetNoRetry ();
<  m_fragmentNumber = 0;
<  NS_LOG_DEBUG ("dequeued size=" << m_currentPacket->GetSize () <<
<        ", to=" << m_currentHdr.GetAddr1 () <<
<        ", seq=" << m_currentHdr.GetSequenceControl ());
<       Low ()->SetCurrentPacket (m_currentPacket, &m_currentHdr);
<     }
<   }
<
<   // Dec 20th 2013
< //   m_queue->GetQueueStatus (Low ()->GetAddress ());
< }
<
513,514c422
<
< void //---------- Feb 5th 2014
---
> void
517,521c425,444
<   /* Channel access granted
<    * With probability 0.5, transmit broadcast if there is any
<    * Else, transmit polling packet
<    */
<   NS_LOG_DEBUG ("dca-txop access granted");
---
>   NS_LOG_FUNCTION (this);
>   if (m_currentPacket == 0)
>     {
>       if (m_queue->IsEmpty ())
>         {
>           NS_LOG_DEBUG ("queue empty");
>           return;
>         }
>       m_currentPacket = m_queue->Dequeue (&m_currentHdr);
>       NS_ASSERT (m_currentPacket != 0);
>       uint16_t sequence = m_txMiddle->GetNextSequenceNumberfor (&m_currentHdr);
>       m_currentHdr.SetSequenceNumber (sequence);
>       m_currentHdr.SetFragmentNumber (0);
>       m_currentHdr.SetNoMoreFragments ();
>       m_currentHdr.SetNoRetry ();
>       m_fragmentNumber = 0;
>       NS_LOG_DEBUG ("dequeued size=" << m_currentPacket->GetSize () <<
>                     ", to=" << m_currentHdr.GetAddr1 () <<
>                     ", seq=" << m_currentHdr.GetSequenceControl ());
>     }
524,557c447,457
<
<   NS_LOG_FUNCTION (this);
<
<   UniformVariable p (0,1);
<   bool p_txBroadcast = p.GetValue () < m_ptxBroadcast;
<
<   WifiMacHeader broadcastHdr;
<   Ptr<const Packet> broadcastPacket = m_queue->PeekByAddress (&broadcastHdr, Mac48Address::GetBroadcast ());
<   bool gotBroadcast = (broadcastPacket != 0);
<
<   if (gotBroadcast and p_txBroadcast)
<   {
<     m_currentPacket = m_queue->DequeueByAddress (&m_currentHdr, Mac48Address::GetBroadcast ());
<     NS_ASSERT (m_currentPacket != 0);
<     uint16_t sequence = m_txMiddle->GetNextSequenceNumberfor (&m_currentHdr);
<     m_currentHdr.SetSequenceNumber (sequence);
<     m_currentHdr.SetFragmentNumber (0);
<     m_currentHdr.SetNoMoreFragments ();
<     m_currentHdr.SetNoRetry ();
<     m_fragmentNumber = 0;
<     NS_LOG_DEBUG ("dequeued size=" << m_currentPacket->GetSize () <<
<    ", to=" << m_currentHdr.GetAddr1 () <<
<    ", seq=" << m_currentHdr.GetSequenceControl ());
<
<     params.DisableRtr (); // desabilitar RTR
<     params.DisableAck ();
<     params.DisableNextData ();
<     Low ()->StartTransmission (m_currentPacket, // pacote broadcast
<        &m_currentHdr,
<        params, // carrega a informaĂğĂčo de que o RTR estĂą desabilitado
```

```
<        m_transmissionListener);
<     NS_LOG_DEBUG ("tx broadcast");
<     return;
<   }
---
>   if (m_currentHdr.GetAddr1 ().IsGroup ())
>     {
>       params.DisableRts ();
>       params.DisableAck ();
>       params.DisableNextData ();
>       Low ()->StartTransmission (m_currentPacket,
>                                  &m_currentHdr,
>                                  params,
>                                  m_transmissionListener);
>       NS_LOG_DEBUG ("tx broadcast");
>     }
559,565c459,503
<   {
<     NS_LOG_DEBUG ("dca-txop polling");
<     params.EnableRtr (); // habilitar RTR
<     Low ()->StartTransmission (m_currentPacket, // objeto vazio
<  &m_currentHdr, // objeto vazio
<  params, m_transmissionListener);
<   }
---
>     {
>       params.EnableAck ();
>
>       if (NeedFragmentation ())
>         {
>           WifiMacHeader hdr;
>           Ptr<Packet> fragment = GetFragmentPacket (&hdr);
>           if (NeedRts (fragment, &hdr))
>             {
>               params.EnableRts ();
>             }
>           else
>             {
>               params.DisableRts ();
>             }
>           if (IsLastFragment ())
>             {
>               NS_LOG_DEBUG ("fragmenting last fragment size=" << fragment->GetSize ());
>               params.DisableNextData ();
>             }
>           else
>             {
>               NS_LOG_DEBUG ("fragmenting size=" << fragment->GetSize ());
>               params.EnableNextData (GetNextFragmentSize ());
>             }
>           Low ()->StartTransmission (fragment, &hdr, params,
>                                      m_transmissionListener);
>         }
>       else
>         {
>           if (NeedRts (m_currentPacket, &m_currentHdr))
>             {
>               params.EnableRts ();
>               NS_LOG_DEBUG ("tx unicast rts");
>             }
>           else
>             {
>               params.DisableRts ();
>               NS_LOG_DEBUG ("tx unicast");
>             }
>           params.DisableNextData ();
>           Low ()->StartTransmission (m_currentPacket, &m_currentHdr,
>                                      params, m_transmissionListener);
>         }
>     }
578a517
>   NS_LOG_DEBUG ("cwmin=" << m_dcf->GetCwMin ());
591,656d529
< void // RIMA
< DcaTxop::GotData (double snr, WifiMode txMode)
< {
<   NS_LOG_FUNCTION (this << snr << txMode);
<   NS_LOG_DEBUG ("got data");
<
<   /*
<    * ApÃşs o recebimento do DATA, conclui-se que o handshake
```

```
<     * foi bem sucedido, reseta a janela de contenÃğÃčo e
<     * reinicia o backoff.
<     */
<   m_dcf->ResetCw (); // A atualizaÃğÃčo da CW Ãľ feita pelo consultor
<   NS_LOG_DEBUG ("cwmin=" << m_dcf->GetCwMin () << " cwmax=" << m_dcf->GetCwMax ());
<   m_dcf->StartBackoffNow (m_rng->GetNext (0, m_dcf->GetCw ()));
<   RestartAccessIfNeeded ();
< }
< void // RIMA
< DcaTxop::MissedData (void)
< {
<   NS_LOG_FUNCTION (this);
<   NS_LOG_DEBUG ("missed data from " << Low ()->GetCurrentPollAddr ());
<   if (!NeedRtrRetransmission ())
<     /*
<      * Nesta parte, o nÃžmero de tentativas de envio de RTR
<      * chegou ao mÃąximo. A consulta para um determinado nÃş
<      * Ãľ abortada, a janela de contenÃğÃčo Ãľ reiniciada para
<      * consultar um novo vizinho.
<      */
<     {
<       NS_LOG_DEBUG ("Data Fail");
<       m_stationManager->ReportFinalRtrFailed (Low ()->GetCurrentPollAddr ());
< //       if (!m_txFailedCallback.IsNull ())
< //         {
< //           m_txFailedCallback (m_currentHdr);
< //         }
<       // to reset the dcf.
< //       m_currentPacket = 0;
<       m_dcf->ResetCw (); // A atualizaÃğÃčo da CW Ãľ feita pelo consultor
<     }
<   else
<     {
<       /*
<        * Nesta parte, continuam as tentativas de consulta.
<        * A janela de contenÃğÃčo Ãľ aumentada devido o handshake
<        * mal sucedido e reinicia o backoff.
<        */
<       m_dcf->UpdateFailedCw (); // A atualizaÃğÃčo da CW Ãľ feita pelo consultor
<     }
<   NS_LOG_DEBUG ("cwmin=" << m_dcf->GetCwMin () << " cwmax=" << m_dcf->GetCwMax ());
<   m_dcf->StartBackoffNow (m_rng->GetNext (0, m_dcf->GetCw ()));
<   RestartAccessIfNeeded ();
< }
<
< void // RIMA
< DcaTxop::GotNts (void)
< {
<   NS_LOG_FUNCTION (this);
<   NS_LOG_DEBUG ("got nts");
<   m_stationManager->ReportFinalRtrFailed (Low ()->GetCurrentPollAddr ());
< //   m_dcf->ResetCw ();
<   m_dcf->UpdateFailedCw ();
<   NS_LOG_DEBUG ("cwmin=" << m_dcf->GetCwMin () << " cwmax=" << m_dcf->GetCwMax ());
<   m_dcf->StartBackoffNow (m_rng->GetNext (0, m_dcf->GetCw ()));
<   RestartAccessIfNeeded ();
< }
<
683a557
>     NS_LOG_DEBUG ("cwmin=" << m_dcf->GetCwMin ());
687,688c561
<
< void // RIMA Apr 11th 2013
---
> void
704,714d576
<
<       /* Eu estava esperando meu backoff terminar quando de repente
<        * recebo um RTR e tenho DATA para mandar. Interrompo meu
<        * temporizador, mando o DATA e recebo o ACK. Retomo o tempo
<        * do backoff de onde tinha parado e continuo a esperar.
<        *
<        * Portanto, nÃčo preciso resetar a Cw, nem gerar um novo
<        * tempo de backoff para recomeÃğar o recÃžo.
<        *
<        * Com o ACK recebido, eu posso descartar o pacote jÃą enviado.
<        */
716,718c578,581
< //       m_dcf->ResetCw (); // O transmissor nÃčo necessita atualizar Cw. Apenas o consultor.
< //       m_dcf->StartBackoffNow (m_rng->GetNext (0, m_dcf->GetCw ()));
< //       RestartAccessIfNeeded ();
```

```
---
>       m_dcf->ResetCw ();
>       NS_LOG_DEBUG ("cwmin=" << m_dcf->GetCwMin ());
>       m_dcf->StartBackoffNow (m_rng->GetNext (0, m_dcf->GetCw ()));
>       RestartAccessIfNeeded ();
732,735d594
<       /*
<        * Nesta parte, o limite de retransmissÃţes para um dado endereÃğo foi alcanÃğado.
<        * Descartar o pacote.
<        */
741a601
>       // to reset the dcf.
743c603
< //       m_dcf->ResetCw (); // O transmissor nÃčo necessita atualizar Cw. Apenas o consultor.
---
>       m_dcf->ResetCw ();
747,750d606
<       /*
<        * Nesta parte, o nÃžmero de retransmissÃţes para um dado endereÃğo Ãľ menor que o limite.
<        * Devolver o pacote que nÃčo foi reconhecido para a comeÃğo da fila.
<        */
753,754c609
<       m_queue->PushFront (m_currentPacket, m_currentHdr);
< //       m_dcf->UpdateFailedCw (); // O transmissor nÃčo necessita atualizar Cw. Apenas o consultor.
---
>       m_dcf->UpdateFailedCw ();
756,767c611,613
<   /*
<     * Eu estava esperando meu backoff terminar quando de repente
<     * recebo um RTR e tenho DATA para mandar. Interrompo meu
<     * temporizador, mando o DATA e mas nÃčo recebo o ACK. EntÃčo
<     * eu deixo o pacote que mandei para ele ser consultado de novo
<     * e retomo o tempo do backoff de onde tinha parado.
<     *
<     * Portanto, nÃčo preciso resetar a Cw, nem gerar um novo
<     * tempo de backoff para recomeÃğar o recÃžo.
<     */
< //   m_dcf->StartBackoffNow (m_rng->GetNext (0, m_dcf->GetCw ()));
< //   RestartAccessIfNeeded ();
---
>     NS_LOG_DEBUG ("cwmin=" << m_dcf->GetCwMin ());
>   m_dcf->StartBackoffNow (m_rng->GetNext (0, m_dcf->GetCw ()));
>   RestartAccessIfNeeded ();
825c671
< void // RIMA
---
> void
831c677,678
< //   m_dcf->ResetCw ();
---
>   m_dcf->ResetCw ();
>   NS_LOG_DEBUG ("cwmin=" << m_dcf->GetCwMin ());
833,834c680
< //   StartAccessIfNeeded ();
<   RestartAccessIfNeeded ();
---
>   StartAccessIfNeeded ();
```

IV.2 DcfManager

IV.2.1 dcf-manager.h

```
4d3
<  * Copyright (c) 2013,2014 NERds UnB
20,21d18
<  * Modifier: Fadhil Firyaguna <firyaguna@ieee.org>
<  * Modifier: Mateus Marcuzzo <mateusmarcuzzo@ieee.org>
372,381d368
<   /**
<    * Notify that DATA timer has started for the given duration.
<    *
<    * \param duration
<    */
<   void NotifyDataTimeoutStartNow (Time duration); // RIMA
<   /**
<    * Notify that DATA timer has resetted.
<    */
```

```
<   void NotifyDataTimeoutResetNow (); // RIMA
490d476
<   Time m_lastDataTimeoutEnd; // RIMA
```

IV.2.2 dcf-manager.cc

```
4d3
<  * Copyright (c) 2013,2014 NERds UnB
20,21d18
<  * Modifier: Fadhil Firyaguna <firyaguna@ieee.org>
<  * Modifier: Mateus Marcuzzo <mateusmarcuzzo@ieee.org>
211,218d207
<   virtual void DataTimeoutStart (Time duration) // RIMA
<   {
<     m_dcf->NotifyDataTimeoutStartNow (duration);
<   }
<   virtual void DataTimeoutReset () // RIMA
<   {
<     m_dcf->NotifyDataTimeoutResetNow ();
<   }
276d264
<     m_lastDataTimeoutEnd (MicroSeconds (0)), // RIMA
544d531
<   Time dataTimeoutAccessStart = m_lastDataTimeoutEnd + m_sifs; // RIMA
551,552c538
<                                           MostRecent(ctsTimeoutAccessStart,
<  dataTimeoutAccessStart),
---
>                                           ctsTimeoutAccessStart,
732,735d717
<   if (m_lastDataTimeoutEnd > now) // RIMA
<     {
<       m_lastDataTimeoutEnd = now;
<     }
821,831d802
<   DoRestartAccessTimeoutIfNeeded ();
< }
< void // RIMA
< DcfManager::NotifyDataTimeoutStartNow (Time duration)
< {
<   m_lastDataTimeoutEnd = Simulator::Now () + duration;
< }
< void // RIMA
< DcfManager::NotifyDataTimeoutResetNow ()
< {
<   m_lastDataTimeoutEnd = Simulator::Now ();
```

IV.3 EdcaTxopN

IV.3.1 edca-txop-n.h

IV.3.2 edca-txop-n.cc

```
5d4
<  * Copyright (c) 2013,2014 NERds UnB
22d20
<  * Modifier: Fadhil Firyaguna <firyaguna@ieee.org>
83,95d80
<
<   virtual void DequeueByAddress (Mac48Address addr) // RIMA
<   {
<   }
<   virtual void GotData (double snr, WifiMode txMode) // RIMA
<   {
<   }
<   virtual void MissedData (void) // RIMA
<   {
<   }
<   virtual void GotNts (void) // RIMA
<   {
<   }
```

IV.4 MacLow

IV.4.1 mac-low.h

```
5d4
<  * Copyright (c) 2013,2014 NERds UnB
22,23d20
<  * Modifier: Fadhil Firyaguna <firyaguna@ieee.org>
<  * Modifier: Mateus Marcuzzo <mateusmarcuzzo@ieee.org>
66,86d62
<    * \param addr the destination address of the packet
<    */
<   virtual void DequeueByAddress (Mac48Address addr) = 0; // RIMA
<   /**
<    * \param snr the snr of the cts
<    * \param txMode the txMode of the cts
<    *
<    * ns3::MacLow received an expected DATA within
<    * DataTimeout.
<    */
<   virtual void GotData (double snr, WifiMode txMode) = 0; // RIMA
<   /**
<    * ns3::MacLow did not receive an expected DATA
<    * within DataTimeout
<    */
<   virtual void MissedData (void) = 0; // RIMA
<   /**
<    * ns3::MacLow received a NTS within DataTimeout
<    */
<   virtual void GotNts (void) = 0; // RIMA
<   /**
208,217d183
<   /**
<    * Notify that DATA timeout has started for a given duration.
<    *
<    * \param duration duration of DATA timeout
<    */
<   virtual void DataTimeoutStart (Time duration) = 0; // RIMA
<   /**
<    * Notify that DATA timeout has resetted.
<    */
<   virtual void DataTimeoutReset () = 0; // RIMA
258,264d223
<    * Send a RTR, and wait DATATimeout for a DATA. If we get a
<    * DATA on time, call MacLowTransmissionListener::GotData
<    * and send ACK. Otherwise, call MacLowTransmissionListener::MissedData
<    * and do not send ACK.
<    */
<   void EnableRtr (void); // RIMA
<   /**
331,334d289
<    * Do not send rtr before receiving data.
<    */
<   void DisableRtr (void); // RIMA
<   /**
401,405d355
<    /**
<    * \returns true if RTR should be sent before
<    *          receiving data, false otherwise
<    */
<   bool MustSendRtr (void) const; // RIMA
444d393
<   bool m_sendRtr; // RIMA
511,516d459
<    * Set DATA timeout of this MacLow.
<    *
<    * \param ctsTimeout DATA timeout of this MacLow
<    */
<   void SetDataTimeout (Time dataTimeout); // RIMA
<   /**
593,598d535
<    * Return DATA timeout of this MacLow.
<    *
<    * \return DATA timeout
<    */
<   Time GetDataTimeout (void) const; // RIMA
<   /**
641,661d577
<
```

```
<   void SetTransmissionListener (MacLowTransmissionListener *listener); // RIMA
<   void SetCurrentPollAddr (Mac48Address ad); // RIMA
<   Mac48Address GetCurrentPollAddr (void) const; // RIMA
<   Mac48Address GetPollingAddress (void); // RIMA
<
<   void UpdatePollingTable (Mac48Address addr, int txok); // RIMA
<
<   void CheckRouteRequest (Ptr<Packet> packet, Mac48Address addr); // RIMA AODV
<   void CheckRouteReply (Ptr<Packet> packet, Mac48Address addr); // RIMA AODV
<   void CheckRouteError (Ptr<Packet> packet, Mac48Address addr); // RIMA AODV
<   void ResetRoute (Mac48Address addr); // RIMA AODV
<
<   /**
<    * \param packet packet requested by polling
<    * \param hdr 802.11 header for packet
<    *
<    * This method copies the packet from DcaTxop to MacLow, to be transmitted
<    * to the poller node.
<    */
<   void SetCurrentPacket (Ptr<const Packet> packet, const WifiMacHeader* hdr); // RIMA
795,800d710
<    * Return the total RTR size (including FCS trailer).
<    *
<    * \return the total RTR size
<    */
<   uint32_t GetRtrSize (void) const; // RIMA
<   /**
832,840d741
<    * Return a TXVECTOR for the RTR frame given the destination.
<    * The function consults WifiRemoteStationManager, which controls the rate
<    * to different destinations.
<    *
<    * \param from the MAC address of the RTR sender
<    * \return TXVECTOR for the RTS of the given packet
<    */
<   WifiTxVector GetRtrTxVector (Mac48Address from) const; // RIMA
<   /**
896,905d796
<    * Return a TXVECTOR for the NTS frame given the destination and the mode of the RTR
<    * used by the sender.
<    * The function consults WifiRemoteStationManager, which controls the rate
<    * to different destinations.
<    *
<    * \param to the MAC address of the NTS receiver
<    * \return TXVECTOR for the NTS
<    */
<   WifiTxVector GetNtsTxVectorForRtr (Mac48Address to) const; // RIMA
<   /**
1009,1021d899
<    * DATA timer should be started for the given
<    * duration.
<    *
<    * \param duration
<    */
<   void NotifyDataTimeoutStartNow (Time duration); // RIMA
<   /**
<    * Notify DcfManager (via DcfListener) that
<    * DATA timer should be resetted.
<    */
<   void NotifyDataTimeoutResetNow (); // RIMA
<   /**
<    * Notify DcfManager (via DcfListener) that
1064,1067d941
<    * Event handler when DATA timeout occurs.
<    */
<   void DataTimeout (void); // RIMA
<   /**
1076,1084d949
<    * Send NTS after receiving RTR.
<    *
<    * \param source
<    * \param duration
<    * \param txMode
<    * \param rtsSnr
<    */
<   void SendNtsAfterRtr (Mac48Address source, Time duration, WifiMode txMode, double rtsSnr); // RIMA
<   /**
1103,1110d967
<    * Send DATA after receiving RTR.
<    *
<    * \param source
```

```
<    * \param duration
<    * \param txMode
<    */
<   void SendDataAfterRtr (Mac48Address source, Time duration, WifiMode txMode, double rtsSnr); // RIMA
<   /**
1129,1132d985
<    * Send RTS to begin RTR-DATA-ACK transaction.
<    */
<   void SendRtrForPacket (void); // RIMA
<   /**
1236d1088
<   EventId m_dataTimeoutEvent; // RIMA //!< DATA timeout event
1240d1091
<   EventId m_sendNtsEvent; // RIMA //!< Event to send NTS
1250d1100
<   Mac48Address m_currentPollAddr; // RIMA   //!< Address to poll
1256d1105
<   Time m_dataTimeout; // RIMA     //!< DATA timeout
```

IV.4.2 mac-low.cc

```
5d4
<  * Copyright (c) 2013,2014 NERds UnB
22,23d20
<  * Modifier: Fadhil Firyaguna <firyaguna@ieee.org>
<  * Modifier: Mateus Marcuzzo <mateusmarcuzzo@ieee.org>
41,43d37
< #include "ns3/ipv4-header.h"
< #include "ns3/aodv-packet.h"
<
88,97d81
< void // RIMA
< MacLowTransmissionParameters::EnableRtr (void)
< {
<   m_sendRtr = true;
< }
< void // RIMA
< MacLowTransmissionParameters::DisableRtr (void)
< {
<   m_sendRtr = false;
< }
198,202d181
< bool // RIMA
< MacLowTransmissionParameters::MustSendRtr (void) const
< {
<   return m_sendRtr;
< }
316d294
<     m_dataTimeoutEvent (), // RIMA
416,420d393
<   if (m_dataTimeoutEvent.IsRunning ()) // RIMA
<     {
<       m_dataTimeoutEvent.Cancel ();
<       oneRunning = true;
<     }
482,486d454
< void // RIMA
< MacLow::SetDataTimeout (Time dataTimeout)
< {
<   m_dataTimeout = dataTimeout;
< }
552,556d519
< Time // RIMA
< MacLow::GetDataTimeout (void) const
< {
<   return m_dataTimeout;
< }
593,724d555
< void // RIMA
< MacLow::SetTransmissionListener (MacLowTransmissionListener *listener)
< {
<   m_listener = listener;
< }
<
< void // RIMA
< MacLow::SetCurrentPollAddr (Mac48Address addr)
< {
<   m_currentPollAddr = addr;
< }
```

```
<
< Mac48Address // RIMA
< MacLow::GetCurrentPollAddr (void) const
< {
<   return m_currentPollAddr;
< }
< void // RIMA
< MacLow::UpdatePollingTable (Mac48Address addr, int txok)
< {
<   m_stationManager->UpdateTxProbability (addr, txok);
< }
<
< Mac48Address // RIMA
< MacLow::GetPollingAddress (void)
< {
<   if ((m_stationManager->NeedRtrRetransmission (m_currentPollAddr)
<        and m_currentPollAddr != Mac48Address ("00:00:00:00:00:00"))
<       or m_stationManager->HasMoreData (m_currentPollAddr))
<     /*
<      * Insistir a consulta ao mesmo endereÃğo atÃľ o nÃžmero mÃąximo de tentativas.
<      *
<      * ou
<      * Se Ãžltimo transmissor tiver mais dados, insistir tambÃľm.
<      */
<   {
<     return m_currentPollAddr;
<   }
<   else
<   {
<     return m_stationManager->NextPollingAddress (m_currentPollAddr);
<   /*/------------------- TEST -----------------------------//
<     uint8_t addr[6];
<     m_self.CopyTo (addr);
<     Mac48Address nextPoll;
<     addr[5] = addr[5] - 1;
<     if (addr[5] == 0) addr[5] = 9;
<     nextPoll.CopyFrom (addr);
<     return nextPoll;
<   //------------------- TEST -----------------------------/*/
<   }
< }
<
< void // RIMA
< MacLow::SetCurrentPacket (Ptr<const Packet> packet, const WifiMacHeader* hdr)
< {
<   if (packet != 0)
<   {
<     m_currentPacket = packet->Copy ();
<     m_currentHdr = *hdr;
<   }
< }
<
< void // RIMA AODV
< MacLow::CheckRouteRequest (Ptr<Packet> packet, Mac48Address addr)
< {
<     /* RIMA AODV CROSS LAYER - Apr 1st 2014
<     * checar se Ãľ pacote ip
<     * tiro cabeÃğalho ip
<     * checar se Ãľ pacote aodv (verificar socket 654)
<     * olho cabeÃğalho TypeHeader genÃľrico
<     * verifico se Ãľ RREQ
<     * se sim, m_routeRequested = true
<     * coloco cabeÃğalho ip de volta
<     */
<   Ptr<Packet> copy = packet->Copy ();
<   PacketMetadata::ItemIterator metadataIterator = copy->BeginItem ();
<   PacketMetadata::Item item;
<   while (metadataIterator.HasNext ())
<   {
<     item = metadataIterator.Next ();
<     NS_LOG_FUNCTION ("item name: " << item.tid.GetName ());
<     if (item.tid.GetName () == "ns3::aodv::RreqHeader")
<     {
<       m_stationManager->SetRouteRequester (addr, true);
<     }
<   }
< }
<
< void // RIMA AODV
< MacLow::CheckRouteReply (Ptr<Packet> packet, Mac48Address addr)
< {
```

```
<   Ptr<Packet> copy = packet->Copy ();
<   PacketMetadata::ItemIterator metadataIterator = copy->BeginItem ();
<   PacketMetadata::Item item;
<   while (metadataIterator.HasNext ())
<   {
<     item = metadataIterator.Next ();
<     NS_LOG_FUNCTION ("item name: " << item.tid.GetName ());
<     if (item.tid.GetName () == "ns3::aodv::RrepHeader")
<     {
<       m_stationManager->SetRouteReplier (addr, true);
<       Time activeRouteTimeout = Seconds (3);
<       Simulator::Schedule (activeRouteTimeout, &MacLow::ResetRoute, this, addr);
<     }
<   }
< }
<
< void // RIMA AODV
< MacLow::CheckRouteError (Ptr<Packet> packet, Mac48Address addr)
< {
<   Ptr<Packet> copy = packet->Copy ();
<   PacketMetadata::ItemIterator metadataIterator = copy->BeginItem ();
<   PacketMetadata::Item item;
<   while (metadataIterator.HasNext ())
<   {
<     item = metadataIterator.Next ();
<     NS_LOG_FUNCTION ("item name: " << item.tid.GetName ());
<     if (item.tid.GetName () == "ns3::aodv::RerrHeader")
<     {
<       m_stationManager->SetRouteReplier (addr, false);
<       m_stationManager->SetRouteRequester (addr, false);
<     }
<   }
< }
<
< void // RIMA AODV
< MacLow::ResetRoute (Mac48Address addr)
< {
<   m_stationManager->SetRouteReplier (addr, false);
< }
<
758,766c589,590
<   /*
<    * Se nĂčo havia pacote na fila, entĂčo o objeto Ptr<const Packet> packet
<    * foi passado nulo, logo nĂčo Ăľ necessĂąrio realizar a cĂşpia para o MacLow.
<    */
<   if (packet != 0)
<   {
<     m_currentPacket = packet->Copy ();
<     m_currentHdr = *hdr;
<   }
---
>   m_currentPacket = packet->Copy ();
>   m_currentHdr = *hdr;
773,777c597,600
<   /*
<    * Verificar se foi RTR foi habilitado (envio de pacotes comuns)
<    * ou nĂčo (envio de broadcast)
<    */
<   if (m_txParams.MustSendRtr ())
---
>   NS_LOG_DEBUG ("startTx size=" << GetSize (m_currentPacket, &m_currentHdr) <<
>                 ", to=" << m_currentHdr.GetAddr1 () << ", listener=" << m_listener);
>
>   if (m_txParams.MustSendRts ())
779c602
<       SendRtrForPacket ();
---
>       SendRtsForPacket ();
781c604
<   else if (packet != 0)
---
>   else
783,785d605
<       NS_LOG_DEBUG ("startTx size=" << GetSize (m_currentPacket, &m_currentHdr) <<
<      ", to=" << m_currentHdr.GetAddr1 () << ", listener=" << m_listener);
<
846,857d665
<
<   /*
<    * Atualizar lista de vizinhos
<    */
```

```
<   if (!m_stationManager->IsNeighbor (hdr.GetAddr2 ())
<       && hdr.GetAddr2 () != Mac48Address ("00:00:00:00:00:00"))
<   {
< //     NS_LOG_DEBUG ("Vizinhos de " << m_self);
<     m_stationManager->AddNeighbor (hdr.GetAddr2 ());
<   }
<   m_stationManager->UpdateNeighborhood ();
<   m_stationManager->RecSnr (hdr.GetAddr2 (), rxSnr);
912,983d719
<   else if (hdr.IsRtr ())
<     {
<       /*
<        * A STA that is addressed by an RTR frame shall transmit a DATA frame after a SIFS
<        * period if the NAV at the STA receiving the RTR frame indicates that the medium is
<        * idle. If the NAV at the STA receiving the RTR indicates the medium is not idle,
<        * that STA shall not respond to the RTR frame.
<        */
<       if (isPrevNavZero && hdr.GetAddr1 () == m_self)
<         {
<    if (m_currentPacket == 0)
<    {
<      /*
<       * Verifica se hÃą algum pacote na fila para o endereÃğo de origem do RTR.
<       */
<      m_listener->DequeueByAddress (hdr.GetAddr2 ());
<    }
<           NS_LOG_DEBUG ("rx RTR from=" << hdr.GetAddr2 () << " to=" << hdr.GetAddr1 ());
<           NS_ASSERT (m_sendDataEvent.IsExpired ());
<           m_stationManager->ReportRxOk (hdr.GetAddr2 (), &hdr,
<                                         rxSnr, txMode);
<
<    if (m_currentHdr.GetAddr1 () == hdr.GetAddr2 () && m_currentPacket != 0)
<      /*
<       * O pacote atual Ãľ destinado ao endereÃğo requisitante.
<       * Enviar o pacote de dados para o destino.
<       */
<    {
<      m_txParams.EnableAck ();
<      m_sendDataEvent = Simulator::Schedule (GetSifs (),
<    &MacLow::SendDataAfterRtr, this,
<    hdr.GetAddr2 (),
<    hdr.GetDuration (),
<    txMode,
<    rxSnr);
<    }
<    else
<      /*
<       * O pacote atual nÃčo Ãľ destinado ao endereÃğo requisitante.
<       * Enviar o resposta negativa (NTS) para o destino.
<       */
<    {
<      m_sendNtsEvent = Simulator::Schedule (GetSifs (), // criar objeto m_sendNtsEvent - ok fadhil
<    &MacLow::SendNtsAfterRtr, this,
<    hdr.GetAddr2 (),
<    hdr.GetDuration (),
<    txMode,
<    rxSnr);
<    }
<  }
<       else
<  /*
<   * Foi recebido um RTR alheio.
<   * Apenas atualizar NAV e aguardar o final da transmissÃčo.
<   */
<         {
<           NS_LOG_DEBUG ("rx RTR from=" << hdr.GetAddr2 () << ", not for me");
<         }
<     }
<   else if (hdr.IsNts ()
<     && hdr.GetAddr1 () == m_self)
<     {
<       NS_LOG_DEBUG ("rx NTS from=" << m_currentPollAddr << ", no DATA for " << m_self);
<       SnrTag tag;
<       packet->RemovePacketTag (tag);
<       m_stationManager->ReportRxOk (m_currentPollAddr, &hdr, rxSnr, txMode);
<       m_stationManager->ReportRtrOk (m_currentPollAddr, rxSnr, txMode, tag.Get ());
<       UpdatePollingTable (hdr.GetAddr2 (), 0);
<       m_dataTimeoutEvent.Cancel ();
<       NotifyDataTimeoutResetNow ();
<       m_listener->GotNts ();
<     }
```

```
1084,1109d819
<
<       // RIMA
<       SnrTag tag;
<       packet->RemovePacketTag (tag);
<       m_stationManager->ReportRtrOk (m_currentPollAddr,
<                                     rxSnr, txMode, tag.Get ());
<       m_dataTimeoutEvent.Cancel ();
<       NotifyDataTimeoutResetNow ();
<       m_listener->GotData (rxSnr, txMode);
<
< //       CheckRouteRequest (packet, hdr.GetAddr2 ()); // RIMA AODV
<
<       // RIMA DATA STREAM
<       bool moreData = false;
<       if (m_stationManager->IsMoreDataEnabled ())
<       {
<  moreData = hdr.IsMoreData ();
<       }
<       m_stationManager->SetMoreData (hdr.GetAddr2 (), moreData);
<       if (!moreData)
<       {
<  // depois que termina o burst (nÃčo hÃą mais dados),
<  // atualiza a prob de sucesso de handshake
<  UpdatePollingTable (hdr.GetAddr2 (), 1); // RIMA
<       }
<
1232,1238d941
< MacLow::GetRtrSize (void) const // RIMA
< {
<   WifiMacHeader rtr;
<   rtr.SetType (WIFI_MAC_CTL_RTR);
<   return rtr.GetSize () + 4;
< }
< uint32_t
1310,1315d1012
< WifiTxVector // RIMA
< MacLow::GetRtrTxVector (Mac48Address address) const
< {
<   return m_stationManager->GetRtrTxVector (address); // TODO
< }
<
1353,1358d1049
< WifiTxVector // RIMA
< MacLow::GetNtsTxVectorForRtr (Mac48Address to) const
< {
<   return m_stationManager->GetRtrTxVector (to); // a priori utilizar o mesmo txMode do RTR
< }
<
1458c1149
<       if (hdr.IsRtr () && navUpdated)
---
>       if (hdr.IsRts () && navUpdated)
1533,1548d1223
< void // RIMA
< MacLow::NotifyDataTimeoutStartNow (Time duration)
< {
<   for (DcfListenersCI i = m_dcfListeners.begin (); i != m_dcfListeners.end (); i++)
<     {
<       (*i)->DataTimeoutStart (duration);
<     }
< }
< void // RIMA
< MacLow::NotifyDataTimeoutResetNow ()
< {
<   for (DcfListenersCI i = m_dcfListeners.begin (); i != m_dcfListeners.end (); i++)
<     {
<       (*i)->DataTimeoutReset ();
<     }
< }
1580,1599d1254
< void // RIMA
< MacLow::DataTimeout (void)
< {
<   NS_LOG_FUNCTION (this);
<   NS_LOG_DEBUG ("Data timeout");
<   // XXX: should check that there was no rx start before now.
<   // we should restart a new data timeout now until the expected
<   // end of rx if there was a rx start before now.
<   m_stationManager->ReportRtrFailed (m_currentPollAddr);
< //   UpdatePollingTable (m_currentPollAddr, 0);
```

```
<   MacLowTransmissionListener *listener = m_listener;
<   // Normalmente, se apaga o listener pois Ãľ criado um novo
<   // quando comeÃğa um novo handshake no StartTransmission.
<   // PorÃľm, sÃş Ãľ chamado o StartTransmission quando faz polling.
<   // Quando vc nÃčo faz polling, vc estÃą esperando receber e
<   // o listener deve estar sempre ativo.
< //   m_listener = 0;
<   listener->MissedData ();
< }
<
1624,1629c1279
<   // Normalmente, se apaga o listener pois Ãľ criado um novo
<   // quando comeÃğa um novo handshake no StartTransmission.
<   // PorÃľm, sÃş Ãľ chamado o StartTransmission quando faz polling.
<   // Quando vc nÃčo faz polling, vc estÃą esperando receber e
<   // o listener deve estar sempre ativo.
<   // m_listener = 0; // RIMA Apr 12th 2013
---
>   m_listener = 0;
1679,1750d1328
< void // RIMA
< MacLow::SendRtrForPacket (void)
< {
<   /* send an RTR polling for a chosen destination */
<
<   WifiMacHeader rtr;
<   rtr.SetType (WIFI_MAC_CTL_RTR);
<   rtr.SetDsNotFrom ();
<   rtr.SetDsNotTo ();
<   rtr.SetNoRetry ();
<   rtr.SetNoMoreFragments ();
<   rtr.SetAddr2 (m_self); // endereÃğo do remetente do RTR
<
<   if (!m_stationManager->IsNeighborhoodEmpty ())
<   {
<     m_currentPollAddr =  GetPollingAddress (); // mÃľtodo que retorna o endereÃğo de quem deve ser consultado
<   }
<   else
<   {
<     m_currentPollAddr = Mac48Address ("00:00:00:00:00:00");
<   }
<   Mac48Address addr1 = m_currentPollAddr;
<   rtr.SetAddr1 (addr1); // endereÃğo do destino do RTR
<
<   /* No mÃľtodo SendRtsForPacket original, Ãľ utilizado o WifiMode do quadro de dados (dataTxMode)
<    * que depende do m_currentPacket para o cÃąlculo da duraÃğÃčo da transmissÃčo (txDuration). Como
<    * nÃčo hÃą aqui um m_currentPacket (Ãľ nulo), o WifiMode Ãľ o padrÃčo enquanto nÃčo haver decisÃčo de
<    * projeto melhor.
<    */
<   WifiTxVector rtrTxVector = GetRtrTxVector (m_self);
<   Time duration = Seconds (0);
<
<   WifiPreamble preamble;
<   //standard says RTS packets can have GF format sec 9.6.0e.1 page 110 bullet b 2
<     preamble=WIFI_PREAMBLE_LONG;
<
<     duration += GetSifs ();
<     /* duraÃğÃčo mÃąxima de um pacote
<      * Deve estimar a duraÃğÃčo total da transmissÃčo sem saber o tamanho do quadro de dados que
<      * se espera receber. Como o receptor nÃčo tem como fazer essa previsÃčo, Ãľ estimado o pior
<      * caso que Ãľ a de um quadro com o tamanho mÃąximo MTU (1500 bytes).
<      */
<     uint16_t maxDataSize = 1500; // MTU - The MAC-level Maximum Transmission Unit
<     duration += m_phy->CalculateTxDuration (maxDataSize, rtrTxVector, preamble);
<     duration += GetSifs ();
<     duration += GetAckDuration (m_self, rtrTxVector);
<
<   /* Este valor da duraÃğÃčo da transmissÃčo Ãľ guardado no cabeÃğalho do RTR.
<    * Todos os nÃşs que receberem este RTR atualizarÃčo seus NAVs com este valor
<    * que Ãľ a priori a duraÃğÃčo mÃąxima de uma transmissÃčo. Entretanto, quando
<    * os nÃşs receberem o DATA de resposta, o cabeÃğalho do DATA informarÃą o valor
<    * correto da duraÃğÃčo de sua transmissÃčo, assim os nÃşs atualizarÃčo seus NAVs
<    * novamente e portanto, ficarÃčo aguardando pelo tempo correto.
<    */
<   rtr.SetDuration (duration);
<
<   Time txDuration = m_phy->CalculateTxDuration (GetRtrSize (), rtrTxVector, preamble);
<   Time timerDelay = txDuration + duration - GetAckDuration (m_self, rtrTxVector);
< //   NS_LOG_DEBUG ("timerDelay=" << timerDelay);
<
<   NS_ASSERT (m_dataTimeoutEvent.IsExpired ());
```

```
<   NotifyDataTimeoutStartNow (timerDelay);
<   m_dataTimeoutEvent = Simulator::Schedule (timerDelay, &MacLow::DataTimeout, this);
<
<   Ptr<Packet> packet = Create<Packet> ();
<   packet->AddHeader (rtr);
<   WifiMacTrailer fcs;
<   packet->AddTrailer (fcs);
<
<   ForwardDown (packet, &rtr, rtrTxVector, preamble);
< }
<
2073,2165d1650
< void // RIMA
< MacLow::SendDataAfterRtr (Mac48Address source, Time duration, WifiMode txMode, double rtrSnr)
< {
<   NS_LOG_FUNCTION (this);
<   /* send the second step in a
<    * RTR/DATA/ACK handshake
<    *
<    * mÃľtodo anÃąlogo ao SendDataAfterCts
<    * m_currentPacket Ãľ o pacote que veio da fila da camada superior
<    */
<   NS_ASSERT (m_currentPacket != 0);
<
<   WifiTxVector dataTxVector = GetDataTxVector (m_currentPacket, &m_currentHdr);
<
<   StartDataTxTimers (dataTxVector);
<
<   WifiPreamble preamble;
<      preamble=WIFI_PREAMBLE_LONG;
<
<   Time newDuration = Seconds (0);
<   newDuration += GetSifs ();
<   newDuration += GetAckDuration (m_currentHdr.GetAddr1 (), dataTxVector);
<
<   /*
<    * Como a duraÃğÃčo calculada pelo RTR foi uma estimativa do pior caso (duraÃğÃčo mÃąxima),
<    * a nova duraÃğÃčo deve ser atualizada no envio do quadro DATA. A duraÃğÃčo calculada aqui
<    * Ãľ menor ou igual Ãă calculada no RTR. Portanto, apenas subtrair as duraÃğÃţes jÃą passadas
<    * (SIFS e DATA) seria diferente do que usar a duraÃğÃčo dos eventos seguintes (SIFS e ACK).
<    */
<   duration = newDuration;
<   NS_ASSERT (duration >= MicroSeconds (0));
<   m_currentHdr.SetDuration (duration);
<
<   m_currentPacket->AddHeader (m_currentHdr);
<   WifiMacTrailer fcs;
<   m_currentPacket->AddTrailer (fcs);
<
<   SnrTag tag;
<   tag.Set (rtrSnr);
<   m_currentPacket->AddPacketTag (tag);
<
< //   CheckRouteReply (m_currentPacket, m_currentHdr.GetAddr1 ()); // RIMA AODV
< //   CheckRouteError (m_currentPacket, m_currentHdr.GetAddr1 ()); // RIMA AODV
<
<   ForwardDown (m_currentPacket, &m_currentHdr, dataTxVector, preamble);
<   m_currentPacket = 0;
< }
<
< void // RIMA
< MacLow::SendNtsAfterRtr (Mac48Address source, Time duration, WifiMode rtrTxMode, double rtrSnr)
< {
<   NS_LOG_FUNCTION (this << source << duration << rtrTxMode << rtrSnr);
<   /* no DATA to transmit
<    * send a NTS when you receive a RTR
<    * right after SIFS.
<    */
<   WifiTxVector ntsTxVector = GetNtsTxVectorForRtr (source);
<
<   WifiPreamble preamble;
<      preamble=WIFI_PREAMBLE_LONG;
<
<   WifiMacHeader nts;
<   nts.SetType (WIFI_MAC_CTL_NTS);
<   nts.SetDsNotFrom ();
<   nts.SetDsNotTo ();
<   nts.SetNoMoreFragments ();
<   nts.SetNoRetry ();
<   nts.SetAddr1 (source);
<   nts.SetAddr2 (m_self);
```

```
<   /*
<    * A duraÃğÃčo original da transmissÃčo (RTR+DATA+ACK) Ãľ cancelada
<    * jÃą que, como nÃčo hÃą dados para ser transmitido, a duraÃğÃčo do
<    * handshake Ãľ diminuida. Assim, o NAV Ãľ atualizado mais cedo.
<    */
<     uint16_t maxDataSize = 1500; // MTU - The MAC-level Maximum Transmission Unit
<     duration -= m_phy->CalculateTxDuration (maxDataSize, ntsTxVector, preamble);
<     duration -= GetSifs ();
<     duration -= GetAckDuration (source, ntsTxVector);
<   NS_ASSERT (duration >= MicroSeconds (0));
<   nts.SetDuration (duration);
<
<   Ptr<Packet> packet = Create<Packet> ();
<   packet->AddHeader (nts);
<   WifiMacTrailer fcs;
<   packet->AddTrailer (fcs);
<
<   SnrTag tag;
<   tag.Set (rtrSnr);
<   packet->AddPacketTag (tag);
<
<   ForwardDown (packet, &nts, ntsTxVector, preamble);
< }
<
2216c1701
< //   m_listener = 0; // RIMA
---
>   m_listener = 0;
```

IV.5 RegularWifiMac

IV.5.1 regular-wifi-mac.h

IV.5.2 regular-wifi-mac.cc

```
718d717
<       cwmax = 1023;
720,721c719
< //       cwmax = m_dca->GetMaxCw ();
<       NS_LOG_DEBUG ("cwmin=" << cwmin << " cwmax=" << cwmax);
---
>       cwmax = 1023;
```

IV.6 WifiMacHeader

IV.6.1 wifi-mac-header.h

```
5d4
<  * Copyright (c) 2013, 2014 NERDs UnB
22,23d20
<  * Modifier: Fadhil Firyaguna <firyaguna@ieee.org>
<  * Modifier: Mateus Marcuzzo <mateusmarcuzzo@ieee.org>
46,48d42
<
<   WIFI_MAC_CTL_RTR, // RIMA - Request to Receive
<   WIFI_MAC_CTL_NTS, // RIMA - Nothing to Send
249,256d242
<    * Un-set the More Data bit in the Frame Control Field
<    */
<   void SetNoMoreData (void);
<   /**
<    * Set the More Data bit in the Frame Control field
<    */
<   void SetMoreData (void);
<   /**
391,402d376
<    * Return true if the header is a RTR header.
<    *
<    * \return true if the header is a RTR header, false otherwise
```

```
<    */
<   bool IsRtr (void) const; // RIMA
<   /**
<    * Return true if the header is a NTS header.
<    *
<    * \return true if the header is a NTS header, false otherwise
<    */
<   bool IsNts (void) const; // RIMA
<   /**
541,546d514
<   /**
<    * Return if the More Data bit is set.
<    *
<    * \return true if the More Data bit is set, false otherwise
<    */
<   bool IsMoreData (void) const;
```

IV.6.2 wifi-mac-header.cc

```
5d4
<  * Copyright (c) 2013, 2014 NERds UnB
22,23d20
<  * Modifier: Fadhil Firyaguna <firyaguna@ieee.org>
<  * Modifier: Mateus Marcuzzo <mateusmarcuzzo@ieee.org>
47,49c44
<   SUBTYPE_CTL_CTLWRAPPER=7,
<   SUBTYPE_CTL_RTR = 14, // RIMA
<   SUBTYPE_CTL_NTS = 15  // RIMA
---
>   SUBTYPE_CTL_CTLWRAPPER=7
180,187d174
<     case WIFI_MAC_CTL_RTR: // RIMA
<       m_ctrlType = TYPE_CTL;
<       m_ctrlSubtype = SUBTYPE_CTL_RTR;
<       break;
<     case WIFI_MAC_CTL_NTS: // RIMA
<       m_ctrlType = TYPE_CTL;
<       m_ctrlSubtype = SUBTYPE_CTL_NTS;
<       break;
345,352d331
< } // RIMA DATA STREAM
< void WifiMacHeader::SetNoMoreData (void)
< {
<   m_ctrlMoreData = 0;
< } // RIMA DATA STREAM
< void WifiMacHeader::SetMoreData (void)
< {
<   m_ctrlMoreData = 1;
516,521d494
<         case SUBTYPE_CTL_RTR: // RIMA
<           return WIFI_MAC_CTL_RTR;
<           break;
<         case SUBTYPE_CTL_NTS: // RIMA
<           return WIFI_MAC_CTL_NTS;
<           break;
641,650d613
< bool // RIMA
< WifiMacHeader::IsRtr (void) const
< {
<   return (GetType () == WIFI_MAC_CTL_RTR);
< }
< bool // RIMA
< WifiMacHeader::IsNts (void) const
< {
<   return (GetType () == WIFI_MAC_CTL_NTS);
< }
768,772d730
< bool // RIMA DATA STREAM
< WifiMacHeader::IsMoreData (void) const
< {
<   return (m_ctrlMoreData == 1);
< }
913d870
<         case SUBTYPE_CTL_RTR: // RIMA
917d873
<         case SUBTYPE_CTL_NTS: // RIMA
955,956d910
<       FOO (CTL_RTR); // RIMA
<       FOO (CTL_NTS); // RIMA
```

```
1031d984
<     case WIFI_MAC_CTL_RTR: // RIMA
1036d988
<     case WIFI_MAC_CTL_NTS: // RIMA
1141d1092
<         case SUBTYPE_CTL_RTR: // RIMA
1145d1095
<         case SUBTYPE_CTL_NTS: // RIMA
1197d1146
<         case SUBTYPE_CTL_RTR: // RIMA
1201d1149
<         case SUBTYPE_CTL_NTS: // RIMA
```

IV.7 WifiMacQueue

IV.7.1 wifi-mac-queue.h

```
5d4
<  * Copyright (c) 2013, 2014 NERdS UnB
22d20
<  * Author: Fadhil Firyaguna <firyaguna@ieee.org>
146,169d143
<   /**
<    * Searchs and returns, if is present in this queue, first packet having
<    * address equals to <i>addr</i>.
<    * This method removes the packet from this queue.
<    * Is typically used by ns3::MacLow in order to return the packet
<    * requested by an specified poller.
<    */
<   Ptr<const Packet> DequeueByAddress (WifiMacHeader *hdr, Mac48Address addr); // RIMA
<   /**
<    * Searchs and returns, if is present in this queue, first packet having
<    * address equals to <i>addr</i>.
<    * This method doesn't removes the packet from this queue.
<    * Is typically used by ns3::MacLow in order to return the packet
<    * requested by an specified poller.
<    */
<   Ptr<const Packet> PeekByAddress (WifiMacHeader *hdr, Mac48Address addr); // RIMA
<   /**
<    * Searchs and returns, if is present in this queue, first packet having
<    * address indicated by <i>type</i> equals to <i>addr</i>, and tid
<    * equals to <i>tid</i>. This method removes the packet from this queue.
<    * Is typically used by ns3::EdcaTxopN in order to perform correct MSDU
<    * aggregation (A-MSDU).
<    */
<   void GetQueueStatus (Mac48Address addr); // RIMA Dec 20th 2013
```

IV.7.2 wifi-mac-queue.cc

```
5d4
<  * Copyright (c) 2013, 2014 NERdS UnB
22d20
<  * Author: Fadhil Firyaguna <firyaguna@ieee.org>
32,36d29
< #include <iostream>
< #include <fstream>
< #include <cstdio>
< #include <iomanip>
<
218,310d210
< }
<
< Ptr<const Packet> // RIMA DATA STREAM
< WifiMacQueue::DequeueByAddress (WifiMacHeader *hdr, Mac48Address dest)
< {
<   Cleanup ();
<   Ptr<const Packet> packet = 0;
<   if (!m_queue.empty ())
<     {
<       PacketQueueI it;
<       for (it = m_queue.begin (); it != m_queue.end (); ++it)
<         {
<           if (it->hdr.GetAddr1 () == dest)
<             {
<               packet = it->packet;
```

```
<
<         it++;
<         bool nextIsForDest = (it->hdr.GetAddr1 () == dest);
<         it--;
<         if (nextIsForDest)
<  { // se o prÃşximo tbm Ãľ
<    // set moredata flag
<    it->hdr.SetMoreData ();
<  }
<         else
<  {
<    it->hdr.SetNoMoreData ();
<  }
<
<               *hdr = it->hdr;
<               m_queue.erase (it);
<               m_size--;
<               break;
<             }
<         }
<     }
<   return packet;
< }
<
< Ptr<const Packet> // RIMA
< WifiMacQueue::PeekByAddress (WifiMacHeader *hdr, Mac48Address dest)
< {
<   Cleanup ();
<   if (!m_queue.empty ())
<     {
<       PacketQueueI it;
<       for (it = m_queue.begin (); it != m_queue.end (); ++it)
<         {
<           if (it->hdr.GetAddr1 () == dest)
<             {
<               *hdr = it->hdr;
<               return it->packet;
<             }
<         }
<     }
<   return 0;
< }
< // Dec 20th 2013
< void
< WifiMacQueue::GetQueueStatus (Mac48Address self)
< {
<   if (!m_queue.empty ())
<   {
<   // contar quantos pacotes de cada endereÃğo existem na fila
<     int numPacketsbyNode[51];
<     for (int i=0; i<51; i++) { numPacketsbyNode[i] = 0; }
<     PacketQueueI it;
<     WifiMacHeader hdr;
<     for (it = m_queue.begin (); it != m_queue.end (); ++it)
<     {
<       uint8_t addr[6];
<       it->hdr.GetAddr1 ().CopyTo (addr);
<       numPacketsbyNode[addr[5]]++;
<     }
<   // imprimir histograma no arquivo
<     uint8_t s_add[6];
<     self.CopyTo (s_add);
<     std::ofstream hist;
<     char filename[50] = "tracing/histograms/hist_node";
<     sprintf (filename, "%s%d", filename, s_add[5]);
<     hist.open (filename, std::ios::app);
<     if (hist)
<     {
< //       std::cout << "getting queue status of " << self << std::endl;
<       hist << Simulator::Now ().GetSeconds () << ";";
<       for (int i=1; i<51; i++)
<       {
<  hist << numPacketsbyNode[i] << ";";
<       }
<       hist << std::endl;
<     }
<     hist.close ();
<   }
```

IV.8 WifiRemoteStationManager

IV.8.1 wifi-remote-station-manager.h

```
4d3
<  * Copyright (c) 2013,2014 NERds UnB
20,21d18
<  * Modifier: Fadhil Firyaguna <firyaguna@ieee.org>
<  * Modifier: Mateus Marcuzzo <mateusmarcuzzo@ieee.org>
354,360d350
<    *
<    * \return the transmission mode to use to send the RTR prior to the
<    *         transmission of the data packet itself.
<    */
<   WifiTxVector GetRtrTxVector (Mac48Address address);
<   /**
<    * \param address remote address
391,395d380
<    * Should be invoked whenever the DataTimeout associated to a transmission
<    * attempt expires.
<    */
<   void ReportRtrFailed (Mac48Address address); // RIMA
<   /**
412,421d396
<    * Should be invoked whenever we receive the DATA associated to an RTR
<    * we just sent.
<    *
<    * \param address the address of the receiver
<    * \param dataSnr the SNR of the DATA we received
<    * \param dataMode the WifiMode the receiver used to send the DATA
<    * \param rtsSnr the SNR of the RTS we sent
<    */
<   void ReportRtrOk (Mac48Address address, double dataSnr, WifiMode dataMode, double rtsSnr); // RIMA
<   /**
447,453d421
<    * Should be invoked after calling ReportRtrFailed if
<    * NeedRtrRetransmission returns false
<    *
<    * \param address the address of the polled node
<    */
<   void ReportFinalRtrFailed (Mac48Address address); // RIMA
<   /**
483,489d450
<    * \returns true if we want to use an RTR polling for packet
<    *         false otherwise.
<    */
<   bool NeedRtr (Mac48Address address); // RIMA
<
<   /**
<    * \param address remote address
507,512d467
<    * \returns true if we want to restart a failed RTR polling,
<    *         false otherwise.
<    */
<   bool NeedRtrRetransmission (Mac48Address address); // RIMA
<   /**
<    * \param address remote address
606,621d560
<
<    //----------- Neighborhood -------------//
<   bool IsNeighbor (Mac48Address addr);
<   void AddNeighbor (Mac48Address addr);
<   bool IsNeighborhoodEmpty (void);
<   Mac48Address NextPollingAddress (Mac48Address m_lastAddr);
<   void UpdateNeighborhood (void);
<   void UpdateTxProbability (Mac48Address addr, int success);
<   void RecRxData (Mac48Address addr, uint32_t rxBytes);
<   void SetRouteRequester (Mac48Address addr, bool rreq); // RIMA AODV
<   void SetRouteReplier (Mac48Address addr, bool rrep); // RIMA AODV
<   void RecSnr (Mac48Address addr, double snr); // ADAPTIVE POLLING
<   void SetMoreData (Mac48Address addr, bool moreData); // RIMA DATA STREAM
<   bool HasMoreData (Mac48Address addr); // RIMA DATA STREAM
<   bool IsMoreDataEnabled (void); // RIMA DATA STREAM
<   //----------- Neighborhood -------------//
832,838d770
<    * \param station the station that we failed to send RTR
<    */
< //   virtual void DoReportRtrFailed (WifiRemoteStation *station) = 0; // RIMA
<   /**
```

```
<    * This method is a pure virtual method that must be implemented by the sub-class.
<    * This allows different types of WifiRemoteStationManager to respond differently,
<    *
854,864d785
<    * \param datantsSnr the SNR of the CTS we received
<    * \param datantsMode the WifiMode the receiver used to send the CTS
<    * \param rtsSnr the SNR of the RTS we sent
<    */
< //   virtual void DoReportRtrOk (WifiRemoteStation *station,
< //                              double datantsSnr, WifiMode datantsMode, double rtrSnr) = 0; // RIMA
<   /**
<    * This method is a pure virtual method that must be implemented by the sub-class.
<    * This allows different types of WifiRemoteStationManager to respond differently,
<    *
<    * \param station the station that we successfully sent RTS
886,892d806
<    * \param station the station that we failed to send RTR
<    */
< //   virtual void DoReportFinalRtrFailed (WifiRemoteStation *station) = 0; // RIMA
<   /**
<    * This method is a pure virtual method that must be implemented by the sub-class.
<    * This allows different types of WifiRemoteStationManager to respond differently,
<    *
963,1014d876
<   //----------- Neighborhood -------------//
<   struct Neighbor
<   {
<    Mac48Address m_macAddress;
<    Time m_startTime; // neighborhood start time
<    Time m_expirationTime; // Simulator::Now () + Estimated link life
<    uint32_t m_rxData; // received data since start time
<    double m_txProb; // successful transmission probability
<    double m_pollProb; // polling probability
<    uint32_t m_pollCounter; // polling counter
<    uint32_t m_maxPollCounter; // max polling counter
<
<    // RIMA AODV CROSS LAYER - Apr 1st 2014
<    bool m_routeRequested; // true if that neighbor requested a route for me
<    bool m_routeReplied; // true if I replied a route to that neighbor
<    double m_routeTxProb; // transmission probability weighted by route existance
<
<    // ADAPTIVE POLLING - Apr 9th 2014
<    double m_snr; // average snr
<
<    // DATA STREAM - Jun 18th 2014
<    bool m_moreData;
<
<    Neighbor (Mac48Address address, Time startTime, Time expirationTime) :
<     m_macAddress (address),
<     m_startTime (startTime),
<     m_expirationTime (expirationTime),
<     m_rxData (0),
<     m_txProb (0.001),
<     m_pollProb (1),
<     m_pollCounter (0),
<     m_maxPollCounter (100),
<     m_routeRequested (false),
<     m_routeReplied (false),
<     m_routeTxProb (1),
<     m_snr (0),
<     m_moreData (false)
<    {}
<   };
<
<   typedef std::list <Neighbor *> Neighbors;
<   Neighbors m_neighbors;
<   uint32_t m_currentPollNeighbor;
<
<   double m_updateRate;
<   uint16_t m_pollingMode;
<   double m_estimatedLifeTime;
<   uint32_t m_nNeighborsThreshold;
<   double m_snrVarThreshold;
<   bool m_enableMoreData;
<   //----------- Neighborhood -------------//
<
1058,1061d919
<    * The trace source fired when the transmission of a single RTR has failed
<    */
<   TracedCallback<Mac48Address> m_macTxRtrFailed; // RIMA
<   /**
```

```
1069,1073d926
<   /**
<    * The trace source fired when the transmission of a RTR has
<    * exceeded the maximum number of attempts
<    */
<   TracedCallback<Mac48Address> m_macTxFinalRtrFailed; // RIMA
```

IV.8.2 wifi-remote-station-manager.cc

```
4d3
<  * Copyright (c) 2013,2014 NERds UnB
20,21d18
<  * Modifier: Fadhil Firyaguna <firyaguna@ieee.org>
<  * Modifier: Mateus Marcuzzo <mateusmarcuzzo@ieee.org>
37,39d33
< #include "ns3/random-variable.h"
< #include <fstream>
<
265,290d258
<     .AddAttribute ("PollingMode", "Polling discipline mode", // RIMA
<      UintegerValue (0),
<      MakeUintegerAccessor (&WifiRemoteStationManager::m_pollingMode),
<      MakeUintegerChecker<uint16_t> ())
<     .AddAttribute ("UpdateRate", "Update rate of neighbor successful tx probability" // RIMA
<                    " 'alpha' weight of the moving average computation.",
<      DoubleValue (0.05),
<      MakeDoubleAccessor (&WifiRemoteStationManager::m_updateRate),
<      MakeDoubleChecker<double> ())
<     .AddAttribute ("EstimatedLifeTime", "Estimated life time of neighbor entry on table" // RIMA
<                    " 'alpha' weight of the moving average computation.",
<      DoubleValue (2.0),
<      MakeDoubleAccessor (&WifiRemoteStationManager::m_estimatedLifeTime),
<      MakeDoubleChecker<double> ())
<     .AddAttribute ("nNeighborsThreshold", "Number of neighbors threshold", // ADAPTIVE RIMA
<      UintegerValue (5),
<      MakeUintegerAccessor (&WifiRemoteStationManager::m_nNeighborsThreshold),
<      MakeUintegerChecker<uint32_t> ())
<     .AddAttribute ("SnrVarThreshold", "SNR variation threshold", // ADAPTIVE RIMA
<      DoubleValue (5.0),
<      MakeDoubleAccessor (&WifiRemoteStationManager::m_snrVarThreshold),
<      MakeDoubleChecker<double> ())
<     .AddAttribute ("EnableMoreData", "If true, moreData function is enabled", // RIMA DATA STREAM
<                    BooleanValue (false),
<                    MakeBooleanAccessor (&WifiRemoteStationManager::m_enableMoreData),
<                    MakeBooleanChecker ())
609,615d576
< WifiTxVector // RIMA
< WifiRemoteStationManager::GetRtrTxVector (Mac48Address address)
< {
<   NS_ASSERT (!address.IsGroup ());
<   uint8_t tid = 0; // assuming non-QoS
<   return DoGetRtsTxVector (Lookup (address, tid));
< }
659,675d619
< void // RIMA
< WifiRemoteStationManager::ReportRtrFailed (Mac48Address address)
< {
<   NS_ASSERT (!address.IsGroup ());
<   uint8_t tid = 0; // assuming non-QoS
<   WifiRemoteStation *station = Lookup (address, tid);
<   if (station->m_ssrc != GetMaxSsrc ())
<   {
<     station->m_ssrc++;
<   }
<   else
<   {
<     station->m_ssrc = 0;
<   }
<   m_macTxRtrFailed (address);
< //   DoReportRtrFailed (station);  // mÃľtodo apenas de DEBUG
< }
694,704d637
< void // RIMA
< WifiRemoteStationManager::ReportRtrOk (Mac48Address address, double dataSnr, WifiMode dataMode, double rtrSnr)
< {
<   NS_ASSERT (!address.IsGroup ());
<   uint8_t tid = 0; // assuming non-QoS
<   WifiRemoteStation *station = Lookup (address, tid);
<   station->m_state->m_info.NotifyTxSuccess (station->m_ssrc);
```

```
<   station->m_ssrc = GetMaxSsrc ();
< //   station->m_ssrc = 0;
< //   DoReportRtrOk (station, dataSnr, dataMode, rtrSnr); // mÃľtodo apenas de DEBUG
< }
725,735d657
< void // RIMA
< WifiRemoteStationManager::ReportFinalRtrFailed (Mac48Address address)
< {
<   NS_ASSERT (!address.IsGroup ());
<   uint8_t tid = 0; // assuming non-QoS
<   WifiRemoteStation *station = Lookup (address, tid);
<   station->m_state->m_info.NotifyTxFailed ();
<   station->m_ssrc = GetMaxSsrc ();
<   m_macTxFinalRtrFailed (address); // ok - fadhil
< //   DoReportFinalRtrFailed (station); // mÃľtodo apenas de DEBUG
< }
767,778d688
< bool // RIMA
< WifiRemoteStationManager::NeedRtr (Mac48Address address)
< {
<   if (address.IsGroup ())
<   {
<     return false;
<   }
<   else
<   {
<     return true;
<   }
< }
818,826d727
< bool // RIMA
< WifiRemoteStationManager::NeedRtrRetransmission (Mac48Address address)
< {
<   NS_ASSERT (!address.IsGroup ());
<   uint8_t tid = 0; // assumin non-QoS
<   WifiRemoteStation *station = Lookup (address, tid);
<   bool normally = station->m_ssrc < GetMaxSsrc ();
<   return normally;
< }
1490,1881d1390
<
< //---------- Neighborhood -------------//
< bool
< WifiRemoteStationManager::IsNeighbor (Mac48Address address)
< {
<   /*
<    * Verifica se o endereÃğo for encontrado na lista de vizinhos.
<    */
<   for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<   {
<     if ((*i)->m_macAddress == address)
<     {
<       return true;
<     }
<   }
<   return false;
< }
<
< // bool
< // WifiRemoteStationManager::IsNeighborExpired (std::_List_const_iterator<ns3::WifiRemoteStationManager::Neighbor*>& n)
< // {
< //   /*
< //    * Verifica se a vizinhanÃğa expirou.
< //    */
< //   return ((*n)->m_expirationTime < Simulator::Now ());
< // }
<
< void
< WifiRemoteStationManager::AddNeighbor (Mac48Address address)
< {
<   NS_LOG_DEBUG ("antes");
<   for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<   {
<     NS_LOG_DEBUG ((*i)->m_macAddress);
<   }
<   /*
<    * Adiciona um novo endereÃğo na lista de vizinhos.
<    */
<   Time estimatedLifeTime = Seconds (m_estimatedLifeTime);
<   Time expirationTime = Simulator::Now () + estimatedLifeTime;
<   Neighbor *new_Neighbor = new Neighbor (address, Simulator::Now (), expirationTime);
```

```
<   const_cast<WifiRemoteStationManager *> (this)->m_neighbors.push_back (new_Neighbor);
<
<   NS_LOG_DEBUG ("depois");
<   for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<   {
<     NS_LOG_DEBUG ((*i)->m_macAddress);
<   }
< }
<
< bool
< WifiRemoteStationManager::IsNeighborhoodEmpty (void)
< {
<   return m_neighbors.empty ();
< }
<
< Mac48Address
< WifiRemoteStationManager::NextPollingAddress (Mac48Address m_lastAddr)
< {
<   Neighbor *temp_Neighbor;
< //   Mac48Address nextAddr;
<   bool tempCopied = false;
< //   if (m_neighbors.size () > 1)
< //   {
< //     for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
< //     {
< //       if ((*i)->m_macAddress == m_lastAddr)
< //       {
< //  temp_Neighbor = (*i);
< //  tempCopied = true;
< //  m_neighbors.remove (*i);
< //  break;
< //       }
< //     }
< //   }
< //
<   uint16_t local_pollingMode = 0;
<   if (m_pollingMode == 0) // algoritmo de decisÃčo de polling
<   {
<     /*
<      * Dynamic adaptive polling mode.
<      * Set local_pollingMode according to:
<      *  number of neighbors in table
<      *  snr variation in neighborhood table
<      */
<     uint32_t n_Neighbors = m_neighbors.size ();
<     double snrVar = 0;
<     double snrAvg = 0;
<     for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<     {
<       snrAvg = (*i)->m_snr++;
<     }
<     snrAvg = (double) snrAvg / n_Neighbors;
<
<     double sum_aux = 0;
<     for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<     {
<       sum_aux = sum_aux + ((*i)->m_snr - snrAvg)*((*i)->m_snr - snrAvg);
<     }
<     snrVar = (double) sum_aux / (n_Neighbors - 1);
<
<     if (snrVar > m_snrVarThreshold and n_Neighbors > m_nNeighborsThreshold)
<     {
<       local_pollingMode = 1; // RIPF
<     }
<     else //if (n_Neighbors > m_nNeighborsThreshold)
<     {
<       local_pollingMode = 2; // RIBB
<     }
< //     else
< //     {
< //       local_pollingMode = 3; // RIRR
< //     }
<
<   }
<   else
<   {
<     local_pollingMode = m_pollingMode;
<   }
<
<   if (local_pollingMode == 1)
<   {
```

```
<    /* Disciplina de consulta do Tiago
<     * Calcular Beta (normalizaÃğÃčo)
<     *  Somar todas as m_txProb dos vizinhos
<     *  Beta = inverso da soma
<     * Calcular probabilidade de consulta
<     *  m_pollProb = m_txProb / Beta
<     * Definir intervalos de tamanho m_pollProb para cada estaÃğÃčo entre 0 e 1
<     * Sortear um nÃžmero uniformemente aleatÃşrio entre 0 e 1
<     * Verificar o intervalo em que o nÃžmero foi sorteado correspondente a station
<     * Retornar o respectivo endereÃğo dessa station
<     */
<     //---------- Disciplina de consulta do Tiago -------------//
<     double beta = 0;
<     for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<     {
<       beta += (*i)->m_txProb;
<     }
<     beta = 1.0 / beta;
<     for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<     {
<       (*i)->m_pollProb = (*i)->m_txProb * beta;
<     }
<     for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<     {
<       NS_LOG_DEBUG ((*i)->m_macAddress << "\t" << (*i)->m_pollProb);
<     }
<
<     UniformVariable uv (0,1);
<     double x = uv.GetValue ();
<     NS_LOG_DEBUG ("sort: " << x);
<     double range = 0;
<     for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<     {
<       if (x < (*i)->m_pollProb + range)
<       {
<  if (tempCopied) m_neighbors.push_back (temp_Neighbor);
<  return (*i)->m_macAddress;
<       }
<       else
<       {
<  range += (*i)->m_pollProb;
<       }
<     }
<     return Mac48Address ("00:00:00:00:00:00");
<     //---------- Disciplina de consulta do Tiago -------------/*/
<   }
<   else if (local_pollingMode == 2)
<   {
<    /* Proportional Fair Sharing
<     * The algorithm proposed by Qualcomm performs this
<     * sharing by comparing the given rate for each user
<     * with its average throughput to date, and selecting
<     * the one with the maximum ratio.
<     */
<     //------ PROPORTIONAL FAIR --------//
<     double best = 0;
<     Neighbors::const_iterator best_i;
<     for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<     {
<       Time dt = Simulator::Now () - (*i)->m_startTime;
<       double avg = (double) (*i)->m_rxData * 8 / dt.GetSeconds (); // average throughput to date (bps)
<       double given = 1e6; // constant given rate 1 Mbps
<       double ratio = given / avg;
<       if (ratio > best)
<       {
<  best = ratio;
<  best_i = i;
<       }
<     }
<     if (tempCopied) m_neighbors.push_back (temp_Neighbor);
<     return (*best_i)->m_macAddress;
<     //------ PROPORTIONAL FAIR --------/*/
<   }
<   else if (local_pollingMode == 3)
<   {
<    /*
<     * Consulta cÃŋclica
<     */
<     //------ ROUND ROBIN --------//
<     Neighbors::const_iterator i = m_neighbors.begin ();
<     if (m_currentPollNeighbor >= m_neighbors.size ())
```

```
<     {
<       m_currentPollNeighbor = 0;
<     }
<     std::advance (i, m_currentPollNeighbor);
<     m_currentPollNeighbor++;
<     if (tempCopied) m_neighbors.push_back (temp_Neighbor);
<     return (*i)->m_macAddress;
<     //------ ROUND ROBIN --------/*/
<   }
<   else if (local_pollingMode == 4)
<   {
<     //---------- Disciplina de consulta do Tiago + ponderaÃğÃčo de rota AODV -------------//
<     for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<     {
<       double routeOn = 0;
<       if ((*i)->m_routeReplied and (*i)->m_routeRequested) routeOn = 1;
<       NS_LOG_DEBUG ("Route from " << (*i)->m_macAddress << " ON");
<       (*i)->m_routeTxProb = ((*i)->m_txProb + 6*routeOn) / 7.0;
<     }
<
<     double beta = 0;
<     for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<     {
<       beta += (*i)->m_routeTxProb;
<     }
<     beta = 1.0 / beta;
<     for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<     {
<       (*i)->m_pollProb = (*i)->m_txProb * beta;
<     }
<     for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<     {
<       NS_LOG_DEBUG ((*i)->m_macAddress << "\t" << (*i)->m_pollProb);
<     }
<
<     UniformVariable uv (0,1);
<     double x = uv.GetValue ();
<     NS_LOG_DEBUG ("sort: " << x);
<     double range = 0;
<     for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<     {
<       if (x < (*i)->m_pollProb + range)
<       {
<  if (tempCopied) m_neighbors.push_back (temp_Neighbor);
<  return (*i)->m_macAddress;
<       }
<       else
<       {
<  range += (*i)->m_pollProb;
<       }
<     }
<     return Mac48Address ("00:00:00:00:00:00");
<     //---------- Disciplina de consulta do Tiago + ponderaÃğÃčo de rota AODV -------------/*/
<   }
<   else
<   {
<    /*
<     * Escolhe aleatoriamente um vizinho para consultar.
<     */
<     //------ TEST RANDOM DISCIPLINE------//
<     Neighbors::const_iterator i = m_neighbors.begin ();
<     UniformVariable x;
<     std::advance (i, x.GetInteger (0, m_neighbors.size () - 1));
<     if (tempCopied) m_neighbors.push_back (temp_Neighbor);
<     return (*i)->m_macAddress;
<     //------ TEST ------/*/
<   }
< }
<
< void
< WifiRemoteStationManager::UpdateNeighborhood (void)
< {
<   /*
<    * Remove os vizinhos cujos tempos jÃą expiraram.
<    */
<   for (Neighbors::iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<   {
<     if ((*i)->m_expirationTime < Simulator::Now ())
<     {
<       NS_LOG_DEBUG ((*i)->m_macAddress << " expired");
<       i = m_neighbors.erase (i);
```

```
<     }
<   }
< //   NS_LOG_DEBUG ("depois");
< //   for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
< //   {
< //     NS_LOG_DEBUG ((*i)->m_macAddress);
< //   }
< }
<
< void
< WifiRemoteStationManager::UpdateTxProbability (Mac48Address addr, int eta)
< {
<   for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<   {
<     if ((*i)->m_macAddress == addr)
<     {
<       (*i)->m_txProb = (1 - m_updateRate) * (*i)->m_txProb + m_updateRate * eta;
<     }
<   }
< }
<
< void
< WifiRemoteStationManager::RecRxData (Mac48Address addr, uint32_t rxBytes)
< {
<   /*
<    * Record received data for Proportional Fair computation
<    */
<   for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<   {
<     if ((*i)->m_macAddress == addr)
<     {
<       (*i)->m_rxData += rxBytes;
<     }
<   }
< }
<
< void // ADAPTIVE POLLING
< WifiRemoteStationManager::RecSnr (Mac48Address addr, double snr)
< {
<   /*
<    * Record received SNR value and compute the weighted moving average
<    */
<   for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<   {
<     if ((*i)->m_macAddress == addr)
<     {
<       (*i)->m_snr = (1 - m_updateRate) * (*i)->m_snr + m_updateRate * snr;
<     }
<   }
< }
<
< void // RIMA AODV
< WifiRemoteStationManager::SetRouteRequester (Mac48Address addr, bool rreq)
< {
<   for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<   {
<     if ((*i)->m_macAddress == addr)
<     {
<       (*i)->m_routeReplied = rreq;
<     }
<   }
< }
<
< void // RIMA AODV
< WifiRemoteStationManager::SetRouteReplier (Mac48Address addr, bool rrep)
< {
<   for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<   {
<     if ((*i)->m_macAddress == addr)
<     {
<       (*i)->m_routeRequested = rrep;
<     }
<   }
< }
<
< void // RIMA DATA STREAM
< WifiRemoteStationManager::SetMoreData (Mac48Address addr, bool moreData)
< {
<   for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<   {
<     if ((*i)->m_macAddress == addr)
```

```
<     {
<       (*i)->m_moreData = moreData;
<     }
<   }
< }
<
< bool // RIMA DATA STREAM
< WifiRemoteStationManager::HasMoreData (Mac48Address addr)
< {
<   for (Neighbors::const_iterator i = m_neighbors.begin (); i != m_neighbors.end (); i++)
<   {
<     if ((*i)->m_macAddress == addr)
<     {
< //       std::cout << "more data to " << addr << std::endl;
<       return (*i)->m_moreData;
<     }
<   }
<   return false;
< }
<
< bool // RIMA DATA STREAM
< WifiRemoteStationManager::IsMoreDataEnabled (void)
< {
<   return m_enableMoreData;
< }
< //----------- Neighborhood -------------//
<
```