DISSERTAÇÃO DE MESTRADO

Desempenho da Estratégia do Aperto em Leilões Recursivos para Descarregamento de Tráfego via Comunicações Dispositivo-a-Dispositivo

Lucas Soares de Brito

Brasília, Março de 2016

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

DISSERTAÇÃO DE MESTRADO

Desempenho da Estratégia do Aperto em Leilões Recursivos para Descarregamento de Tráfego via Comunicações Dispositivo-a-Dispositivo

Lucas Soares de Brito

Dissertação submetida 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, CIn/UFPE _____
Examinador interno

Prof. André Costa Drummond, CIC/UnB _____
Examinador externo

Dedicatória

Dedico este trabalho a todos os que têm o prazer em descobrir e pesquisar os inúmeros segredos das máquinas que a raça humana domina.
Dedico a todos que dedicam seu tempo pesquisando nessa incrível área das comunicações entre as máquinas.
A todos que buscam todos os dias utilizar o conhecimento que temos sobre os mais variados tipos de Redes de Comunicações em prol de uma sociedade melhor, em sintonia com o ambiente em que vivemos.
E por fim, a todos que irão se beneficiar direta ou indiretamente no futuro com esta tecnologia, dentre os quais se incluem as empresas, a sociedade e o governo do Brasil e de quaisquer outros locais que venham se interessar por este trabalho técnico.

Lucas Soares de Brito

Agradecimentos

Agradeço o apoio dado a mim, direta ou indiretamente, pelas seguintes instituições: Universidade de Brasília (em especial à Faculdade de Tecnologia, ao Departamento de Engenharia Elétrica e ao programa PGEA de pós-graduação), Ministério da Educação (em especial ao programa REUNI, fonte dos recursos deste projeto) e CNPq pelo projeto ESTRADAS (Escoamento de Tráfego em Redes Ad Hoc Dinâmicas, Edital MCTI/CNPq No14/2013 Universal).

Agradeço ao professor Marcelo Menezes de Carvalho pelo apoio, paciência, orientação e atenção que teve comigo proporcionando enorme aprendizado durante todo o Mestrado. Também agradeço aos professores Paulo Roberto de Lira Gondim e Renato Mariz de Moraes pelo apoio no projeto REUNI. Agradeço também a todos os envolvidos no Laboratório de Engenharia Elétrica, o SG11. Mais especificamente aos envolvidos no laboratório GPDS (Grupo de Processamento Digital de Sinais) que me cederam espaço e estrutura para pesquisa, graças também à professora Mylène Farias

Agradeço aos colegas e amigos de Universidade. À Larissa Eglem e Tiago Bonfim por me incentivarem a começar o Mestrado e pelo exemplo que nos deram em seus trabalhos. A Fadhil Firyaguna e Juan Camilo pelo companheirismo durante as difíceis disciplinas de Mestrado e pelas contrbuições para este projeto. Aos outros "hermanos" Stephanie Alvarez (colega e amiga de sala de trabalho), Helard Becerra, Mauro Luciano, Ruben Ortega, entre outros (Ruben, Toni Rubio,...) por proporcionarem à Universidade um toque ainda mais latino e por me passarem um pouco mais de suas culturas. Aos brasileiros também, como Fadhil (quase brasileiro), Dário, Ricardo, Ana, Éverton, Thayanne, Marina e outros com quem tive o prazer de conhecer e/ou trabalhar junto. Aos amigos externos à Universidade pela companhia constante nos bons e maus momentos e pela paciência que tiveram (ou não) para esperar o fim do meu Mestrado. Por fim à equipe da Embrapa Agroenergia, entidade que me deu apoio e tempo disponível para concluir este trabalho enquanto prestei serviços de pesquisa em Bioinformática paralelamente ao Mestrado (vale um agradecimento especial a Eduardo Formighieri pelas dicas sobre Mestrado e pesquisa, ao diretor da Embrapa Manoel Sousa e ao CNPq, que me sustentou nesse período final). E principalmente não posso deixar de agradecer à minha família e a Deus. Aos membros de minha família pela compreensão e apoio durante esses primeiros anos acadêmicos. Eles me deram a paciência e apoio necessários para passar por essa grande experiência.

Lucas Soares de Brito

"I applied my mind to study and to explore by wisdom all that is done under the heavens. What a heavy burden God has laid on mankind! ".

— Ecclesiastes 1:13.

RESUMO

O crescimento explosivo do tráfego de dados móveis nas redes das operadoras de telefonia móvel (MNO, do inglês *mobile network operator*) observado nos últimos anos tem levado as operadoras a procurar maneiras eficientes de descongestionar a sua infra-estrutura central. Em particular, as comunicações dispositivo-a-dispositivo têm surgido como uma tecnologia viável para alcançar este objetivo. No entanto, a fim de que isto se torne realidade, os clientes das MNOs necessitam ser devidamente incentivados a compartilhar os recursos dos seus dispositivos para o benefício dos outros usuários. Uma solução para promover a colaboração entre usuários é a implementação de leilões recursivos, ou seja, "licitações" salto-a-salto para encaminhar pacotes para o seu destino. Neste cenário, cada cliente pode implementar sua própria estratégia de participação nos leilões, a fim de que ele possa compartilhar os incentivos fornecidos pela MNO para realizar sua tarefa. A operadora estabelece um orçamento máximo para cada pacote, e os clientes pagam uma multa se o pacote não for entregue dentro de um dado prazo. Nesta dissertação, apresentamos a avaliação de desempenho da *Estratégia do Aperto* para tais leilões recursivos. Esta estratégia baseia-se na ideia do quão "apertado" um nó está para encaminhar um pacote para o seu destino dentro do prazo estipulado. Diferentes funções de preferência (para decisão do vencedor dos leilões) são investigadas, e o desempenho da estratégia é estudado em redes homogêneas, ou seja, quando todos os dispositivos implementam a mesma estratégia. Este estudo é realizado com base em simulações a eventos discretos em cenários estáticos e móveis. Para comparação de desempenho, duas estratégias básicas também são investigadas: uma que prioriza a entrega de pacotes em detrimento de ganhos orçamentários, e uma gananciosa, que sempre escolhe o menor lance independente da entrega de pacote dentro do prazo. Todas estratégias são avaliadas a partir de simulações computacionais utilizando o simulador ns-3, e comparadas segundo as medidas de taxa de entrega de pacotes, ganho (lucro) médio por nó, justiça na distribuição dos ganhos, e número de saltos médio até o destino. Os resultados apresentados mostram que a *Estratégia do Aperto* é mais eficaz que simplesmente usar roteamento de menor caminho sem levar em conta os lances dos nós. Isso acontece porque os nós que percebem uma condição "apertada" para entregar um pacote dentro do prazo anunciado desencorajam o leiloeiro a escolhê-los escolhendo lances altos. A única métrica que a *Estratégia do Aperto* é levemente inferior é a justiça, apesar de não sofrer grandes variações conforme se aumenta a mobilidade, ou seja, é mais robusta.

ABSTRACT

The explosive growth of mobile data traffic in the last few years has lead mobile network operators (MNO) to seek efficient ways to offload their core infrastructure. In particular, device-to-device communications has emerged as a key technology to accomplish that. In order to work, the MNO's clients need to be properly incentivized to share their devices' resources to the benefit of others. One solution to promote user collaboration is the deployment of recursive auctions, i.e., hop-by-hop bidding contests for forwarding packets to their destinations. In this scenario, each client can implement its own auction strategy, so it can share the incentives provided by the MNO (payments, etc.) to accomplish its job. The operator sets a maximum budget for each packet, and the clients pay a fine if the packet is not delivered within a given deadline. In this dissertation, the *Tightness* strategy for such recursive auctions is evaluated, which is based on the idea of how "tight" a node is to forward a packet to its destination within the associated deadline. Different preference functions (for auction winner decision) are investigated, and the performance of the *Tightness* strategy is studied in homogeneous networks, i.e., when all devices implement the same strategy. This study is carried out based on discrete-event simulations under static and mobile scenarios. For performance comparison, two baseline strategies are also investigated: one that prioritizes packet delivery over budget gains, and a greedy one, that always pick the lowest bid regardless of packet delivery within the deadline. All strategies are evaluated on discrete-event simulations based on the ns-3 simulator, and compared according to packet delivery ratio, average budget per node, budget fairness, and average number of hops to destination. The presented results show that the *Tightness* strategy is more effective than simply using shortest-path routing without taking into account the nodes' bids. This happens because the nodes who perceive a "tight" condition to deliver a packet within the announced deadline discourage the auctioneer from choosing them by bidding high values. The only metric that the *Tightness* strategy is slightly lower is the fairness, despite not presenting higher variations as the mobility increases, i.e., it is more robust.

CONTENTS

List of Figures

List of Tables

Chapter 1

Introdução

De acordo com uma série de projeções feitas pela indústria [1, 6], o crescimento explosivo do tráfego de dados móveis que temos testemunhado até o momento deverá continuar nos próximos anos, sobretudo impulsionado por uma infinidade de comunicações máquina-a-máquina e um aumento crescente do tráfego de vídeo digital (ver Figura 1.1). Essa demanda de tráfego sem precedentes imposta sobre as operadoras de telefonia móvel (MNO, do inglês *mobile network operators*) tem despertado várias iniciativas de pesquisa a fim de elaborar soluções para contornar o congestionamento de tráfego de dados e a sobrecarga da infraestrutura central e redes de acesso via rádio (RAN, do inglês *radio access networks*) das MNOs. Dentre as várias técnicas imaginadas para combater essa questão, o *escoamento de dados* tem surgido como uma solução viável e efetiva para aliviar esse problema. Em geral, o escoamento de dados é compreendido como qualquer mecanismo que desvie parte do tráfego originalmente direcionado às redes celulares para outras tecnologias sem fio alternativas. Para isso, muitas técnicas têm sido propostas, e tem sido mostrado que, não somente um escoamento de tráfego significativo pode ser atingido, mas também uma maior vazão agregada, extensão de cobertura da rede, melhor eficiência energética, e redução no tempo de entrega do conteúdo [7, 8, 9].

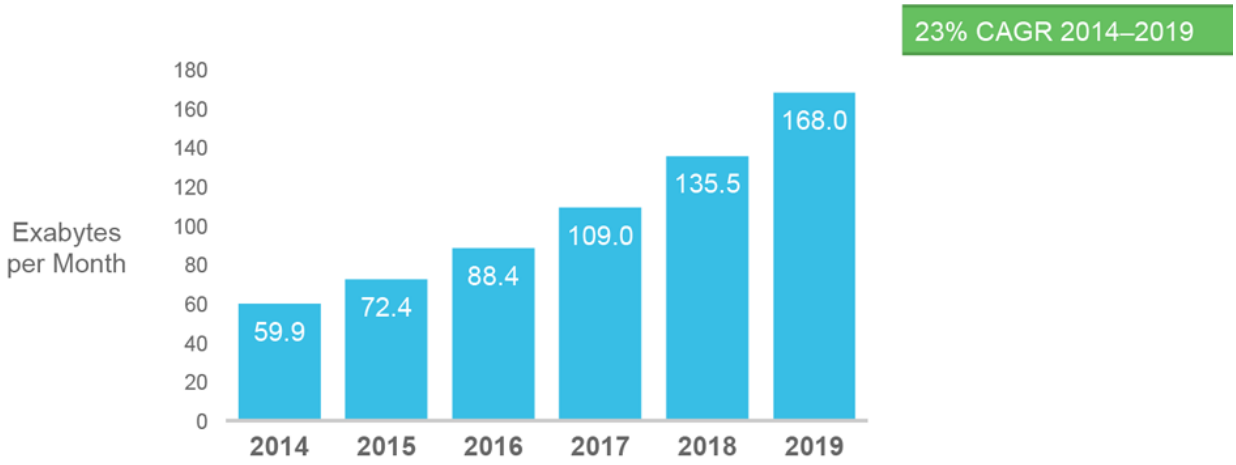

Figure 1.1: Previsão de crescimento de tráfego feita pela CISCO para os anos de 2014 a 2019 [1]

Dentre as várias soluções que têm sido propostas até o momento, diversos trabalhos têm defendido o paradigma das comunicações dispositivo-a-dispositivo (D2D)[1] para implementar o escoamento de dados das redes de telefonia móvel (veja [10, 11, 12, 13] e suas referências). A Figura 1.2 ilustra algumas aplicações possíveis das redes D2D, como a disseminação de conteúdo, jogos, retransmissão de dados dos usuários, e entre essas aplicações também se encontra o escoamento de tráfego de dados móveis. Em todos esses trabalhos sobre escoamento de dados via redes D2D, entretanto, a participação voluntária do usuário é geralmente tratada como algo *certo*, ainda que o compartilhamento dos recursos do cliente seja necessário para que o escoamento de dados funcione (ex: energia, memória, largura de banda, etc.). Na prática, alguém pode esperar resistência significativa do cliente para repassar o tráfego de alguém sem receber quaisquer incentivos. Mas, conforme observado corretamente por Rebecchi et al. [13], a colaboração de usuários é essencial para a implementação do escoamento de dados dispositivo-a-dispositivo. Mesmo assim, a questão de *como incentivar usuários para colaborar ativamente no escoamento de tráfego de redes móveis* não tem recebido muita atenção na literatura. Note que, esta questão é ligeiramente diferente de simplesmente incentivar clientes tolerantes a atrasos a recuperarem seus dados de outras redes não-celulares em instantes futuros [14]. Recentemente, alguns trabalhos começaram a investigar mecanismos de incentivo para que as MNOs aluguem pontos de acesso (APs, do inglês *access points*) de terceiros para fins de escoamento de tráfego [15, 3], como mostrado na Figura 1.3. Mas, até o momento, nenhum trabalho olhou para a questão de se incentivar os próprios clientes a participarem do escoamento da infraestrutura (que pode funcionar como um segundo passo do escoamento após a negociação anterior de locação entre MNO/AP).

Alguns anos atrás, o desafio MANIAC (*Mobile Ad Hoc Networking Interoperability and Cooperation Challenge*) de 2013 [16] apresentou um problema de escoamento de dados baseado em um cenário onde os pontos de acesso (APs) da operadora escoariam pacotes para os dispositivos dos clientes, que, em troca, teriam a tarefa de entregar os pacotes para APs de destino indicados por APs fonte. Para realizar isso, um esquema de *leilões recursivos* deveria ser aplicado em cima de uma rede *ad hoc* formada pelos dispositivos dos usuários, ou seja, "licitações" salto-a-salto decidiriam o caminho de cada pacote de dados em direção ao seu destino. Neste cenário, cada pacote anunciado por um AP fonte estaria associado a um *orçamento máximo* (para fins de pagamento) e uma *multa* a ser paga no caso em que o pacote não fosse entregue dentro do *prazo* anunciado (traduzido em um número máximo de saltos para o destino). O AP fonte sempre selecionaria o próximo salto com o menor valor oferecido por um nó, enquanto que todos os outros dispositivos escolheriam o nó seguinte baseado na *sua própria estratégia*, desde que respeitando um conjunto de regras. Durante a disputa do MANIAC 2013, cada time foi autorizado a usar somente dois tablets executando Android, o protocolo de roteamento OLSR (Optimized Link State Routing [17], com base no qual os nós poderiam implementar suas próprias estratégias de encaminhamento), e a API de desenvolvimento do MANIAC [18]. Muito embora a realização do MANIAC Challenge 2013 tenha permitido a obtenção de informações preliminares importantes, seu formato, duração, e limitações físicas—adicionados ao número limitado de participantes e dispositivos—não permitiram uma profunda avaliação de desempenho das estratégias concorrentes. Dificultando ainda mais a

[1] Nessa dissertação, o termo "comunicações device-to-device" é usado em um sentido amplo, ou seja, não é específico à definição do 3GPP.

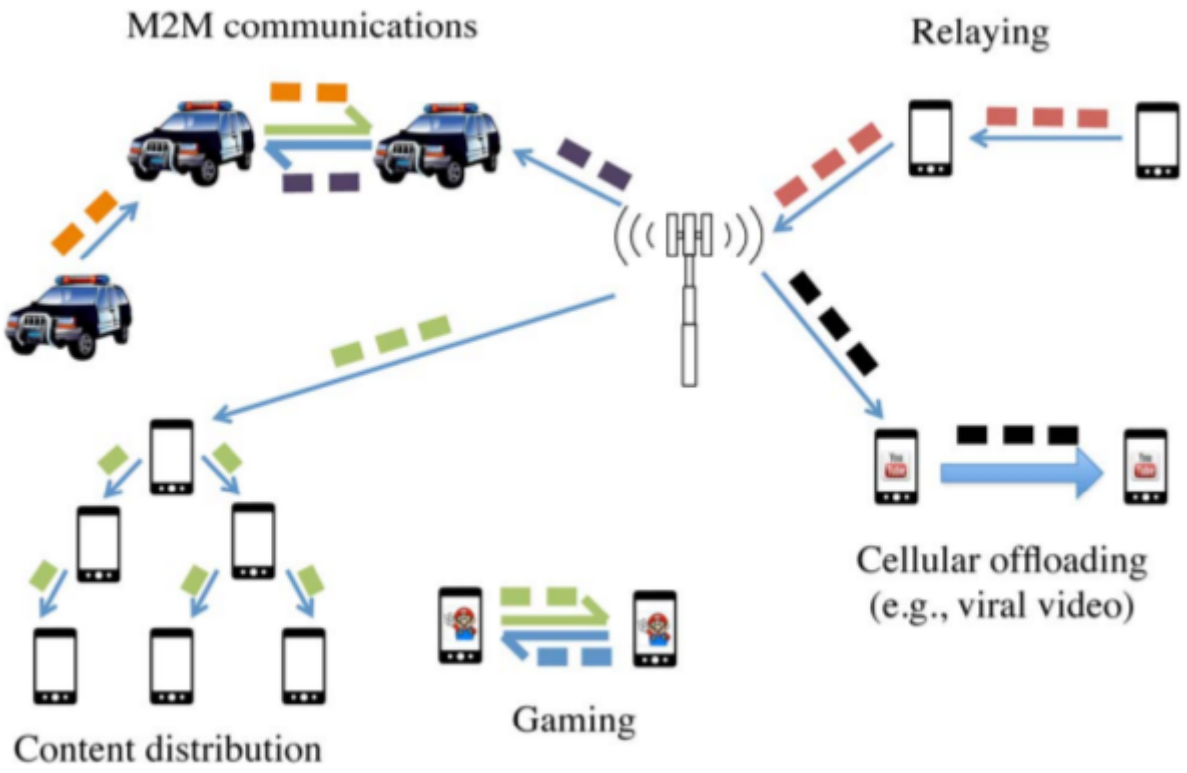

Figure 1.2: Exemplos de implementação de escoamento de dados via comunicações D2D [2]

situação, diferentes estratégias atuaram na rede concomitantemente, o que tornou muito difícil entender as suas interações, e o impacto que as estratégias poderiam ter umas nas outras. Mais ainda, dependendo da localização e movimentação dos membros da equipe, nem todas estratégias estavam competindo contra todas as outras em todos leilões, devido à topologia da rede. Assim, para cada leilão executado na rede, poderiam existir diferentes conjuntos de competidores e leiloeiros, resultando em diferentes comportamentos e resultados. Em outras palavras, esse foi um ambiente extremamente heterogêneo e complexo, cujas características tornaram-no bastante difícil de entender o desempenho e o verdadeiro potencial de qualquer estratégia.

Baseado nessas observações, essa dissertação apresenta uma avaliação de desempenho da então chamada *Estratégia do Aperto*, projetada para o desafio MANIAC de 2013 [18, 19]. O termo "aperto" refere-se ao fato que a estratégia utiliza uma estimativa do quão "apertado" um nó está com relação ao cumprimento da tarefa de entregar o pacote para o seu destino alvo *dentro do prazo estipulado*. Na verdade, o conceito de "aperto" foi concebido com o objetivo de deixar algum "espaço" (com respeito ao prazo) para absorver eventuais más decisões de encaminhamento feitas por outros nós ao longo do caminho, de tal forma que a chance de entrega de pacote aumente (dentro do prazo). A estratégia é composta por três sub-estratégias: a estratégia de *construção do lance para um leilão*, a estratégia de *definição do preço máximo a ser pago e multa associados ao pacote a ser leiloado*, e a estratégia de *tomada de decisão com relação ao vencedor do leilão anunciado*. O conceito de *aperto* é usado em ambas as estratégias de *definição de lance* e de *tomada de decisão*.

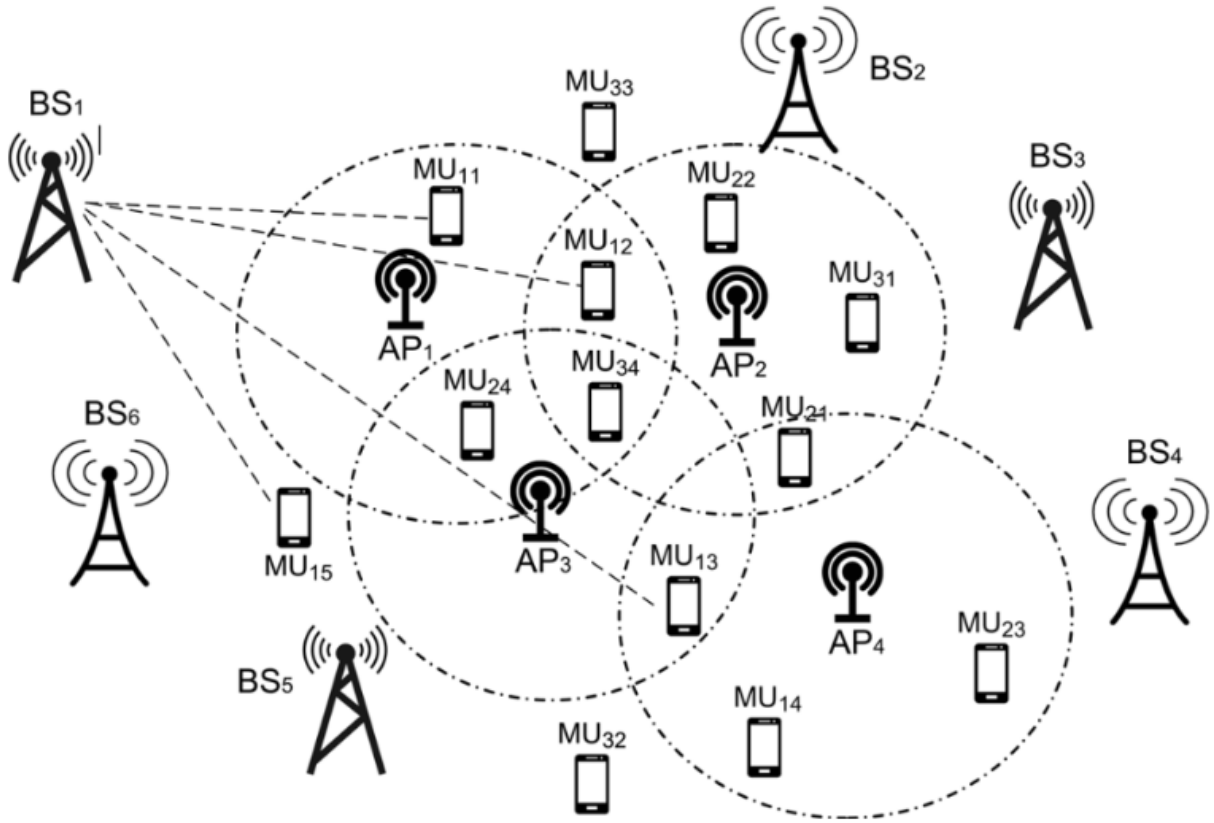

Figure 1.3: Exemplo de aluguel de APs pelas operadoras (MNOs) para escoamento de dados. Como mostrado nesta figura, olhamos para o caso geral onde cada AP pode servir mais de uma MNO, e cada MNO possui várias estações de base (BSs) e podem alugar múltiplos APs em diferentes locais para escoar o tráfego dos seus usuários (APs são sobrepostos) [3].

Nessa dissertação, também estendemos a estratégia original [19] ao introduzir uma segunda *função de preferência* para a sub-estratégia de tomada de decisão (um nó utiliza a função de preferência para determinar quem vence seus leilões). Dois pontos de operação particulares são escolhidos para essa segunda função de preferência, que nos permite investigar o desempenho de três variantes da *Estratégia do Aperto* (incluindo a estratégia original).

Para efeito de comparação, duas estratégias básicas também são investigadas: uma que prioriza a entrega de pacotes em detrimento dos ganhos (em créditos, unidades monetárias, etc) possíveis (ao aplicar roteamento de menor caminho independente dos lances dos nós), e uma gananciosa, onde os nós sempre escolhem o maior lance independente da sua chance de entregar o pacote dentro do prazo. Essa dissertação estuda a implementação *homogênea* de cada estratégia, ou seja, quando todos os nós da rede D2D implementam a mesma estratégia (exceto os APs). Uma implementação homogênea é o primeiro passo para a compreensão do desempenho alcançável de cada estratégia, ao contrário de uma heterogênea (pelas razões descritas anteriormente). Também, com uma implementação homogênea, pode-se não somente avaliar os ganhos médios possíveis de créditos (ou unidades monetárias) dos nós na rede D2D, mas também as características de justiça da estratégia na distribuição de ganhos entre os nós. A avaliação de desempenho é baseada em

simulações a eventos discretos realizadas com o simulador ns-3 [20], e estudamos ambos os cenários estático e móvel sob diferentes velocidades dos nós.

Todas as estratégias são avaliadas com respeito à *taxa de entrega de pacotes*, *ganho (lucro) médio por nó*, *justiça na distribuição dos lucros*, e *número médio de saltos* necessários para um pacote alcançar seu destino. Até onde sabemos, este é o primeiro trabalho a avaliar o desempenho de uma estratégia para leilões recursivos em comunicações dispositivo-a-dispositivo para escoamento de dados sob restrição de prazo.

1.1 Objetivos da Dissertação

Esta dissertação teve como objetivos específicos:

- Avaliar e entender o comportamento da estratégia do aperto para escoamento de tráfego via comunicações dispositivo-a-dispositivo;
- Avaliar o desempenho da implementação homogênea da estratégia do aperto, ou seja, quando todos os nós da rede executam a mesma estratégia;
- Avaliar o impacto da velocidade dos nós no desempenho da estratégia do aperto;
- Avaliar o desempenho da estratégia do aperto quando implementada em redes maiores do que aquelas estudadas no MANIAC Challenge;
- Comparar o desempenho da estratégia do aperto frente a outras estratégias básicas. Essas estratégias básicas geram lances de valores aleatórios e não usam de inteligência para otimizar a escolha entre a economia e a garantia na entrega de dados;
- Explorar o desempenho de outras sub-estratégias para a estratégia do aperto, em particular, de variações da sub-estratégia de tomada de decisão.

1.2 Contribuições

As principais contribuições deste trabalho são listadas a seguir:

- Implementação no ns-3 do cenário de escoamento de tráfego proposto no MANIAC Challenge, para fins de estudo do problema de leilões recursivos via comunicações máquina-a-máquina;
- Implementação no simulador ns-3 de um arcabouço que implementa outras estratégias de participação de leilões recursivos;
- Avaliação de desempenho da estratégia do aperto e variantes, assim como duas estratégias básicas, segundo a *taxa de entrega de pacotes*, *ganho (lucro) médio por nó*, *justiça na distribuição dos lucros*, e *número médio de saltos* necessários para um pacote alcançar seu destino; As avaliações apresentadas consideram os cenários estático e móvel, sob diferentes velocidades;

- Segundo levantamento bibliográfico extenso, esta é a primeira avaliação de desempenho de estratégias para leilões recursivos aplicadas ao problema de escoamento de tráfego via comunicações máquina-a-máquina sob restrições de prazo.

1.3 Organização da Dissertação

O restante da dissertação está dividido conforme a seguir. O Capítulo 3 descreve trabalhos relacionados na literatura. O Capítulo 4 descreve o cenário de rede para escoamento de tráfego e as regras dos leilões recursivos e apresenta as estratégias avaliadas nesta dissertação, começando com a própria *Estratégia do Aperto*. O Capítulo 5 descreve os cenários de simulação e explica a estrutura do simulador e do arcabouço implementado, e o Capítulo 6 apresenta os resultados de simulação. Finalmente, o Capítulo 7 contém as conclusões deste trabalho com uma discussão sobre possíveis trabalhos futuros.

Chapter 2

Introduction

According to a number of industry reports [1, 6], the explosive growth of mobile data traffic we have witnessed so far is expected to continue in the years to come, especially driven by a multitude of machine-to-machine communications and video-related content (see Figure 2.1). This unprecedented traffic demand imposed on mobile network operators (MNOs) has ignited a number of research initiatives to devise solutions to circumvent traffic congestion and overload of the MNO's core infrastructure and radio access networks (RANs). Among the various techniques envisioned to tackle this issue, *mobile data offloading* has emerged as a viable and effective solution to alleviate this problem. In general, mobile data offloading is understood as any mechanism that deviates part of the traffic originally targeted for macrocellular networks to alternative wireless technologies. For that, many techniques have been proposed, and it has been shown that, not only significant traffic offloading can be achieved, but also higher aggregate throughput, extended network coverage, better energy efficiency, and reduction of content delivery time [7, 8, 9].

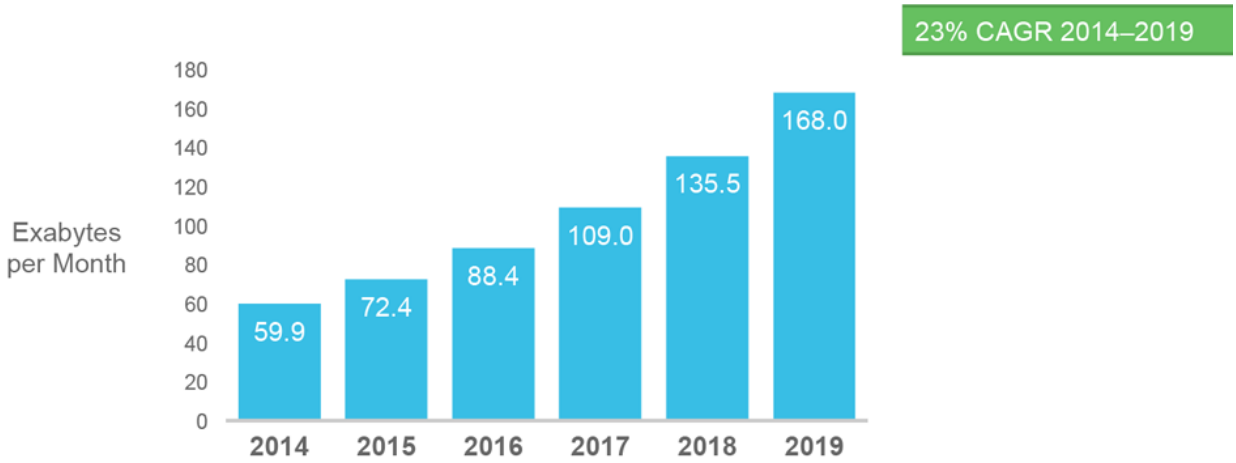

Figure 2.1: Traffic growth forecast made by CISCO for the years of 2014 to 2019 [1]

Among the many solutions that have been proposed so far, a number of works have advocated

the paradigm of device-to-device (D2D)[1] communications to implement mobile data offloading (see [10, 11, 12, 13] and references therein). The Figure 2.2 illustrates some possible applications of the D2D networks, like content dissemination, gaming, user data relaying, and among these applications also is the mobile data traffic offloading. In all these works about data offloading via D2D networks, however, user participation is generally treated as a *given*, even though the sharing of the client's resources is needed for the data offloading to work (e.g., energy, memory, bandwidth, etc.). In practice, one should expect significant client resistance to relay someone's traffic without receiving any incentives. But, as correctly pointed out by Rebecchi et al. [13], user collaboration is key for deployment of device-to-device data offloading. Nevertheless, the issue of *how to incentivize users to actively collaborate in the offloading infrastructure* has not received much attention in the literature. Note that, this is slightly different from simply incentivizing delay-tolerant clients to retrieve their data from other non-cellular networks [14]. Recently, some works have started looking at incentive mechanisms for MNOs to lease third-party access points (APs) for offloading purposes [15, 3], as showed in Figure 2.3. But, so far, no work has looked at the issue of incentivizing the clients themselves to participate in the offloading infrastructure (which could work as a second offloading step after previous MNO/AP leasing negotiation).

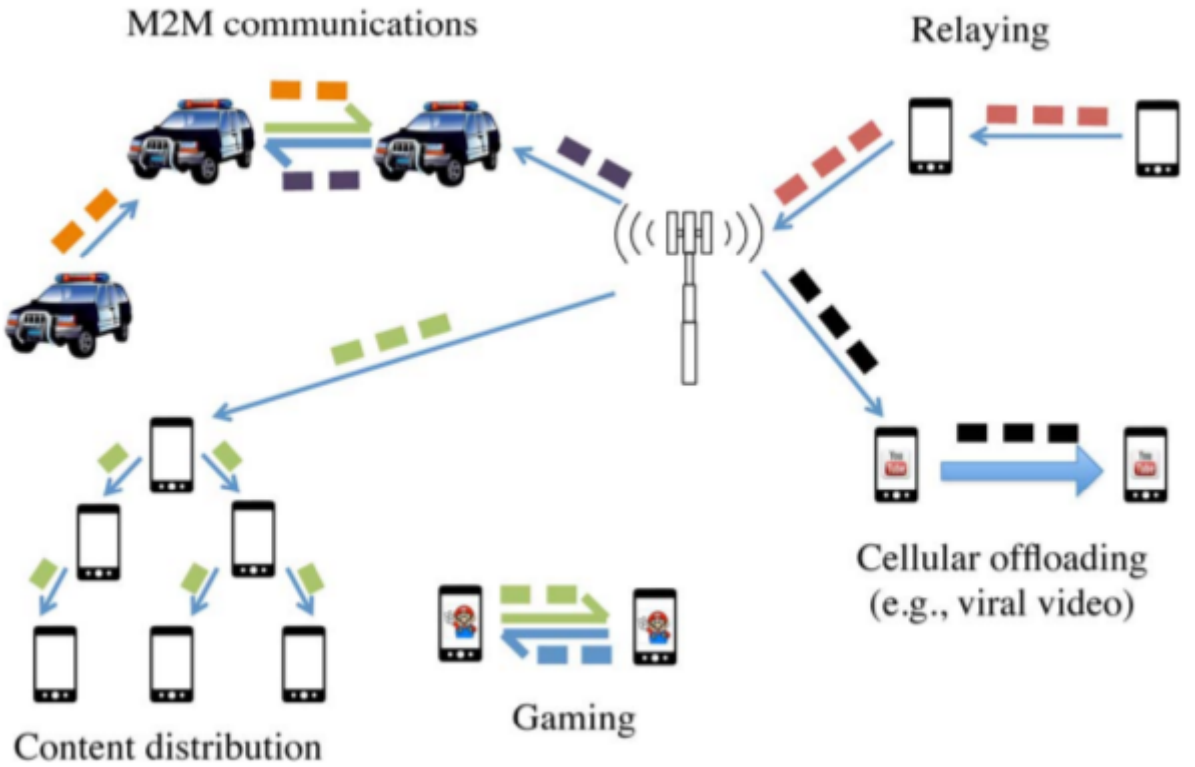

Figure 2.2: Examples of data offloading via D2D communications implementation [2]

A few years ago, the *Mobile Ad Hoc Networking Interoperability and Cooperation (MANIAC) Challenge* 2013 [16] posed a data offloading problem based on a scenario where the operator's access

[1]In this dissertation, the term "device-to-device communications" is used in a broad sense, i.e., it is not specific to the 3GPP definition.

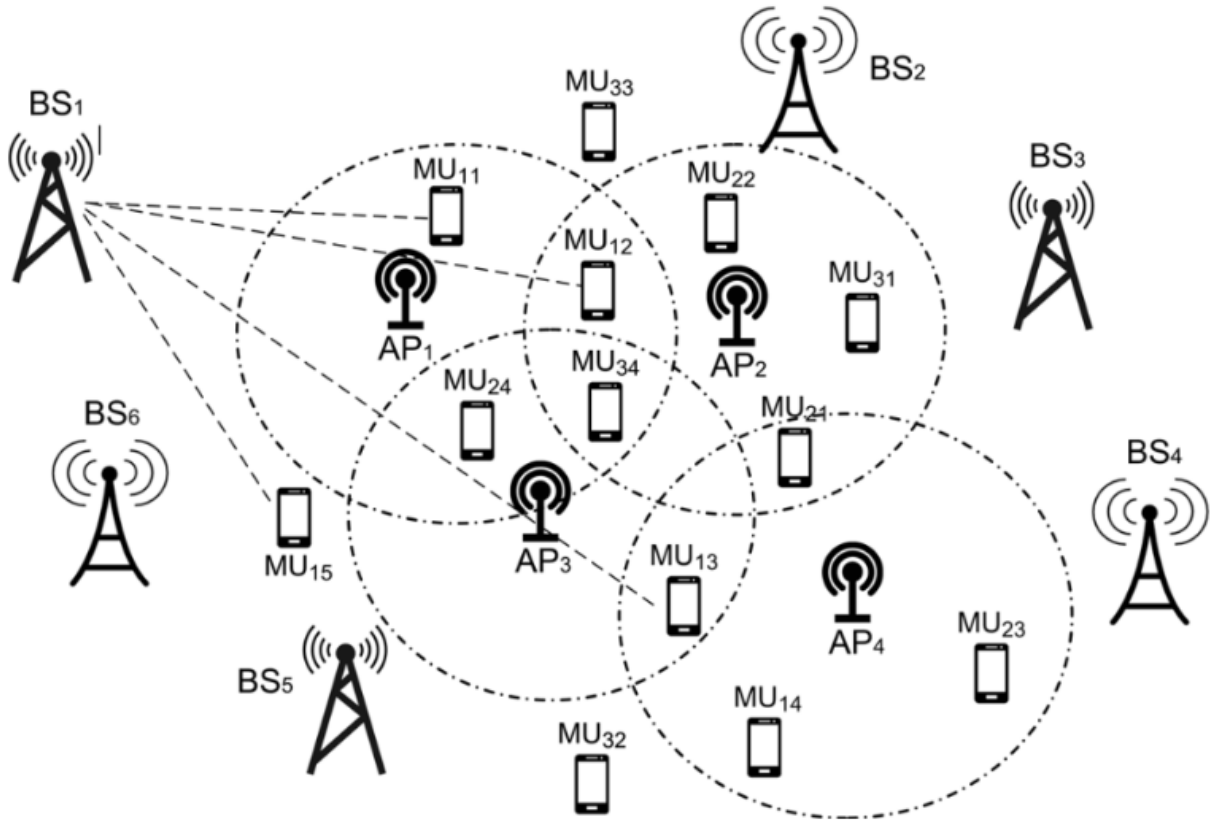

Figure 2.3: Example of the leasing of APs by operators (MNOs) for data offloading. As shown in this figure, we look at the general case where each AP can serve more than one MNO, and each MNO owns several base stations (BSs) and may lease multiple APs at different locations to offload the traffic of its users (APs are overlapping) [3].

points (APs) would offload data packets to clients' devices, which, in turn, would have the job to deliver these packets to destination APs indicated by the source APs. The incentive for customers would be discounted monthly fees, and the incentive for operators would be decreased infrastructure costs. To accomplish that, a scheme of *recursive auctions* should be deployed on top of an ad hoc network formed by the clients' devices, i.e., a hop-by-hop bidding contest would decide the path of each packet towards its destination. In this scenario, each packet announced by a source AP would be associated to a *maximum budget* (for payment purposes), and a *fine* to be paid in case the packet was not delivered within the announced *deadline* (translated into a maximum number of hops to destination). The source AP would always select the next hop with the lowest bid, while all other devices would choose a downstream node based on *their own strategy*, as long as they abided to a set of rules. During the contest, each team was allowed to use only two tablets running Android, the OLSR (Optimized Link State Routing) routing protocol [17] (based on which the nodes could implement their own forwarding ideas), and the MANIAC framework API [18]. In spite of the invaluable preliminary insights resulted from the MANIAC Challenge 2013, its format, time, and physical constraints—coupled with a limited number of devices and participants—did not allow an in-depth performance evaluation of the competing strategies. To complicate matters, different

strategies were in place concurrently, which made it very hard to understand their interactions, and the impact that each strategy could have on one another. Moreover, depending on the location and movement of team members, not all strategies were competing against each other in every auction, as a result of network topology. Therefore, for every single auction in the network, there could exist different sets of competitors and auctioneers, resulting in different behaviors and outcomes. In other words, this was a highly heterogeneous and complex environment, whose characteristics made it very difficult to understand the performance and true potential of any single strategy.

Based on these observations, this dissertation presents a performance evaluation of the so-called *Tightness* strategy designed for the MANIAC Challenge 2013 [18, 19]. The *tightness* term refers to the fact that it uses an estimate of how "tight" a node is with respect to fulfilling the job of delivering a packet to its target destination *within the given deadline*. In fact, the "tightness" concept was designed with the goal of making some "room" (with respect to the deadline) to absorb eventual bad forwarding decisions made by other downstream nodes as a result of their own auctions, so that the likelihood of packet delivery is increased (within the deadline). The strategy comprises three sub-strategies: the *bidding* strategy, the *budget-and-fine* setup strategy, and the *decision-making* strategy. The tightness concept is used in both *bidding* and *decision-making* sub-strategies. In this dissertation, we also augment the original strategy [19] by introducing a second *preference function* for the decision-making sub-strategy (a node uses the preference function to determine who wins its auctions). Two particular operating points are chosen for this second preference function, which allows us to investigate the performance of three variants of the *Tightness* strategy (including the original one).

For comparison purposes, two baseline strategies are also investigated: one that prioritizes packet delivery over budget gains (by applying shortest-path routing regardless of nodes' bids), and a greedy one, where the nodes always choose the highest bid regardless of its likelihood of delivering the packet within the deadline. This dissertation studies the *homogeneous* deployment of each strategy, i.e., when all nodes in the D2D network implement the same strategy (apart from the APs). A homogeneous deployment is the first step towards understanding the achievable performance of each strategy, as opposed to a heterogeneous one (for the reasons described previously). Also, with a homogeneous deployment, one can better assess not only the average budget gains of nodes in the D2D network, but also the fairness characteristics of the strategy in budget distribution among nodes. The performance evaluation is based on discrete-event simulations carried out with the ns-3 simulator [20], and we study both static and mobile scenarios under different node speeds. All strategies are evaluated with respect to packet delivery ratio, average budget per node, budget fairness, and average number of hops needed for a packet to reach its destination. To the best of our knowledge, this is the first work to evaluate the performance of a strategy for recursive auctions in device-to-device communications for data offloading under deadline constraints.

2.1 Dissertation Objectives

This dissertation had the following objectives:

- Evaluate and understand the behaviour of tightness strategy for traffic offloading via device-to-device communications;
- Evaluate the homogeneous implementation performance of the tightness strategy, i.e., when all network nodes execute the same strategy;
- Evaluate the nodes speed impact in the tightness strategy performance;
- Evaluate the tightness strategy performance when implemented in networks bigger than those studied in the MANIAC Challenge;
- Compare tightness strategy performance against other basic strategies. These basic strategies generate bids with random values and does not use intelligence to optimize the choice between economy and the data delivery guarantee;
- Explore the performance of other sub-strategies for the tightness strategy, in particular, of variations of the decision making sub-strategy.

2.2 Contributions

The main contributions of this work are listed as follows:

- Implementation in the ns-3 simulator of the traffic offloading scenario proposed in the 2013 MANIAC Challenge, for study of recursive auction strategies via device-to-device communications;
- Implementation in the ns-3 simulator of a framework that implements other strategies for participation in recursive auctions;
- Performance evaluation of the "Tightness Strategy" its variants, as well as two baseline strategies, based on *packet delivery ratio*, *average budget per node*, *budget fairness*, and *average number of hops*; The assessments presented considers the static and mobile scenarios, under different speeds;
- Based on an extensive bibliographic survey, this is the first work to evaluate the performance of a strategy for recursive auctions in a device-to-device communications for data offloading under deadline constraints.

2.3 Dissertation Organization

The rest of the Dissertation is divided as follows. Chapter 3 describes related works in the literature. Chapter 4 describes the network scenario and auction rules and presents the strategies evaluated in this dissertation, starting with the *Tightness* strategy itself. Chapter 5 contains the simulation scenarios and explains the simulator structure and the framework deployed, and the

Chapter 6 presents the simulation results. Finally, Chapter 7 contains the conclusions of this work with a discussion of possible future works.

Chapter 3

Related Work

One of the key issues to fully realize multi-hop terminal-to-terminal (or D2D) mobile data offloading is how to incentivize users to let their devices work as packet relays to the benefit of others. Buttyán and Hubaux [21] were one of the first to tackle this problem by introducing a virtual currency named *nuglets*, with which nodes could pay other nodes to forward their packets. According to their *packet purse model*, a source node needs to load a packet with sufficient *nuglets* to reach its destination. Each forwarding node gets some *nuglets* from the packet in order to cover its forwarding costs. A packet is discarded if it does not contain enough *nuglets* to be forwarded. To control the number of *nuglets* taken out from a packet, a *sealed bid second price auction* is run at each hop: each bidder determines the price for which it is willing to forward the packet, and sends it to the forwarding node in a sealed form. The price is obtained from two utility functions that are based on battery level and number of *nuglets* in the node. It is assumed that a bidding node has no information about the total number of bidders participating in the auction, and the auction winner is always the one with the lowest bid. Then, the forwarding node puts the value of the *second lowest bid* in the packet and sends it to the winner. For proper operation, this scheme requires the use of routing algorithms that allow nodes to have multiple entries in their routing tables with different next hops to the same destination (e.g., TORA [22]). The performance evaluation of the proposed approach has focused on packet delivery ratio for different battery energy levels over static topologies. Also, in spite of targetting cooperation, the fairness in *nuglets* distribution among nodes has not been evaluated, and no indication was given about the achievable average gain of *nuglets* per node. Finally, this solution did not target packet delivery within a given deadline.

Anderegg and Eidenbenz [23] have proposed Ad hoc-VCG, a reactive routing protocol for ad hoc networks where nodes are selfish and require payments for data forwarding. The protocol is designed to achieve truthfulness (i.e., nodes reveal the true costs to forward data) and cost efficiency (energy required to relay a packet to a given neighbor). It consists of two phases: route discovery and data transmission. During route discovery, a minimum energy route is computed from source to destination based on a weighted graph informed to the destination node. The edge weights represent the payments a node has to receive if it transmits a packet along that edge. The *destination node* computes the shortest path to it and all payments needed to be made. Then, it sends this information back to the source node. In the data transmission phase, the source

node sends the data packets with the electronic payments over the shortest path. For analysis, this work has focused on proving truthfulness and cost efficiency mathematically, and provided some results on experiments with random static topologies to analyze overpayment. However, the protocol was not analyzed under mobility, and its performance was not evaluated regarding packet delivery ratio and individual node profit gains (as well as fairness in profit distribution). Similar to [21], the protocol does not consider packet delivery under a given deadline, and its route discovery phase may stall network operation if routing paths change frequently, as pointed by the authors themselves.

Luo et al. [24] have introduced the Unified Cellular and Ad Hoc Network (UCAN), one of the first ideas to use device-to-device communications coupled with cellular networks with the goal of improving the overall cell throughput (without focusing on cellular offloading). If a node experiences low cellular downlink data rates, the base station transmits its data frames to another client (the proxy client) with better channel conditions. Then, these frames are further relayed through IP tunneling via intermediate clients over Wi-Fi links. Proxy discovery and maintenance protocols were also proposed, which are used for routing purposes, too. Traffic scheduling at the base station is performed by taking into account the destination client's downlink data rate. By using this metric—instead of the proxy's data rate—it is shown that throughput balance can be achieved between destination and proxy clients, while improving overall cell throughput. This fact is used to incentivize clients to participate in the UCAN architecture, since their own data rates can be improved. Finally, a secure crediting mechanism is proposed by which the base station can keep track of the number of data frames relayed by each client (they focus on the security mechanism, and suggest the use of a crediting scheme such Buttyán's [21]). This work has clearly showed the advantages of coupling cellular with D2D communications to improve overall network performance. But, because it did not targeted the data offloading problem, the issue of data delivery within a given deadline was not a concern.

The idea of using device-to-device communications to support data offloading has been investigated in a number of papers. In particular, delay-tolerant networks (DTNs) (or opportunistic networks) have been considered as a mean to offload delay-tolerant traffic via content dissemination ("epidemically") by exploiting the mobility of nodes. Therefore, a key question in such a problem is how to choose the set of *seeders* to first bootstrap and initiate the distribution process. Han et al. [10] have studied the target-set selection problem with a fixed number of k users, such that the expected number of infected users is maximized. Along the same lines, Li et al. [11] have investigated the optimal target-set selection by proposing a utility maximization problem under multiple linear constraints. In both cases, the network provider needs to gather information about the nodes' contact rates (or contact statistics) in order to compute the best subset of seeders. Whitbeck et al. [12] have also proposed epidemic content dissemination with their Push-and-Track framework. The novelty of their approach is the introduction of a closed-loop control to supervise the need of content re-injection in the network to guarantee full dissemination within some target delay. In all these content dissemination approaches, it has been assumed that all participant nodes are interested in the same content. However, the question of having users forwarding content they are not interested in is left as an open problem. So, some incentive mechanism is needed for

forwarding unwanted traffic.

As far as incentive mechanisms for data offloading are concerned, Zhuo et al. have presented Win-Coupon [14], an incentive framework based on reverse auctions to motivate users to leverage their delay tolerance for 3G data offloading. This was arguably the first proposal to apply auction mechanisms for 3G offloading. In this proposal, users receive discounts ("coupons") for their service charges if they are willing to wait longer for data downloading. During the delay, part of the 3G traffic may be opportunistically offloaded through complementary networks (e.g., DTNs, WiFi hotspots or femtocells). When the delay is over, the remaining part of the data is received via the 3G connection. To receive the coupons, the users send bids to the operator containing the delay they are willing to experience and the discount they want to obtain. The operator acts as a buyer, and decides the optimal auction outcome based on delay tolerance and offloading potential of users to achieve the minimum incentive cost for a given offloading target. Although this work has not addressed incentives mechanisms for D2D offloading via recursive auctions, DTN was employed as a case study for the complementary network. In this case, nodes were assumed to retrieve the desired data with high probability when approaching any node in the network (single-hop transfers).

Yu et al. [25] have proposed INDAPSON, a system that promotes cooperative downloading of cellular traffic by allowing users with a surplus of cellular data traffic to assist others with a shortage of it. In their approach, a user who needs to download data ("primary user") forms a local wireless network in which "assistant users" are chosen to help download data from the Internet through their cellular connections. Then, data segments are relayed to the primary user through WiFi connections, and network management is done via the Bluetooth interface. In order to incentivize users to assist others, a *reputation adaptive pricing* (RAP) scheme is applied, where a primary user pays virtual credits to assistant users, which, in turn, can use these credits to purchase the help of others in future downloads. A centralized virtual credit account is maintained on a server, who keeps track of records of download and relay history of each user. In spite of not addressing the data offloading problem (it actually aims higher download data rates with simultaneous cellular connections), the issue of user reputation is explicitly included in the price formulation. This is certainly another dimension to consider in the realm of D2D data offloading under recursive auctions. The Tightness strategy proposed in this paper allows the incorporation of similar reputation mechanisms through its preference functions. To accomplish that, the nodes would have to maintain a forwarding (and packet delivery) history of participant nodes, but with the added advantage of not relying on a central authority to manage that.

Gao et al. [15] have studied the incentive design problem in network-initiated mobile data offloading through third-party APs (WiFi or femtocell). In this context, the operators propose the price they are willing to pay to each AP for offloading its traffic. Then, each AP decides how much traffic they will offload for each BS. This is studied as a two-stage multi-leader multi-follower game. The authors characterize the Nash equilibrium of this game and compare it with the outcomes of perfect competition market (no price participation) and monopoly market (no price competition), and they show that the equilibrium price is upper-bounded by the market clearing price (that arises in the equilibrium of perfect competition market), and lower-bounded by the monopoly price (no

price competition). In their work, they assume that mobile users (MUs) either comply or are properly incentivized in a way that they offload traffic exactly as the networks intend, or they are unaware of the offloading process, i.e., the data offloading is totally transparent to MUs. Therefore, the incentive for MUs to participate in the offloading process is not treated in their work. Along the same lines, Iosifidis et al. [3] have also studied an offloading market where a set of MNOs compete to lease a set of APs for the offloading service. In this case, the marketplace is managed by a centralized broker, who collects the MNOs' requests and the APs' offers and determines how much traffic of each MNO will be offloaded to each AP and at what price. The MNOs and APs accept the broker's decision if they find it advantageous. The broker is rewarded based on volume of transactions, so that it does not deviate from the desirable market goal of maximizing market efficiency. Then, an iterative double-auction scheme is employed that is proven to be efficient (i.e., it maximizes the welfare), it is weakly budget-balanced (broker does not lose money), and individually rational (MNOs and APs are willing to participate). In this architecture, there is no direct participation of the mobile users.

Recently, Koutsopoulos et al. [26] have considered multi-hop D2D communications where social network ties are leveraged by MNOs to achieve efficient data transport. Their work is based on the D2D paradigm of 4G+ technologies, where end-to-end path formation and resource allocation (e.g., spectrum management) are controled by the operator. However, data forwarding decisions are left to the user, whose willingness to relay data packets (i.e., the user benefit) is related to the strength of his social ties. The problem is modeled as a constrained minimum-cost problem on the communication graph, where the constraints arise from the delivery probability derived from the social network graph. We note that, although social ties may increase one's willingness to forward the packets, a scheme for actual compensation (e.g., discounts, credits, etc.) should suplement this approach, since regardless of its social ties, all users will have to donate part of their precious resources (battery life, storage, bandwidth, etc.) to the task of data offloading.

As we can see, none of previous works have dealt with the problem of employing recursive auctions for purposes of data offloading via device-to-device commmunications under deadline constraints. In fact, such a technique could be thought as a second step towards extending data offloading, for example, after MNOs lease AP providers to carry out data offloading, as discussed previously [15, 3].

Chapter 4

Auction Participation Strategies

In this chapter, we first present a more detailed explanation of the data offloading scenario, as well as the rules for the recursive auctions to happen. Then, we present the strategies we investigate for devices to participate in the recursive auctions. Each strategy comprises three sub-strategies: the *bidding* strategy, which defines how to set the value of a bid for a given RFB received from a neighboring node, the *budget-and-fine* setup strategy, that defines how a node, who just won an auction, sets the budget and fine values of its own RFB, and the *decision-making* strategy, which defines how an auctioneer picks the winner of its announced RFB. Each of these sub-strategies can be specified in different ways, according to different goals. The first strategy we introduce is actually a *class* of strategies that is based on the central idea of "tightness" with respect to packet delivery within a given deadline, i.e., how much "room" a node has (before reaching the deadline) to absorb eventual bad forwarding decisions resulted from the unpredictable outcomes of other downstream auctions. Therefore, we actually present a set of "Tightness Strategies," and we differentiate them in this dissertation with respect to the *preference function* used in the *decision-making* strategy (the other two sub-strategies are kept the same, for the sake of evaluation). The idea of the tightness strategy was introduced previously [19]. For completeness, we reproduce the main ideas in this dissertation, followed by presentation of the other two variations in the *decision-making* sub-strategy. Then, two other strategies are introduced for purpose of performance evaluation and comparision. These two strategies do not make use of the "tightness" concept, and they differ with respect to the *bidding* and *decision-making* strategies.

4.1 Network Scenario and Auction Rules

Figure 4.1 depicts the scenario considered in this dissertation for data offloading through D2D communications using recursive auctions. The lines connecting the devices indicate wireless connectivity, and the arrows indicate a path that a packet might take to traverse the D2D network from a source access point (AP) to a destination AP after recursive auctions are performed by the nodes themselves.

A backbone AP initiates a forwarding request by broadcasting a special packet that we name it

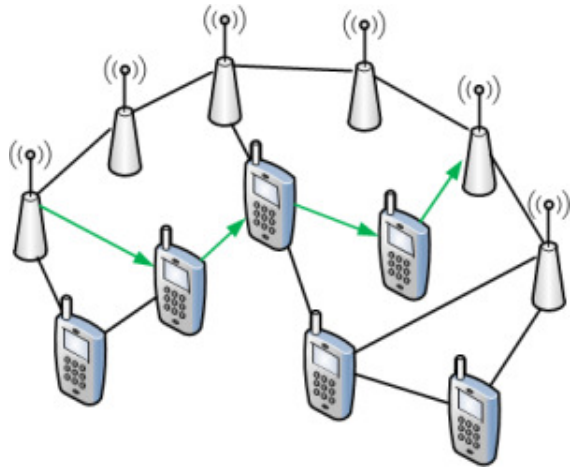

Figure 4.1: Data offloading scenario via D2D communications and recursive auctions. The nodes relay packets from one source AP to a destination AP.

"request for bids" (RFB). The RFB announced by a source AP contains the following information:

- Maximum *budget* B_0 available for successful delivery of a given data packet to its destination AP;
- Deadline for packet delivery, translated into a maximum *number of hops* H_0 allowed for a packet to traverse the D2D network before reaching the destination AP, beyond which it is declared *unsuccessful*; It is assumed that the operator computes this quantity;
- A *fine* F_0 to be paid if the packet is not delivered to destination within the "deadline" of H_0 hops.

Every AP neighbor node that receives the RFB must participate in the auction by making a bid with the sending of a *bid* packet. After waiting for a time interval t_0 long enough to receive the neighbors' bids, the source AP decides for the auction winner by *always* choosing the node with the *lowest bid* $b_i \leq B_0$, $\forall i \in \mathcal{B}$, where $\mathcal{B}$ is the set of nodes whose bids were received by the auctioneer within the time interval t_0. Only then, the source AP forwards the packet to the auction winner. From this point on, per packet hop-by-hop recursive auctions are performed in order to forward each data packet towards the destination AP. Each device has the freedom to deliver the packet to the destination AP either via the D2D network or the provider's infrastructure backbone of APs (if within range, of course). Using any backbone AP for delivery guarantees 100% packet delivery, but a node that bypasses the D2D network by using the backbone must pay a price equal to the initial maximum budget B_0 (when the packet was first introduced in the network).

Each node is free to advertise its own maximum budget and fine in its RFB, except for the maximum number of allowed hops, which decreases from the original H_0 every time the packet is forwarded one hop. In the nth auction for a given packet, $F_n \leq B_n$, i.e., the fine must be smaller than or equal to the budget defined in the RFB. Also, the advertised fine in every auction must be smaller than or equal than the fine agreed upon for the previous hop, i.e., $F_n \leq F_{n-1}$, where n indicates the nth auction for a given packet. After receiving the bids, a node chooses the winner

downstream node based on its own strategy. A node that wins an auction is allowed to drop the packet based on its own strategy. In order to avoid routing loops, a device is not allowed to bid for a data packet it has already forwarded once. An upstream node pays the agreed budget to the chosen downstream node if the packet is *successfully delivered to the destination AP*. Otherwise, the downstream node must pay the agreed fine to the upstream node if the data packet does not reach the AP destination within H_0 hops (and then, successively, all the way upstream). A node's balance may be temporarily negative.

4.2 Tightness Strategies

When a *source access point* (AP) announces its *request for bids* (RFB), it announces the budget B_0 along with a fine F_0 to be paid in case the data packet is not delivered to the destination AP within a deadline H_0. In this dissertation, the deadline is expressed in terms of *number of hops*, since it is a metric easier to deal with in the context of D2D communications[1]. It is assumed that the operator translates the actual time interval into a number of hops based on some estimate of the average delay per hop. Let hc_i denote the number of hops (or "hop count") of the *shortest path* computed from node i to the destination AP. Also, let p_i denote the number of hops traversed by a packet from the source AP to a given node i in the network. A key metric in the tightness strategy is the definition of a "tightness function" Δ_i for a node i in the network, i.e., Δ_i measures how "tight" a node i is with respect to making the deadline H_0 imposed by the source AP. In other words, given the timeout H_0 announced by the source AP, and the number p_u of hops already traversed by the data packet all the way to node i's upstream node u (the one who issues the RFB), Δ_i measures the "surplus" or "deficit" (in number of hops) that node i possess with respect to the timeout H_0 *if the data packet were forwarded through its shortest path to the destination AP*. In other words,

$$\Delta_i = (H_0 - p_u - 1) - hc_i, \quad \forall i \in \mathcal{N}(u), \tag{4.1}$$

where $\mathcal{N}(u)$ is the set of nodes who are able to overhear the RFB from node u, i.e., the neighbors of node u. Therefore, if $\Delta_i < 0$, node i cannot deliver the data packet within the deadline (even if the data packet follows node i's shortest path to the destination AP). On the other hand, if $\Delta_i = 0$, node i needs *exactly* the number of hops contained in its shortest path to the destination AP in order to make the deadline. This is a "tight" situation for node i, since it relies on the unpredicted outcome of other downstream auctions for the packet to arrive within the deadline. Finally, if $\Delta_i > 0$, node i has a higher chance to deliver the data packet within the deadline because the packet may even deviate from its shortest path to the destination AP, but it has a "surplus" of hops before the deadline is up.

4.2.1 Bidding Strategy

The rationale for making the bid takes into account the likelihood of fullfilling the task of delivering the packet at destination within the deadline. Otherwise, a fine will be paid to the

[1]The deadline was also expressed in terms of number of hops to destination in the MANIAC Challenge 2013 [16].

operator. Hence, each node needs to assess how likely it is to deliver the packet as compared to other auction contenders (or competitors). For that, we first need to determine the set $\mathcal{N}(u)$ of neighbors of the upstream node u. This set contains *our competitors* in the upcoming auction, and it can be easily found because all nodes have complete knowledge of the network topology. For each node $i \in \mathcal{N}(u)$, we compute Δ_i according to (4.1). Based on the values of Δ_i, we create a subset $\mathcal{S}(u) \subseteq \mathcal{N}(u)$ that contains all nodes in $\mathcal{N}(u)$ such that $\Delta_i \geq 0$, i.e., the set $\mathcal{S}(u)$ contains all nodes that are actually able to deliver the packet within the deadline and, therefore, they are the ones most likely to win the auction announced by node u (our actual competitors). Observe that, we are assuming that node u will usually prefer not to pay a fine. Given $\mathcal{S}(u)$, we want to estimate how competitive we are in terms of packet delivery from the point of view of node u. It is reasonable to expect that the likelihood of successfully delivering a packet will play a key role in any decision making by any node. Therefore, we choose to find out how competitive we are by using our "tightness function." Specifically, we compute how "tight" we are with respect to the *average tightness* $\overline{\Delta}$ of nodes in $\mathcal{S}(u)$, defined as

$$\overline{\Delta} = \frac{1}{|\mathcal{S}(u)|} \sum_{i \in \mathcal{S}(u)} (H_0 - p_u - 1) - hc_i = (H_0 - p_u - 1) - \overline{hc},$$

where $|\mathcal{S}(u)|$ is the cardinality of $\mathcal{S}(u)$, and $\overline{hc}$ is the average optimal hop count over all $i \in \mathcal{S}(u)$, i.e., the average *shortest path* to the destination AP computed for each node $i \in \mathcal{S}(u)$. Once the average tightness $\overline{\Delta}$ is found, we compute our *relative tightness* c_n, defined by

$$c_n = \frac{\Delta_n}{\overline{\Delta}}, \tag{4.2}$$

where the subscript n is used to identify ourselves. It is important to mention that the above computation will only happen if our tightness function is such that $\Delta_n > 0$ and $|\mathcal{S}(u)| > 0$. Otherwise, we have specific rules for making our bid (explained later).

Observe that, if $c_n < 1$ and $\Delta_n > 0$, then our competitors are better positioned than us (on average, with respect to a surplus of hop counts). Therefore, there is a high chance that they become more aggressive to win the bidding, since they may feel that they can deliver the packet in time. At the same time, since $c_n < 1$, it means that we are running a higher risk on not having the packet delivered to its final destination, compared to others. Therefore, we may want to set a higher bid (closer to the budget B_u) because the risk should not be worth it to take. In case $c_n \approx 1$, we have similar conditions than other competitors and, therefore, we should try to win the auction with a lower bid compared to previous case. However, if $c_n > 1$, it means that we are better positioned than the average of our competitors. Therefore, we should strive to win the bid by offering a very attractive price (closer to the fine F_u). In addition to c_n, another important metric to take into account is how Δ_n (the value of our tightness function) compares to the *biggest* value of Δ_i for $i \in \mathcal{S}(u)$. This is because, if $\Delta_n > \Delta_{\max} = \max_{i \in \mathcal{S}(u)} \Delta_i$, it means that we are the best choice for the upstream node u in terms of a positive surplus of hop counts towards destination. Therefore, we should strive to win the auction by becoming as aggressive as possible in our bid (i.e., to set lower values for the bid to make sure we win the auction). Otherwise, if $\Delta_n \ll \Delta_{\max}$, we should have very low expectations to win the auction and, therefore, we should

not make dramatic changes in our bid for different values of c_n. Based on that, we define the parameter a_n that compares our tightness value with the best tightness value in $\mathcal{S}(u)$, i.e.,

$$a_n = \frac{\Delta_n}{\Delta_{\max}}. \tag{4.3}$$

Given the values of c_n, a_n, the budget B_u, and fine F_u announced by the upstream node u, and since $F_u \leq B_u$ (according to auction rules), our offered bid $O(\boldsymbol{x})$, will be given by

$$O(\boldsymbol{x}) = (B_u - F_u)\left[1 - \frac{1}{1 + e^{-a_n(c_n - 1)}}\right] + F_u, \tag{4.4}$$

where $\boldsymbol{x} = [a_n \ c_n \ F_u \ B_u]$, and $F_u \leq O(\boldsymbol{x}) \leq B_u$, i.e., we opt for never making a bid less than the established fine F_u. As it can be seen, the logistic function is centered on $c_n = 1$, and the steepness of the curve is controlled by a_n. Figure 4.2 shows an example of the offered bid function for different values of a_n when $B_u = 200$ and $F_u = 80$. Finally, if $\Delta_n < 0$, we discourage the

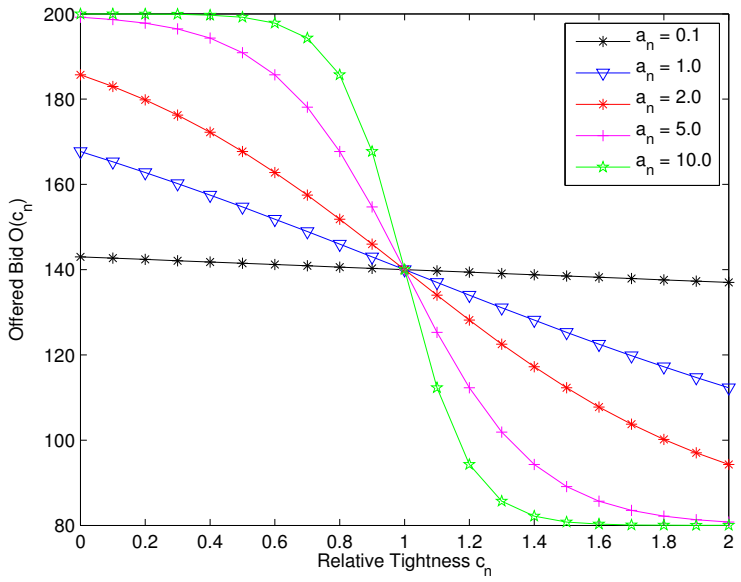

Figure 4.2: Example of offered bid curves $O(c_n)$ for different values of the parameter a_n when $B_u = 200$ and $F_u = 80$.

upstream node from choosing us by setting our bid equal to the budget B_u. Likewise, if there is no competition, i.e., we are the only node reachable by the upstream node, we set our bid to the maximum value B_u, and if $\Delta_n = 0$, it means that we are very "tight" and, therefore, we should set our bid to B_u (high risk).

4.2.2 Budget-and-Fine Set Up Strategy

Once an auction is won, the strategy to set the budget B_n and fine F_n values to be announced in an RFB is based on a fixed rule. Given that an upstream node has paid a node n an amount equal to the winner offer O^*, the budget B_n and fine F_n will be set to

$$B_n = 0.95 \times O^*_{n-1} \quad \text{and} \quad F_n = 0.4 \times B_n, \tag{4.5}$$

where the values of 40% and 95% were assumed to be reasonable values that can afford losses not so high from time to time (according to auction rules).

4.2.3 Decision-Making Strategy

In order to determine who wins an auction, an auctioneer considers both the bid b_i and *relative tightness* c_i of each node i who has replied to the announced RFB within a given time period (a timeout value is set after which a decision is made). Let $\mathcal{B}$ denote the set of bidders to the announced RFB. The node to which the packet is relayed is based on the outcome of a *preference function* P evaluated on the set $\{(b_i, c_i)|i \in \mathcal{B}\}$. The winner bidder is the one that provides the largest P value, i.e.,

$$\text{auction winner} = \arg\max_{i \in \mathcal{B}} P(b_i, c_i). \tag{4.6}$$

Notice that, for a given RFB, $F_n \leq b_i \leq B_n$, and $c_i \leq c_{\max}$, where $c_{\max}$ depends on the largest Δ_i for all $i \in \mathcal{B}$. Therefore, the auctioneer needs to compute c_i for all $i \in \mathcal{B}$ in order to decide the winner. In this work, two types of preference functions are used, based on which *three* different strategies are defined. The first preference function is based on a *hyperplane*, and the second is based on a *Gaussian* function.

Hyperplane Preference Function: the main motivation for a hyperplane as a preference function is its simplicity and low computational complexity. Also, by setting appropriate constant values, the plane can be tilted to reflect a certain weight towards b_i or c_i in the decision-making process. To define the hyperplane, we pick some points of interest and assign specific values to it. For instance, the lowest preference should be given to bidders with $c_i = 0$ and $b_i = B_n$, since these are nodes that charge the most to relay a packet in a very tight condition (no room for mistakes in the forwarding process). Hence, we set $P_n(0, B_n) = 0$. On the other hand, the highest preference should be given to bidders with $c_i = c_{\max}$ and $b_i = 0$, i.e., they have a "surplus" of hops before timeout happens (they are less tight), and they relay the packet for free. Other interesting cases are $P_n(0, 0)$, where the bidder is "tight," but it relays for free, and $P_n(B_n, c_{\max})$, where the bid is maximum, but the bidder has the lowest tightness. Hence, if we let $P_n(0, 0) = k_1$ and $P_n(B_n, c_{\max}) = k_2$, we may choose $0 < k_1 < k_2$ to reflect our tendency to favor packet delivery as opposed to increase our budget. The plane that intersects these points define $P_n(b_i, c_i)$, given by

$$P(b_i, c_i) = k_2 \left(\frac{c_i}{c_{\max}}\right) - k_1 \left(\frac{b_i}{B_n}\right) + k_1. \tag{4.7}$$

Notice that, the input values to the hyperplane are based on the *relative* values $c_i/c_{\max}$ and b_i/B_n. Therefore, this preference function is designed to work with any auction in the network, regardless of the specific RFB and bid values. Figure 4.3 shows an example of a hyperplane preference function with $B_n = 20$, $c_{\max} = 3$, $k_1 = 2$, and $k_2 = 3$.

Gaussian Preference Function: For the second preference function, we want to investigate a function that has a global maximum at a given *local operating point*. For that, we use a two-dimensional Gauss-like function because the operating point can be easily set up and we want to have its shape modified according to specific bid and RFB values of an auction (so, not only the

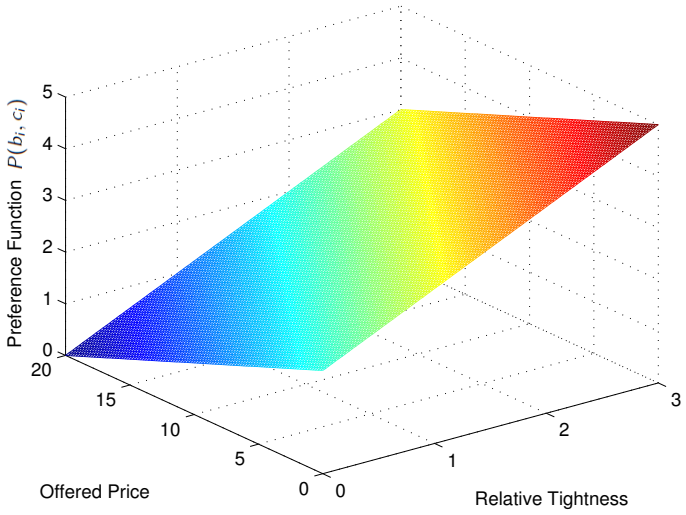

Figure 4.3: Example of preference function for the values $B_n = 20$, $c_{\max} = 3$, $k_1 = 2$, and $k_2 = 3$

operating point, but also the shape of the function is modified in every auction). Hence, $P(b_i, c_i)$ is given by

$$P(b_i, c_i) = \frac{1}{2\pi\sigma_b\sigma_c\sqrt{1-\rho^2}} \exp\left\{-\left[\frac{(b_i - b^*)^2}{2\sigma_b^2(1-\rho^2)} - \frac{2\rho(b_i - b^*)(c_i - c^*)}{2\sigma_b\sigma_c(1-\rho^2)} + \frac{(c_i - c^*)^2}{2\sigma_c^2(1-\rho^2)}\right]\right\}, \quad (4.8)$$

where (b^*, c^*) is the desired operating point, and σ_b, σ_c, and ρ control the shape of the function. Hence, given a set of $n = |\mathcal{B}|$ bid values, σ_b^2 and σ_c^2 are computed as

$$\sigma_b^2 = \frac{1}{n-1}\sum_{i=1}^{n}(b_i - b^*)^2\,, \;\; \sigma_c^2 = \frac{1}{n-1}\sum_{i=1}^{n}(c_i - c^*)^2\,, \quad (4.9)$$

i.e., σ_b and σ_c express the root-mean-square deviation from the operating point (b^*, c^*). Likewise, borrowing from the definition of correlation,

$$\rho = \frac{\sum_{i=1}^{n}(b_i - b^*)(c_i - c^*)}{(n-1)\sigma_b\sigma_c}, \quad (4.10)$$

which gives an idea of how "correlated" the sets $\{b_i\}$ and $\{c_i\}$ are, and define the shape of $P(b_i, c_i)$. Figure 4.4 shows an example of a preference function generated from data drawn from one of the auctions performed in simulations. Observe that, for each auction, one operating point is chosen, and all bids and tightness values are compared to the optimal case in that particular auction. The auction winner is the node whose bid and tightness values are closer to the operating point. In simulations, we investigate two operating points.

4.3 Baseline Strategies

In this chapter we define two baseline strategies for purposes of performance evaluation. The first strategy is designed to investigate what happens if the goal of every auctioneer is to deliver

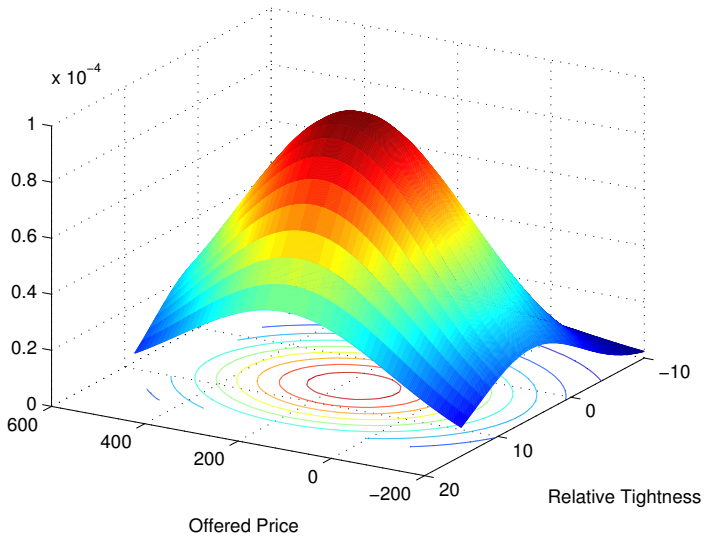

Figure 4.4: Adapted Gauss function, with center at $(179.178, 5.5)$

the packet to the destination no matter the values of the bids. In this case, the decision-making strategy of every auctioneer is simply to use shortest-path routing, i.e., to always relay the packet to the bidder in the shortest path towards destination, regardless of its bid. In addition, because the value of the bid is not taken into consideration, it is assumed that each node has its own, unknown, *bidding* strategy. To represent the collective behavior of every node having its own bidding strategy, we make every node to bid a value uniformly drawn from the interval $[F_u, B_u]$, where F_u and B_u are the fine and budget values announced in the received RFB. Finally, the *budget-and-fine setup* strategy follows the same one defined in Section 4.2. Henceforth, this strategy will be referred to as *Shortest Path* strategy.

The other strategy we investigate assumes that every auctioneer always relay the packet to the node whose bid is the lowest among the nodes $i \in \mathcal{B}$. Therefore, with this strategy, we investigate what happens if every auctioneer is greedy, and always want to increase its own budget regardless of packet delivery. Similar to *Shortest Path*, we assume that nodes run different bidding strategies that are collectively represented by random values chosen in $[F_u, B_u]$. Finally, the *budget-and-fine* setup strategy follows the same one defined in Section 4.2. Henceforth, this strategy will be referred to as *Lowest Bid* strategy.

Chapter 5

Application Model in NS-3

5.1 The NS-3 Simulator

To evaluate the performance of our strategies we used the *ns-3* simulator, which is a discrete-event network simulator targeted primarily for research and educational use. The *ns-3* project [20] is an open-source project that started in 2006. It strives to maintain an open environment for researchers to contribute and share their software (in our work we used the 3.17 version). Its simulation core and models are implemented in C++. *ns-3* is built as a library which may be statically or dynamically linked to a C++ main program that defines the simulation topology and starts the simulator. *ns-3* also exports nearly all of its API to Python, allowing Python programs to import an "ns3" module in much the same way as the *ns-3* library is linked by executables in C++.

5.1.1 NS-3 Organization

The source code for *ns-3* is mostly organized in the `src` directory and can be described by the diagram in Figure 5.1. This figure show the most important *modules* of the *ns-3*. An *ns-3* module may consist of more than one model (for instance, the `internet` module contains models for both TCP and UDP). In general, *ns-3* models do not span multiple software modules, however. Also, modules only have dependencies on modules beneath them. Here it is important to distinguish between modules and models:

- *ns-3* software is organized into separate *modules* that are each built as a separate software library. Individual *ns-3* programs can link the modules (libraries) they need to conduct their simulation.
- *ns-3 models* are abstract representations of real-world objects, protocols, devices, etc.

In Figure 5.1, the first module is the core of the simulator, which encompasses the components that are common across all protocol, hardware, and environmental models. The simulation core is

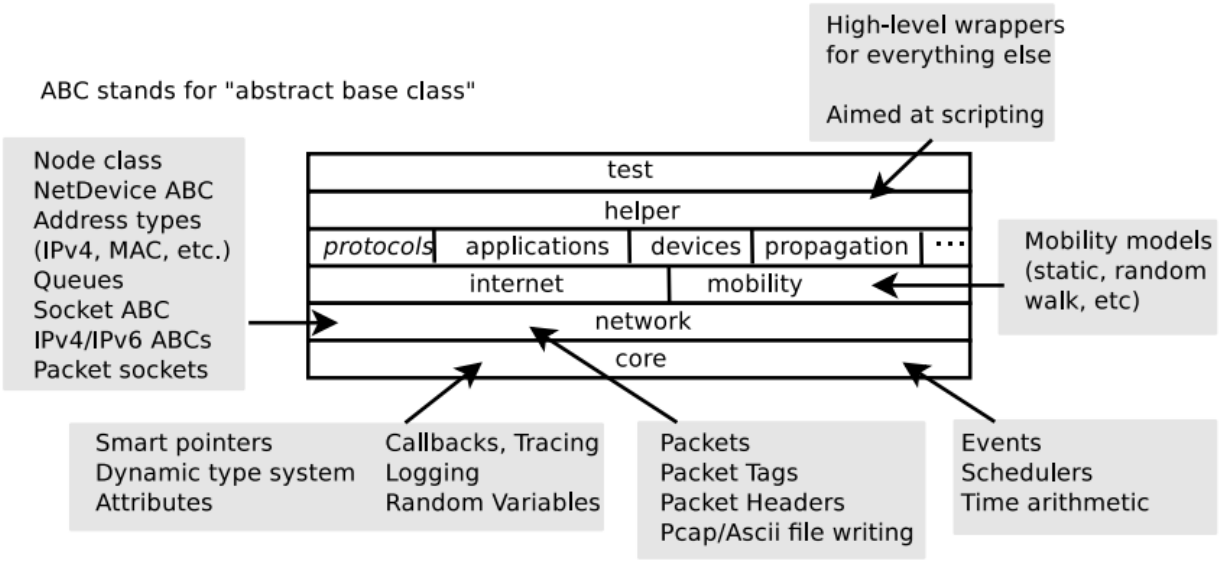

Figure 5.1: Software organization of *ns-3* [4]

implemented in `src/core`. Packets are fundamental objects in a network simulator and are implemented in `src/network`. These two simulation modules by themselves are intended to comprise a generic simulation core that can be used by different kinds of networks, not just Internet-based networks. The above modules of *ns-3* are independent of specific network and device models.

The basic element of the simulator is the *node* (implemented by the `ns3::Node` class). To the node are added the NetDevices (equivalent to the network adapters) and objects related to the protocols and applications. The `ns3::Node` class can be inherited. However, it gives preference to the aggregation and insertion of objects. The Figure 5.2 shows a high-level diagram and some objects that can be aggregated to the node.

5.1.2 A New Application Model

In several cases, users may not be satisfied with a mere adaptation of the existing models. They may want to extend the core of the simulator in an innovative way. The architecture of the Figure 5.1 was designed to facilitate the addition of new features. For this, we need to decide in which sub-folder we have to put the new model. The sub-folder `src/devices` contain the network devices models, such as `Wifi`, `Wimax`, `CSMA`, `P2P`, etc. Whereas the sub-folder `src/applications` comprises several types of applications, for instance, `OnOff`, `UdpEcho`, etc. The sub-folder `src/internet-stack` has the classes that deal with the network and transport protocols. The routing algorithms can be found in the `src/routing` sub-folder.

In this project, we created a new application model for mobile data traffic offloading, located at the `src/applications` folder (application module). That is why we will describe with more details its operation and organization. Below we briefly explain the *packets structure* of our new application, the `ns3::Offloading` application class and the helper class (`ns3::OffloadingHelper`).

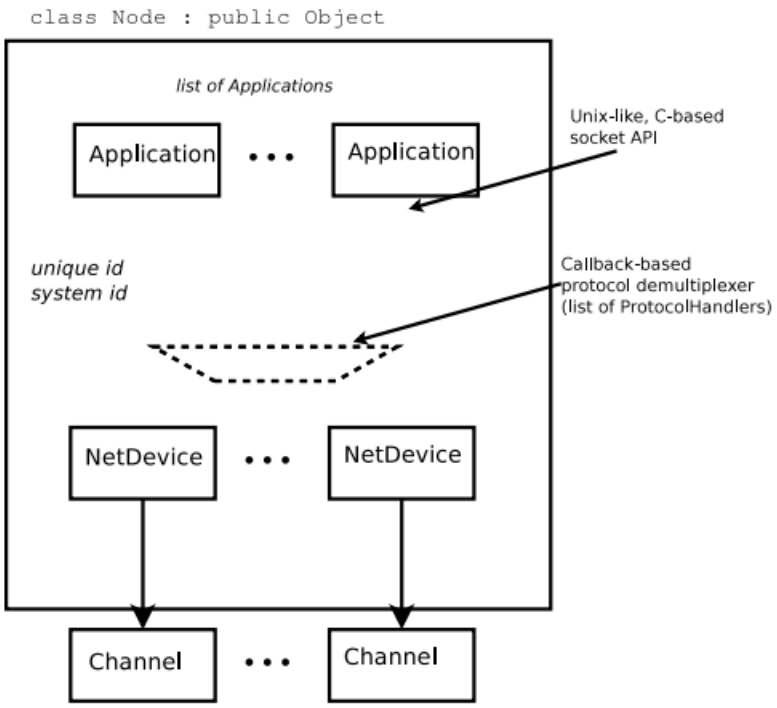

Figure 5.2: High-level node architecture [5]

5.1.2.1 Packet Structure

As explained in Chapter 4, we have different types of packets for our application, each one with different information in its headers. To manage this, we created the files `offloading-packet.cc` and `offloading-packet.h` that contains the description of each packet structure and size (they were placed in the `src/applications/model/` directory). These files are essential to the operation of the application, and are used in the `ns3::Offloading` class operation.

To facilitate the identification of the packets type, we used some notations in these files. The `MessageType` enumerator represents these notations below:

```
enum MessageType
{
  OFFLOADINGTYPE_RFB  = 1,   //!< OFFLOADINGTYPE_RFB
  OFFLOADINGTYPE_BID  = 2,   //!< OFFLOADINGTYPE_BID
  OFFLOADINGTYPE_DATA  = 3,  //!< OFFLOADINGTYPE_DATA
};
```

Then each packet type has its identification. To store this information in the packet we used the `ns3::TypeHeader` class, which is inherited by the `ns3::Header` class. Thus the `ns3::TypeHeader` class defines the type of each packet, which, consequently, defines the packet structure.

The RFB packet header is defined in the `ns3::RFBHeader` class, which is also defined in the `offloading.cc` file (all the packet header classes created are defined in this file). The RFB packet header has its format showed in Figure 5.3, with the length of each field expressed in bits. The

first field describes the type of packet, with a length of 1 byte (8 bits). The "Hop count" field (1 byte) represents the number of hops of the shortest path calculated via OLSR information from the auctioneer node to the destination AP. The "Deadline" field (1 byte) is used by the source AP to set the maximum number of hops allowed for the packet to traverse. It is an integer, constant value that the nodes cannot change. The "Packet ID" field (1 byte) contains the ID of this packet. It is generated by the source AP. The "Bn" and "Fine" fields (4 bytes each) are the budget and fine values respectively, set by the auctioneer node. The "Source IP Address" and "Destination IP Address" fields (4 bytes each) are the addresses of the APs involved (source and destination). The last field, "RFB Source MAC Address" (6 bytes), is the MAC address of the auctioneer node, which is used internally by the `ns3::Offloading` class to avoid collisions among the ARP packets. The ARP protocol is responsible to "translate" the IP addresses to MAC addresses to make the communication in the MAC layer feasible. However, in our auctions too many nodes tries to broadcast bid packets at the same time which causes some collisions between the ARP packets at the MAC layer. Then to prevent this, we pass the MAC address directly through the RFB packet which force the participant nodes to complement its ARP tables and avoid to send ARP packets later. At a real case, this field would not be mandatory since there are other solutions to prevent ARP packets collisions, but we used this solution to our simulations as a temporary solution. Thus, the total length of the RFB header is 26 bytes.

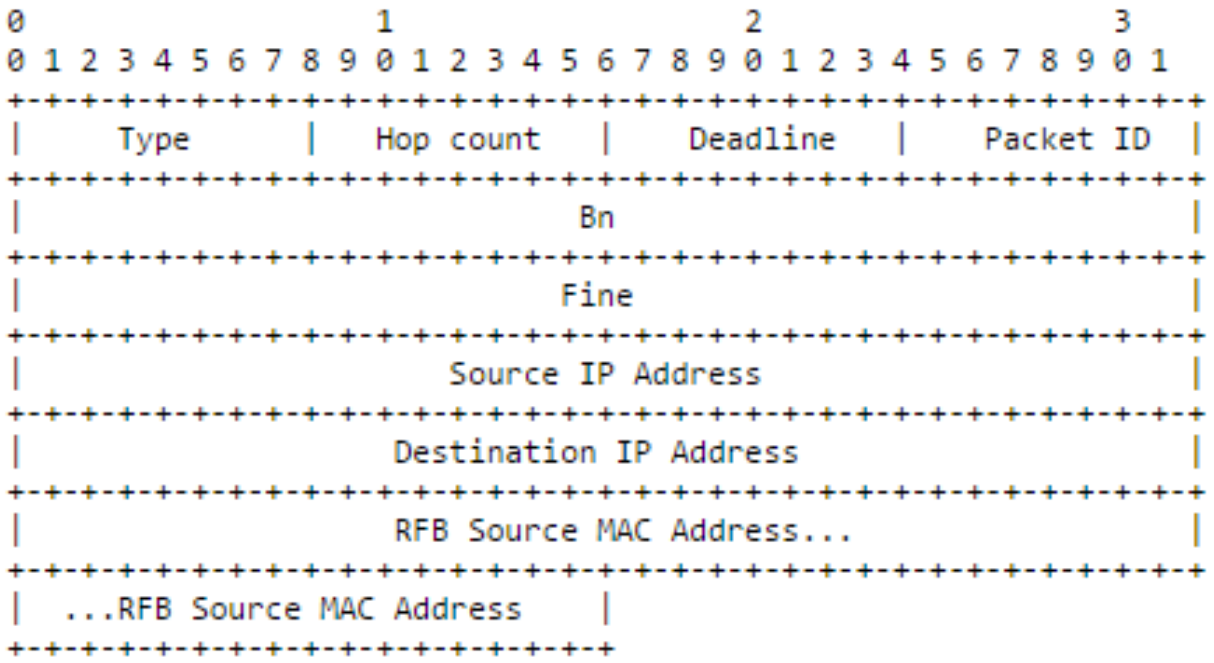

Figure 5.3: RFB packet header format.

For the bid packet header, we created the `ns3::BidHeader` class, and its header format is presented in Figure 5.4. The fields "Type", "Packet ID", "Source IP Address" and "Destination IP Address" have the same length and functions of the RFB header. The "Reserved" field (2 bytes) is not used, but is reserved for this packet to future use. The "Bid Offer" field (4 bytes) is the value of the bid of the node competitor that generated this packet. Finally, the total size of the

bid packet is 16 bytes.

Lastly, the winner node of each auction will receive the packet containing the actual data. We created the `ns3::DataHeader` class to describe the header of this packet, and its structure is showed in Figure 5.5. This packet header is more simple, and only have some basic fields: "Type", "Source IP Address", "Packet ID", "Destination IP Address" and "Hop Count". The total size of this data header is 11 bytes. The size of the data payload is defined in the `ns3::Offloading` class.

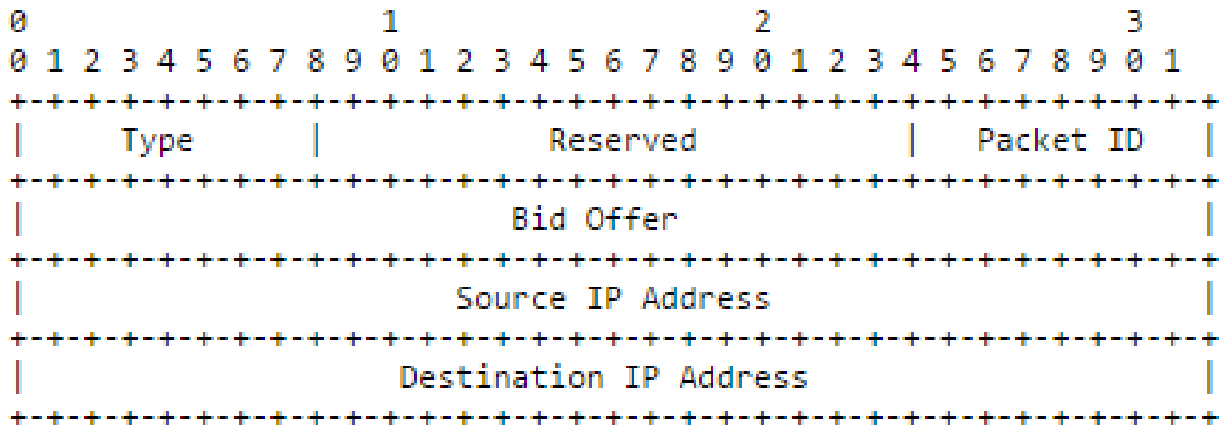

Figure 5.4: Bid packet header format.

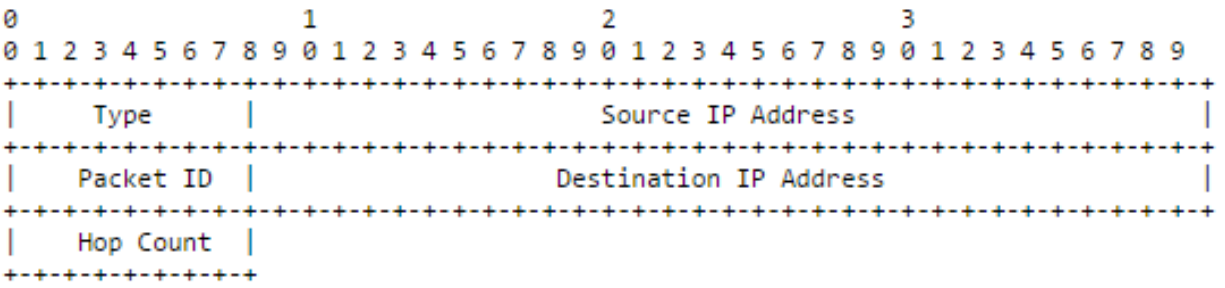

Figure 5.5: Data packet header format.

5.1.2.2 Offloading Helper

The *ns-3* simulator makes use of various advanced programming concepts such as smart pointers for reference-counted memory management, attributes, namespaces, callbacks, etc. Users who work at this low-level API can interconnect *ns-3* objects with fine granularity. However, a simulation program written entirely using the low-level API would be quite long and tedious to code. For this reason, a separate so-called "helper API" has been overlaid on the core *ns-3* API.

A helper API has a few goals:

- The rest of `src/` has no dependencies on the helper API; anything that can be done with the helper API can be coded also at the low-level API

- **Containers**: Often simulations will need to do a number of identical actions to groups of objects. The helper API makes heavy use of containers of similar objects to which similar or identical operations can be performed.

- The helper API is not generic; it does not strive to maximize code reuse. So, programming constructs such as polymorphism and templates that achieve code reuse are not as prevalent. For instance, there are separate CsmaNetDevice helpers and PointToPointNetDevice helpers but they do not derive from a common NetDevice base class.

- The helper API typically works with stack-allocated (vs. heap-allocated) objects. For some programs, *ns-3* users may not need to worry about any low level Object Create or Ptr handling; they can do with containers of objects and stack-allocated helpers that operate on them.

The helper API is really all about making *ns-3* programs easier to write and read, without taking away the power of the low-level interface. And it was based on this principle that we decided to build our own helper for our `ns3::Offloading` application. This helper is described in the `offloading-helper.h` and `offloading-helper.cc` files, and they were placed in the `src/applications/helper/` directory. In these files we basically create the `ns3::OffloadingHelper` class and its functions that will help the simulation to use the `ns3::Offloading` class.

The first function is the constructor `OffloadingHelper::OffloadingHelper`, that creates an instance of this helper. Following, the `OffloadingHelper::SetAttribute` function set the attributes of the `ns3::Offloading` class. These attributes will be presented in Section 5.1.2.3. There are three types of `OffloadingHelper::Install` functions: one that install a single node by reading its smart pointer `Ptr<Node>`, other that installs a single node by reading its name and another that installs all the nodes of the `NodeContainer`. All these three `OffloadingHelper::Install` functions aim to install our `Offloading` application in the nodes.

The function `OffloadingHelper::SetBackbone` is used to inform some node about what is the set of node IP addresses that belongs to the backbone, i.e., to inform some node the whole set of APs. This helps the `Offloading` application to deliver the packets to the right destination. The `OffloadingHelper::SetTightnessParameters` function sets directly to the application 4 basic parameters of the "Tightness" strategies: k_1, k_2 and the RFB percentage parameters (0.95 and 0.4 in our simulations). The function `OffloadingHelper::SetMapNodes` passes the graph mapping to all the nodes of the node container. This mapping is essential to the Dijkstra calculation of the shortest path, because we have to map the IP addresses to integer numbers to allow the graph calculation. Finally, all the remaining functions (`OffloadingHelper::SetTopologyName`, `OffloadingHelper::SetSeedIndex`, `OffloadingHelper::SetParamName` and `OffloadingHelper::SetExpIndex`) are used to create the respective folder of the experiment, which must include the topology name, seed number, parameters (k_1 and k_2 values) and the name of the experiment (the name of the experiment is chosen by the simulator user at each new experiment).

5.1.2.3 The `Offloading` Class

Now we will focus on explaining the `ns3::Offloading` class that is installed in each node of the network. Similar to all *ns-3* models, we have to set some attributes and functions. In Section 5.1.2.2 we explained that the `OffloadingHelper::SetAttribute` is used to give values to the `ns3::Offloading` class attributes. Now we will describe them.

The first attribute is the "NPackets", which is the number of packets that the node will generate if the node is an AP. The "StrategyType" and "PreferenceFunctionType" are the types of the strategy and Preference Function used by the node (in case that the strategy is a *Tightness* strategy), respectively. The following values are the possible options for these two types:

```
enum StrategyType
{
  STRATEGYTYPE_DUMMYBID  = 1,   //!< STRATEGYTYPE_DUMMYBID
  STRATEGYTYPE_DUMMYPATH  = 2,   //!< STRATEGYTYPE_DUMMYPATH
  STRATEGYTYPE_TIGHTNESS  = 3,   //!< STRATEGYTYPE_TIGHTNESS
};

enum PreferenceFunctionType
{
    PREFERENCEFUNCTION_PLANE = 1, //!< PREFERENCEFUNCTION_PLANE
    PREFERENCEFUNCTION_GAUSS = 2, //!< PREFERENCEFUNCTION_GAUSS
    PREFERENCEFUNCTION_GAUSS_1 = 3, //!< PREFERENCEFUNCTION_GAUSS_1 (with cn=1)
};
```

The "Destination Address" is the IP address of the destination AP. "RemotePort" is the destination port of the outbound packets (we used the port number "9" in the main script of the simulations). The "Budget", "Fine" and "Deadline" attributes are the basic parameters of the RFB generated by the source AP, i.e., B_0, F_0 and H_0. After the application is installed and initiated in the AP node, the application waits for "StartOffloading" seconds to initiate packet generation. Finally, "NumberNodes" is the number of nodes using OLSR, the routing protocol used to build the topology graph in the "Tightness" strategies.

After defining the class attributes, we define the functions. The first is the basic constructor `Offloading::Offloading()`, that sets initial values of some internal variables and events. Next, we have a basic function called `Offloading::SetRemote` that sets the socket values (IP address and port number), and it has two variants depending on the input address type. The following set of functions (`Offloading::SetBackbone`, `Offloading::SetTightnessParameters`, `Offloading::SetMapNodes`, `Offloading::SetTopologyName`, `Offloading::SetSeedIndex`, `Offloading::SetParamName` and `Offloading::SetExpIndex`) are the ones that are used by the `OffloadingHelper` to facilitate the initial configuration of the application, as explained in Section 5.1.2.2.

In addition to these basic functions we have the essential functions that must exist in all the application models: `StartApplication` and `StopApplication`. The `Offloading::StartApplica-`

`tion` function configure the initial environment, like creating the offloading report files, separating who is a backbone node (AP) (and not), configuring the application socket, mapping the network graph nodes and initiating (and scheduling) the AP RFB event. The `Offloading::StopApplication` clears the sockets and cancels the application events.

Another essential function and very common in the *ns-3* application models is the `HandleRead` function. The basic idea of this function is to get the packets received by the node and treat each of them. It is important to include this function because of the packet reception mechanism of the *ns-3* simulator. The Figure 5.6 shows how the packet arrives at the application layer, after passing through all the lower layers.

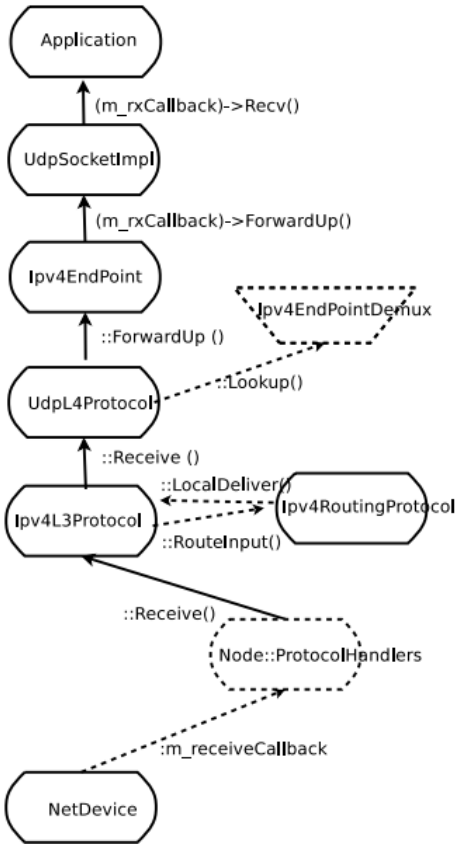

Figure 5.6: Receive path of a packet [5].

To get the packet and treat its content, first we have to deal with the application socket by using the `RecvFrom(Address)` function from the `Ptr<Socket>` smart pointer. This function will return the packet received by the node socket from the node with the address "Address". After that, we remove the packet header, that contains the type of packet being treated (RFB, bid or data). Then, the next action will depend on the type of the packet. For example, if we received an RFB packet, we have to send our bid; or if we receive a bid packet, we have to decide who is the winner node and send a data packet.

To take the actions after treating the packets in the `Offloading::HandleRead` function, we need other three basic functions: `Offloading::SendRFB`, `Offloading::SendData` and `Offloading::-`

`SendBid`. They are responsible for sending the three types of packets of the offloading application. It is important to note that these three functions and the actions of the `HandleRead` function are all based on the rules and strategies discussed in Chapter 4.

It is also very important to remember that this *ns-3* offloading model is a cross-layer implementation, which involves the application layer and the *network layer*, because of the OLSR dependence. To communicate with this layer, and get the OLSR information about the topology of the network, we created some specific functions, like `Offloading::HopCountComputation`, `Offloading::BuildGraphDijkstra`, `Offloading::DijkstraComputePaths` and `Offloading::DijkstraGetShortestPathTo`. All these functions use the OLSR state information, obtained from the OLSR routing protocol model, located in the `src/routing/olsr/` directory. To let the OLSR state public, we had to modify some files, as showed in Section 5.1.3 and Appendix II.

Finally, we created the `Offloading::WritePacket`, which is responsible for writing the packet information in the offloading reports. These reports and its post processing operation will be discussed with more details in the Section 5.2.

5.1.3 Modified Modules

All the *ns-3* files, released 3.17, that have been modified are showed in the Appendix II section. These modifications were made to allow our application to access some *ns-3* class information (as in the OLSR case, where we modified the `olsr-routing-protocol.h` file) or to expand the number of inputs allowed at some *ns-3* core class functions (like the modifications made at the `make-event.h` and `simulator.h` files). In the Apendix, we only show the differences between the original and modified files, by using the Unix command `diff`.

5.2 Post Processing

After running simulations in the *ns-3* simulator, we have to process all offloading data that is gathered. All the simulation data is stored in reports using the text file format. Each node generates a file containing a table with some columns representing packet information, and each row of the table represents a packet that is offloaded by the node. As an example, Table 5.1 illustrates a real case that we collected. This table was generated by the simulation of the Tightness strategy without mobility, with $k_1 = 2$ and $k_2 = 3$ (in this case, the table corresponds to the whole report file of node 032).

Each column of Table 5.1 has its meaning. The first column is the ID of the source Access Point node that generated the packet (three digits, “000” to “999”). The second column indicates the packet ID. Since each source node sends 50 packets, the packet ID ranges from “0” to “49”. The “Next” column indicates the next node to which the packet was sent after the auction was over. In other words, it is the winner node. In case of success (when the packet reaches destination within the deadline), this node has to earn the bid that was negotiated in the auction with the upstream node (“eBID” column), and it has to pay the bid that was negotiated with the next node winner

Source	pktID	Next	eBID	pFINE	pBID	eFINE	sBUDGET	fBUDGET	status	balance	accum
001	22	104	950.00	380.00	902.50	361.00	47.50	28.50	3	28.50	28.50
001	30	104	950.00	380.00	902.50	361.00	47.50	28.50	3	28.50	57.00
001	32	104	950.00	380.00	902.50	361.00	47.50	28.50	3	28.50	85.50
002	11	096	700.00	400.00	665.00	266.00	35.00	-99.00	1	35.00	120.50
002	16	096	700.00	400.00	665.00	266.00	35.00	-99.00	1	35.00	155.50
002	20	096	700.00	400.00	665.00	266.00	35.00	-99.00	1	35.00	190.50
002	38	096	700.00	400.00	665.00	266.00	35.00	-99.00	1	35.00	225.50
002	41	096	700.00	400.00	665.00	266.00	35.00	-99.00	1	35.00	260.50
002	47	096	700.00	400.00	665.00	266.00	35.00	-99.00	3	-99.00	161.50
030	19	034	665.00	266.00	631.75	252.70	33.25	19.95	1	33.25	194.75

Table 5.1: Example of a node offloading report at the end of a simulation.

("pBID"). In case of fail, it will also have to pay the fine imposed by the upload node ("pFINE"), and it will earn the fine of the next node ("eFINE"). In the end of the simulation (and before doing the post processing) we already know that the budget of this packet will be a successful packet budget ("sBUDGET") if the packet reached the destination within the deadline, or a fail packet budget ("fBUDGET") if the packet was dropped. Then, depending on the final packet status, the packet budget will assume one of the values presented in Eq.(5.1):

$$\begin{aligned} &sBUDGET = eBID - pBID, \ or \\ &fBUDGET = eBID - pBID - pFINE + eFINE \end{aligned} \tag{5.1}$$

5.2.1 Accountability Process

Each node report file was generated by the *ns-3* simulator. But, to process these reports after simulations, we used Perl scripts, which is one of the best and simplest languages to manipulate this kind of text files. So, the last three columns are used by the Perl scripts to make the computation, which we called the *accountability process*, and it is the first part of the data processing.

The accountability can be done in real time or offline. The latter is simpler because it is done after the network ends its operation, and then, the telecommunications company can collect all the node reports and make all the accounts, including discounting fines. In the real time mode, it does this process at the same time as the packet reaches the backbone or it is dropped. However, as the real time case is more complex, and it does not represent such an advantage for this study, we decided to use the offline case.

Now, let us understand what we computed and what these last 3 fields of the node report file represents. First, the "status" field is a number that indicates what happened with the packet in the end. If the packet was successful (reached the right destination within the deadline H_0) then $status = 1$; if it reached the backbone within the deadline, but at the wrong destination AP, then $status = 2$; if it reached the backbone after the deadline, then $status = 3$; and finally $status = 4$ if the packet was dropped before reaching the backbone. This "status" variable is important because

it makes possible for the Perl scripts to calculate the Packet Delivery Ratio (PDR) of the AP nodes, to know which nodes will have to pay a fine, how much this fine value would be (in case of failure) and how many hops each packet had to go through.

The "balance" field depends on the status value. If $status = 1$, then $balance = sBUDGET$, otherwise $balance = fBUDGET$. Thus, the balance is nothing but a final "packet budget". Finally, the "accum" field is an accumulated value of each final packet budget. Therefore, the last field of the last line of the report file table represents the total budget accumulated by the node.

Chapter 6

Simulation Results

In the chapter, we present simulation results divided into two groups, according to node mobility: static and mobile topologies. But first, let us discuss about the simulation scenarios. As presented in Section 4.2 there are many possibilities to set up each sub-strategy in the "tightness strategies" class. Therefore, we focus on *three* specific setups, which are defined according to the chosen *decision-making* strategy and respective parameters. To differentiate them, the following nomenclature is used:

- *Tightness*: this is the tightness strategy based on the hyperplane preference function with parameters $k_1 = 2$, and $k_2 = 3$, i.e., a slightly higher weight is given to the ratio $c_i/c_{\max}$ as opposed to b_i/B_n (packet delivery is considered more important than budget);
- *Gauss*: Gaussian preference function with operating point $(F_u, c_{\max})$, i.e., highest preference is given to the bid that is the closest to the smallest possible value (the announced fine F_u), and whose node has a tightness value closest to $c_{\max}$. This would locally maximize the budget and the likelihood of deliverying the packet within the deadline (surplus of hops before timeout is reached);
- *Gauss*$_1$: Gaussian preference function with operating point $(F_u, 1)$. In this case, the highest preference is given to the bid that is the closest to F_u, but whose bidder has a tightness value equal to the average tightness ($c_n = 1$). This is a more relaxed situation, where the surplus of hops to destination is not considered so critical to make a decision on the auction winner;

6.1 Simulation Scenarios

The performance of *Tightness*, *Gauss*, and *Gauss*$_1$, as well as *Shortest Path* and *Lowest Bid*, is evaluated with discrete-event simulations using the *ns-3* network simulator [20]. Ten topologies are used with 100 nodes forming the D2D network, and 32 other nodes acting as Wi-Fi access points (APs) to the operator's backbone. All AP nodes are fixed located and evenly spaced on the border of a square terrain of 800 × 800 m. Two scenarios are investigated: *static* and *mobile*. In the static scenario, one of the topologies is based on a grid of nodes forming the D2D network,

while the other topologies are based on nodes randomly placed on the terrain. Figure 6.1 depicts an example of a random topology used in simulations, where the green lines indicate connectivity between nodes (transmission range).

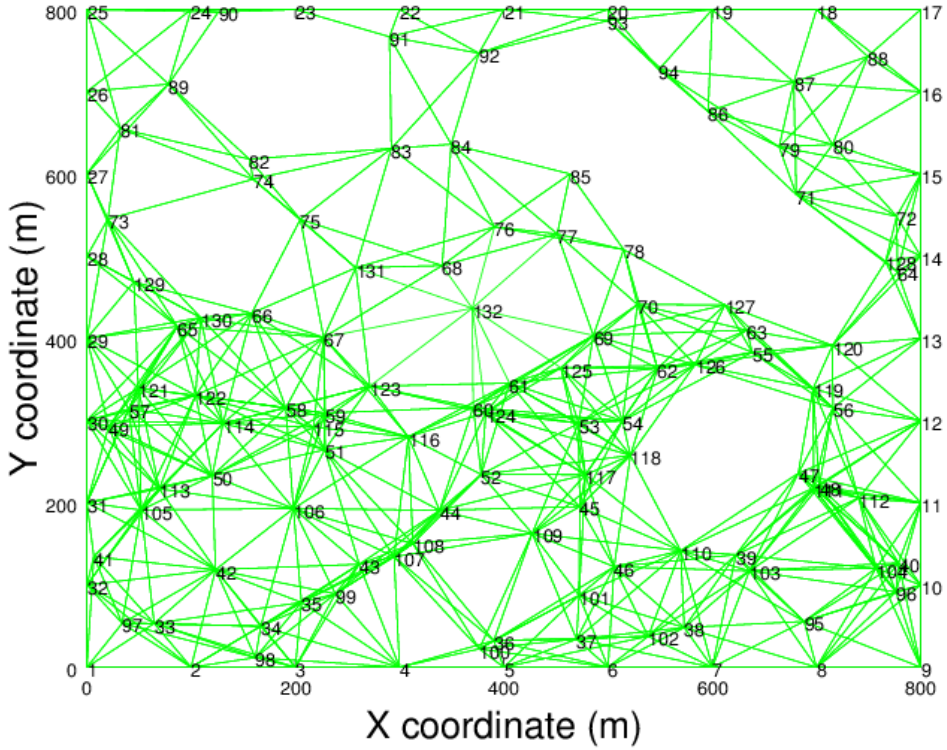

Figure 6.1: Example of random topology used in simulations. The green lines indicate connectivity between nodes based on the transmission range.

In the mobile scenario, all D2D nodes move according to the random walk mobility model available in the *ns-3* simulator. Three mobile scenarios are investigated, based on three different speeds: 0.5 m/s, 0.75 m/s, and 1.0 m/s. In all three scenarios, nodes change direction every 10 m (randomly). The chosen speeds reflect *walking* behavior, which is an appropriate scenario for per-packet recursive auctions, and also because the investigated strategies rely on the knowledge of network topology (apart from *Lowest Bid*): topology information becomes less reliable as mobility becomes too high. All nodes utilize the OLSR protocol to gather topology information and to run their strategies. Every simulation has a "warm-up" period of 30 s before any auction happens, during which the nodes start moving around and OLSR operates. This is to allow dissemination of topology control information and to let the nodes have their routing tables populated before the beginning of any auction.

As far as traffic generation is concerned, each AP node offloads a total of 50 packets into the D2D network. But, each AP only starts its auctions when its neighbor AP finishes the auction of all 50 packets, i.e., AP nodes transmit consecutively, one after the other. Also, each AP node has a fix destination AP to which all of its 50 packets are addressed. The destination AP is roughly located in the opposite direction of the transmitting AP in the topology (i.e., diametrically opposed) so

that the number of hops to destination is maximized (to make the offloading job more challenging). This traffic generation pattern aims to provide a somewhat fair distribution of data flows in the network, while avoiding location-specific interpretation of results (one of the goals of this study is to understand fairness issues between strategies).

The time interval between the issue of requests for bids is 3.0 s. The *auction timeout*, i.e., the time interval that a node waits before deciding for the winner of an RFB is 50 ms. Consequently, the next AP in sequence waits for 160 s before issuing its first RFB (time for offloading all 50 packets plus a guard interval of 10 s). The simulation is over once every AP node finishes offloading all of its packets. This happens after 5,200 seconds of network operation (simulated time), which includes an extra time interval to guarantee the relay of any packet still traveling in the network. Each packet has an initial "deadline" (H_0) of 10 hops, an initial budget (B_0) of 1000, and an initial fine (F_0) of 400. Finally, for the MAC- and PHY-layer parameters, all network nodes operate according to the IEEE 802.11g ad hoc mode in the 2.407 GHz frequency channel. All frames are transmitted at 1 Mb/s, and no RTS/CTS frames are used. Energy detection threshold is set to -67.5785 dBm, while the CCA threshold is set to -71.1003 dBm. Transmit power is 16.0206 dBm, which corresponds to a transmit range of 150 m under the Friis large-scale channel propagation model. No small-scale fading was implemented, since we wanted to minimize the occurrence of errors due to channel impairments (and have a better ideia of packet delivery by each strategy). But, errors due to large-scale propagation effects (path loss) could still occur, as well as packet collisions, especially with OLSR broadcast messages or simultaneous bids (under CSMA/CA operation, of course).

The strategies are investigated based on four performance metrics: *packet delivery ratio* (PDR), defined as the ratio of the number of packets delivered to destinations to the total number of packets offloaded to the network; the *relative average budget* (RAB), defined as the ratio of the average accummulated budget per node to the initial budget announced by every access point (i.e., $B_0 = 1000$). In this case, we compute a relative value because the initial budget is just a symbolic value. Therefore, it makes more sense to understand the average accumulated budget per node as a *gain* over the announced budget per packet. The other two metrics are *fairness*, defined according to Jain's fairness index [27]

$$\mathcal{J}(x_0, x_1, x_2, \ldots, x_n) = \frac{\left(\sum_{i=0}^{n} x_i\right)^2}{n \sum_{i=0}^{n} x_i^2}, \tag{6.1}$$

where x_i is the final budget at node i; The idea of this metric is to understand how fair each strategy is regarding budget distribution among nodes. Finally, we investigate the *average number of hops* (ANH) traversed by all packets that are offloaded to the network (successfully transmitted or not).

6.2 Static Topologies

Now we present simulation results, according to node mobility: static and mobile topologies. Figure 6.2 presents the results for the relative average budget (RAB) of all strategies. All tightness strategies perform better than *Shortest Path*. In particular, *Tightness* and *Gauss* present the best

Parameter type	Values
Topologies	- Mobile scenario: 10 topologies randomly generated - Static scenario: 1 grid + 9 randomly generated 800 × 800 m square terrain
Mobility	- 1.0 m/s, 0.75 m/s and 0.5 m/s - Random walk model: change direction every 10 meters
Traffic	Deterministic (50 packets per AP)
Time Intervals	- OLSR "'warm up" = 30 seconds - Between RFBs = 3.0 seconds - Auction timeout = 50 milliseconds - Guard interval = 10 seconds - Simulation time = 5200 seconds
RFB Values (AP)	H_0 = 10 hops B_0 = 1000 F_0 = 400
MAC and PHY	- IEEE 802.11g ad hoc mode, 2.407 GHz - Frames at 1 Mb/s - Energy detection threshold = -67.5785 dBm - CCA threshold = -71.1003 dBm - Transmit power = 16.0206 dBm - Transmit range (Friis) = 150 meters
Performance Metrics Evaluated	- PDR (packet delivery ratio) - RAB (relative average budget) - ANH (average number of hops) - Fairness: Jain's index

Table 6.1: Summary of parameters used in the simulations.

results, with RAB values of 12.20 and 12.17, respectively, while $Gauss_1$ performs slitghly worse, with 11.74 RAB, but still better than *Shortest Path* with its 8.96 RAB. This indicates that, in the static scenario, the bell-shaped preference functions (with operating points) are as profitable as the hyperplane preference function. *Tightness* provides a gain of 36.15% over *Shortest Path*, while $Gauss_1$ also obtains good performance, with a 31.03% gain over *Shortest Path*. *Lowest Bid* delivers poor performance, since it always picks the node with the smallest bid, regardless of its chances to deliver the packet at destination. These results indicate that one of the design goals of the tightness strategies was achieved, which is that of incentivizing nodes to join the D2D network by delivering high profits to those that participate in the recursive auctions.

Figure 6.3 presents the results for packet delivery ratio (PDR). *Shortest Path*, *Tightness* and *Gauss* deliver similar performance, with 0.94, 0.929 and 0.928 PDR, respectively. $Gauss_1$ also shows competitive performance, with 0.91 PDR. These results are very important, since they show that

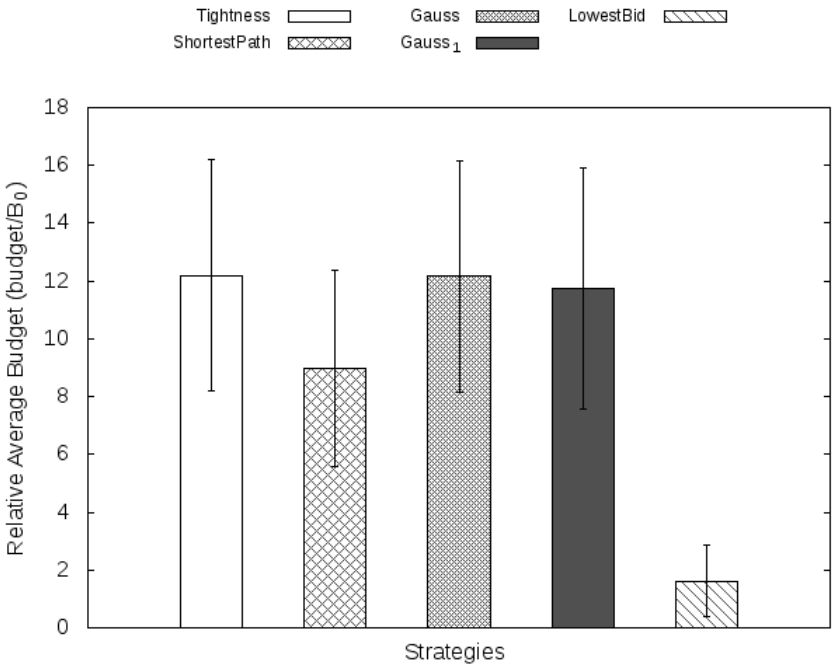

Figure 6.2: Relative average budget of all strategies under static topologies.

it is possible to achieve packet delivery ratios as competitive as those provided by *Shortest Path* while, at the same time, guaranteeing higher RAB values per node. Also, it is interesting to observe the low PDR variability between topologies in all strategies (including the tightness strategies), as indicated by the standard deviation in the graphs. This indicates that, in the static scenario, and compared to budget results, all strategies tend to lead to more stable routes toward destinations (in the sense of PDR within the deadline), while budget accumulation is more dependent on the type of topology. Finally, as expected, *Lowest Bid* performs very poorly, delivering almost no packets to destination. This is because it does not aim to deliver the packets within the deadline, which leads to an excessive number of packet drops.

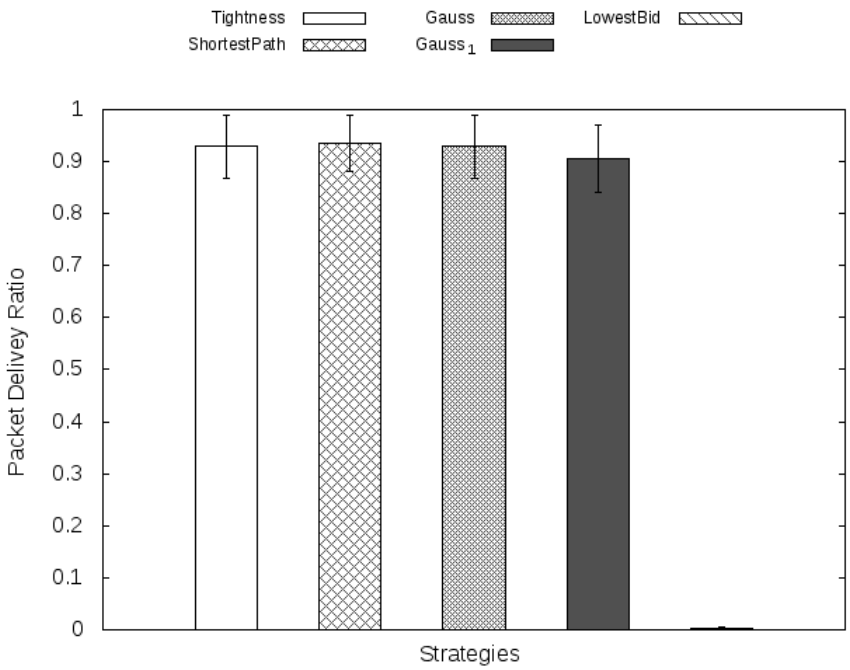

Figure 6.3: Packet delivery ratio (PDR) of all strategies under static topologies.

Figure 6.4 depicts the average number of hops (ANH) traversed by packets (successfully or not). *Gauss* and *Tightness* deliver the best performance with similar results: 8.33 ANH. Surprisingly, *Shortest Path* performs slightly worse than *Gauss* and *Tightness*, with 8.69 ANH This result suggests that introducing the *timeout* constraint into the bidding and decision process (in terms of number of hops) has a clear benefit to the overall ANH traversed by packets. In the tightness strategies, the nodes themselves encourage (or discourage) the reception of a packet through their bids, which take into account how tight a node is to deliver the packet within the deadline. Curiously, however, the *request-for-bids* strategy used by $Gauss_1$ does not provide a good ANH. This is related to the fact that $Gauss_1$ uses the *average* tightness $c_n = 1$ in its operating point, as opposed to $c_{\max}$ used by *Gauss*. Thus, $Gauss_1$ prefers to relay the packet to a node that is as *tight* as its neighbors (on average), and whose bid is close to F_u. Consequently, it is prone to deliver the packet to someone that is not so likely to deliver the packet to destination. Here, it is important to remember that all nodes use the OLSR protocol, which delivers a *partial* view of the network topology, since the paths are computed over the *multipoint relays* only [17]. Therefore, not all nodes are known to everyone, and some inaccuracies exist on shortest-path computation. Using the tightness information for the bidding and decision process, there is a reassurance of the best path since nodes may have different topology information. As expected, *Lowest Bid* has the poorest performance, with about 9.81 ANH.

Figure 6.5 shows the results for budget fairness among nodes. *Shortest Path* achieves the best fairness with 0.69, surpassing *Tightness* with 0.579, *Gauss* with 0.578, and $Gauss_1$ with 0.54. Again, *Lowest Bid* delivers the worst performance with just 0.18. To understand these results, notice that, when nodes follow tightness-based strategies, the auction winner is generally a node whose bid value is close to the announced fine F_u (one of the goals of the preference functions, as

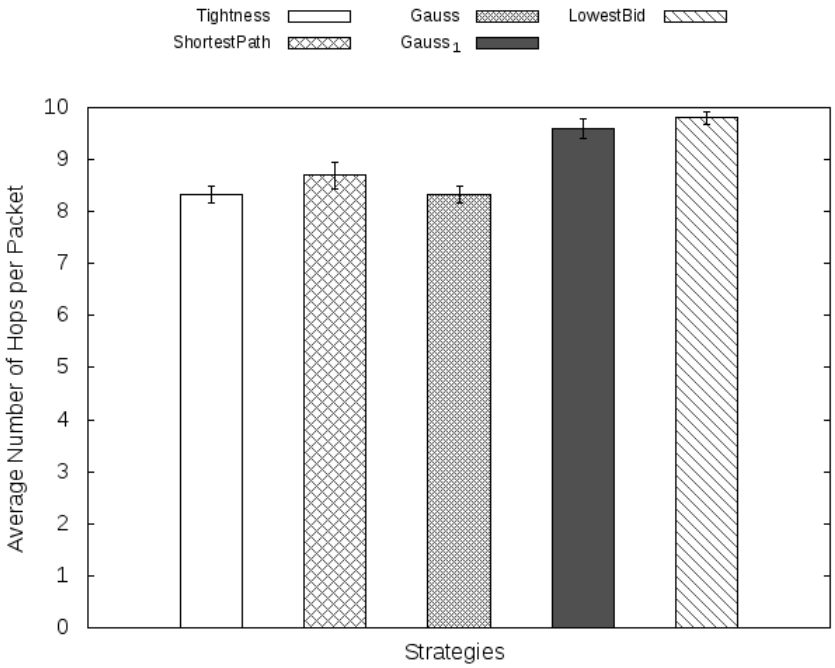

Figure 6.4: Average number of hops per packet under static topologies.

presented in Section 4.2). Thus, because a winner node keeps 5% of its bid before setting up its own budget and fine values, budget gains decay along a route towards destination: the nodes close to the transmitting AP get higher gains than those close to the destination AP. This is actually a reasonable policy, since the nodes close to the transmitting AP assume higher risks early on, with unpredictable outcomes, compared to those close to the destination AP, which have a better assessment of the likelihood of packet delivery within the deadline. Under *Shortest Path*, the budget gains still decay towards destination, since the budget-and-fine setup strategy is the same. But, the bids of the nodes are randomly distributed in $[F_u, B_u]$, and the winner bid is always the one on the shortest path towards destination. Consequently, the winning bid is not necessarily close to F_u, and this leads to larger variations on budget gains for different packets towards the same destination AP. This is why *Shortest Path* achieves higher fairness compared to the other strategies.

It is clear from previous results that *Lowest Bid* performs very poorly because it does not aim to deliver the packet within the deadline. Therefore, it performs even worse under mobile topologies. For this reason, in the following, we omit the results for this strategy.

6.3 Mobile Topologies

Figures 6.6, 6.7 and 6.8 shows the results for the *relative average budget* (RAB) under mobility. Compared to the static case, the RAB values of all strategies decrease as mobility increases, as expected. The most significant decay in performance happens with $Gauss_1$, whose RAB decays by 54.8% just by starting moving at 0.5 m/s. Also, *Tightness* RAB decays by 35.0%, while *Shortest Path* drops by 38.7%, and *Gauss* by 39.6%. As a result, the difference between *Tightness* and

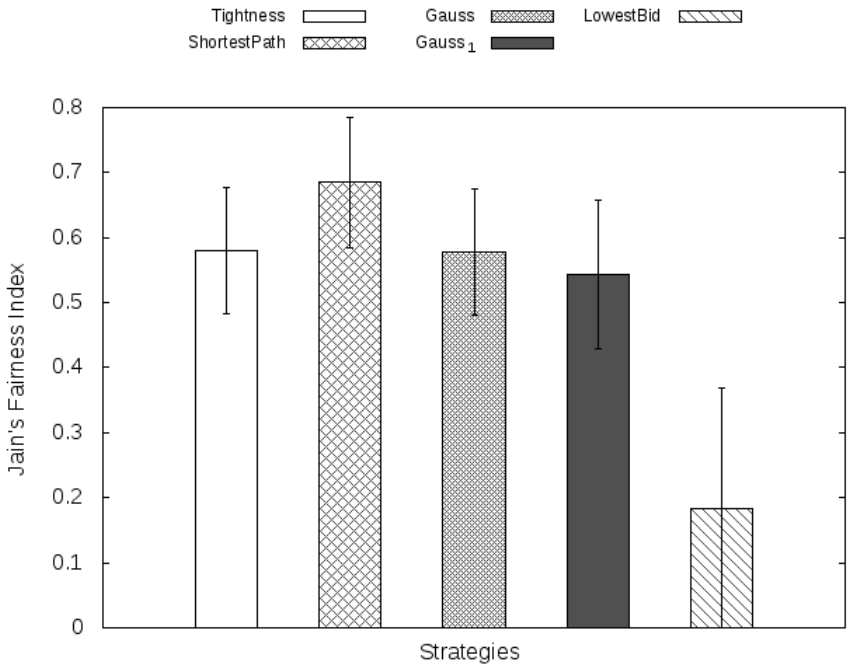

Figure 6.5: Budget fairness under static topologies.

both *Gauss* and $Gauss_1$ increase under mobility. In fact, *Tightness* dominates RAB performance in all speed scenarios. At 0.5 m/s, *Tightness* performance is 7.8% better than *Gauss*, 44.3% better than *Shortest Path*, and 49.2% better than $Gauss_1$. As mobility increases, all tightness strategies surpass *Shortest Path* ($Gauss_1$ is a bit worse than *Shortest Path* at 0.5 m/s). At the speed of 1.0 m/s, *Tightness* RAB decreases to 5.73, which is about 92.93% higher than *Shortest Path* (2.97 RAB), 16.94% higher than *Gauss* (4.90 RAB), and 70.54% better than $Gauss_1$ (3.36 RAB).

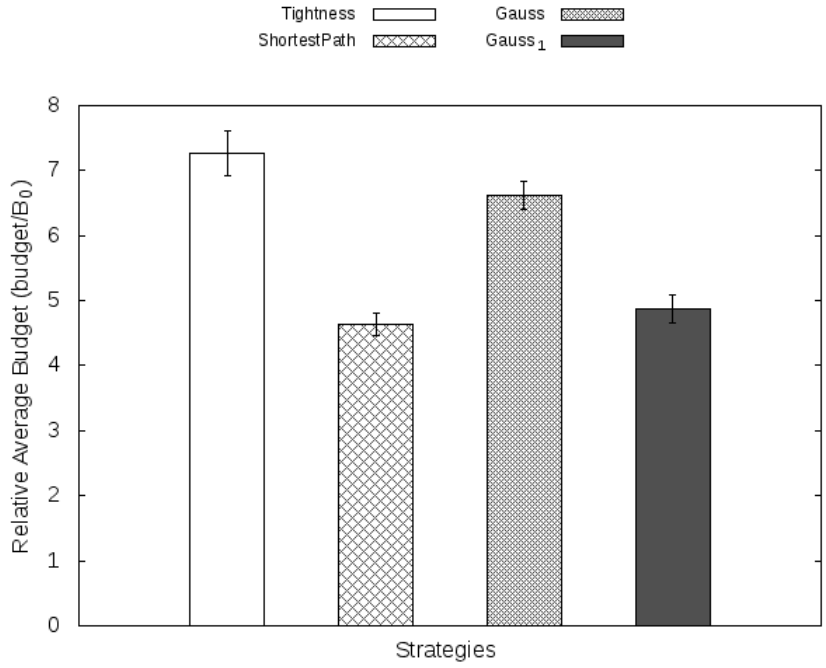

Figure 6.7: Relative average budget per node of each strategy under mobility at 0.75 m/s.

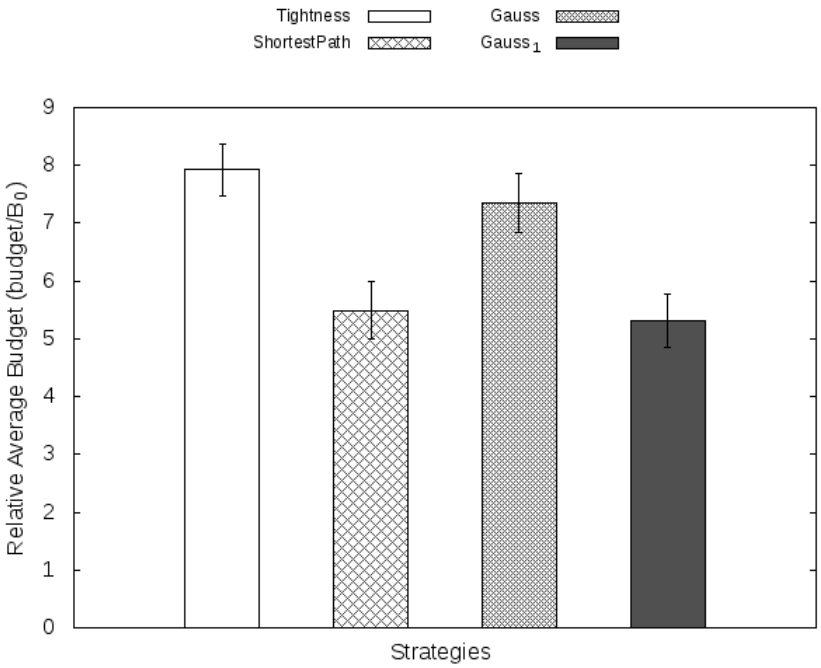

Figure 6.6: Relative average budget per node of each strategy under mobility at 0.5 m/s.

It is interesting to notice that having a preference function that focus on an operating (target) point does not necessarily deliver the best RAB results under mobility. Instead, the *plane* preference function works best. It is worth noting that both the *parameters* and *shape* of the Gauss-like preference functions depend on the specific bid and tightness values of a given auction. Therefore, it seems that the "best" decisions are too localized, which seems to reflect on the overall performance as mobility increases. In the case of the plane preference function, the parameters k_1 and k_2 are kept fixed in every auction performed in the network (we can interpret the ratios op_i/B_n and $c_i/c_{\max}$ as input variables to the plane). Therefore, the same preference function is applied in every single auction. The plane does not define a specific "operation point," based on which a maximum value can be drawn. It simply picks the node whose bid and tightness values lead to the maximum on the plane preference function.

The strength of the "tightness strategies" under mobility is best appreciated if we look at the results for *packet delivery ratio* (PDR) in Figures 6.9, 6.10 and 6.11. Surprisingly, *Tightness* and *Gauss* achieve better PDR than *Shortest Path* and *Gauss*$_1$ when nodes move at all speeds (0.5 m/s, 0.75 m/s and 1 m/s). This is quite interesting, since it means that the "offered price" dimension in the preference function has a positive impact on the achievable PDR. When nodes move at 0.5 m/s, *Tightness* achieves a PDR of 71.92%, while *Shortest Path* reaches 60.10%, *Gauss* 63.80%, and *Gauss*$_1$ 33.36%. In other words, *Tightness* and *Gauss* deliver 19.67% and 6.16% more packets than *Shortest Path*, respectively, while *Gauss*$_1$ is 44.49% worse than *Shortest Path*. When nodes move at 0.75 m/s, the performance of all strategies degrades, but *Tightness* and *Gauss* are 30.25% and 9.54% better than *Shortest Path*, respectively. Still, *Shortest Path* is 85.81% better than *Gauss*$_1$. Finally, as nodes move at 1.0 m/s, performance degrades across all strategies, but *Tightness* delivers 51.57% more packets than *Shortest Path*, while *Gauss* becomes 16.41% better

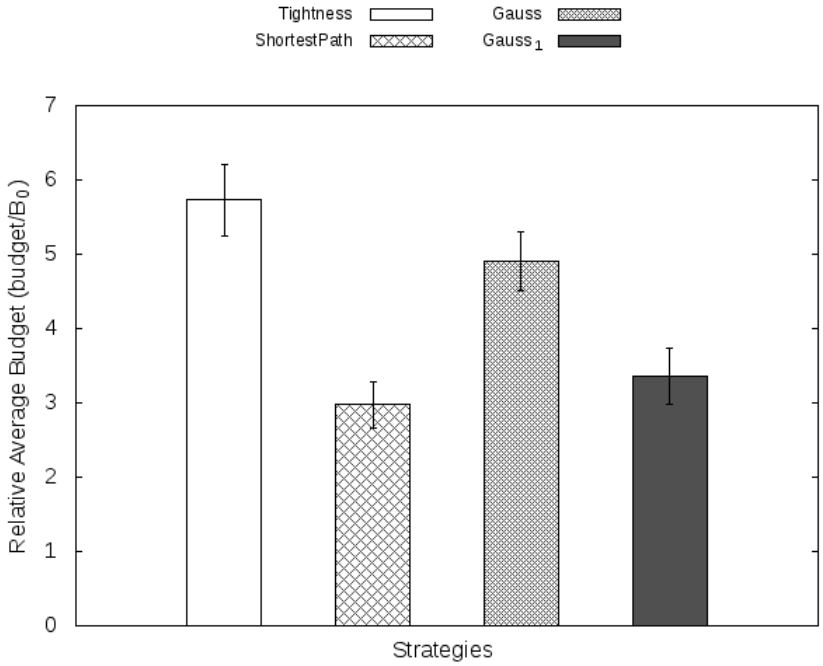

Figure 6.8: Relative average budget per node of each strategy under mobility at 1.0 m/s.

than *Shortest Path*. It is important to remember that all the strategies rely on the information provided by the OLSR protocol. Therefore, as mobility increases, the routing tables at nodes become less reliable, and stale topology control information is disseminated on the network, which reflects on routing decisions (*Shortest Path*) and tightness computations.

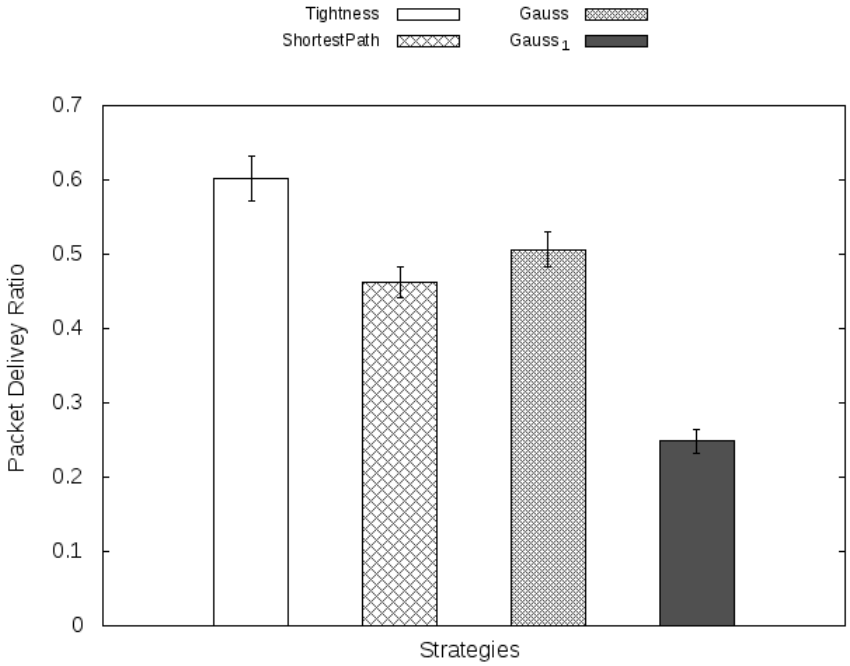

Figure 6.10: Packet delivery ratio of each strategy under mobility at 0.75 m/s.

The PDR results also show that the choice of *operation point* for the bell-shaped preference

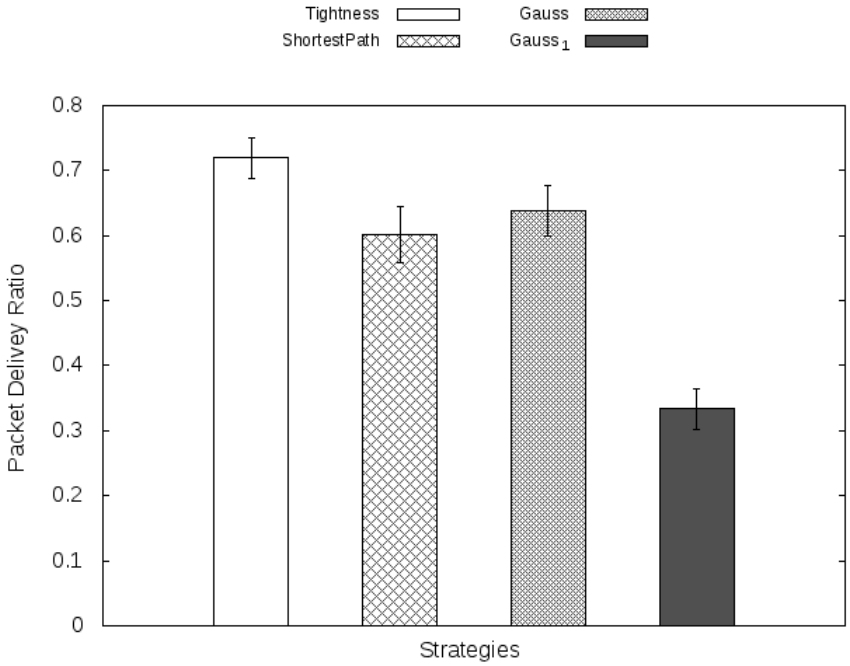

Figure 6.9: Packet delivery ratio of each strategy under mobility at 0.5 m/s.

functions has a clear impact on the final performance. *Gauss* is significantly better than $Gauss_1$ in both RAB and PDR metrics. This means that relaying a packet to a neighbor whose tightness (c_i) is closer to the maximum possible value among competitors ($c_{\max}$) is better than relaying the packet to a node with average tightness ($c_i = 1$) (assuming that in both cases the offered price is close to the minimum possible F_u). It is also worth noting that, in spite of the lower PDR values obtained at 1.0 m/s (compared to *Shortest Path*), $Gauss_1$ deliver higher RAB value than *Shortest Path* at this speed (see Figure 6.11). This means that, although $Gauss_1$ have delivered less packets, the nodes ended up accumulating higher profits. Under high mobility, one should expect a higher reluctance from nodes to participate in the D2D network, because of the likelihood of higher losses in a less predictable and stable environment. Therefore, it is reasonable to trade off PDR with RAB, since the nodes are assuming higher risks (this is certainly not a favorable situation to operators, but it is definitely better to participant nodes in the D2D network—the prospect of some profit under a harsh environment).

Figures 6.12, 6.13 and 6.14 presents the results for the average number of hops (ANH) traversed by successful packets in each strategy. *Tightness*, *Gauss*, and *Shortest Path* present similar results, with *Tightness* delivering the best performance across all speeds. In spite of mobility, *Tightness* manages to deliver packets within 1 to 1.5 hops away from the maximum number of hops allowed to destination (on average). $Gauss_1$ deviates the most from other strategies, delivering slitghtly higher ANH values, especially at low mobility (similar to the static scenario). As mobility increases, *Tightness* ANH values increase from 8.88 (at 0.5 m/s) to 9.11 (at 1.0 m/s), a 2.6% variation. *Gauss* ANH values also increase with speed: from 9.07 (at 0.5 m/s) to 9.26 (at 1.0 m/s), which is a 2.09% variation. Gauss_1, however, is the only strategy whose ANH values *decrease* as mobility increases: from 9.89 (at 0.5 m/s) to 9.61 (at 1.0 m/s), a variation of -2.83%.

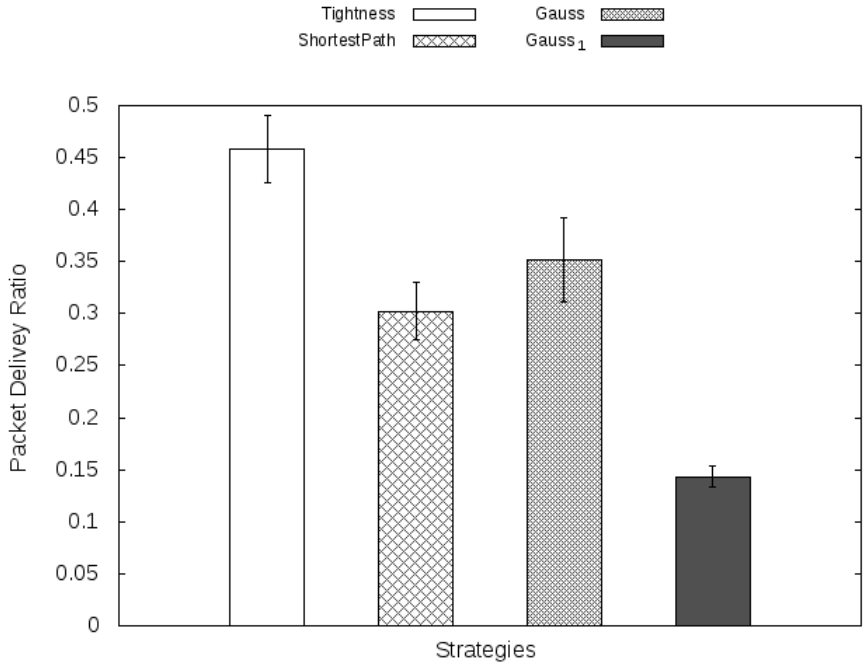

Figure 6.11: Packet delivery ratio of each strategy under mobility at 1.0 m/s.

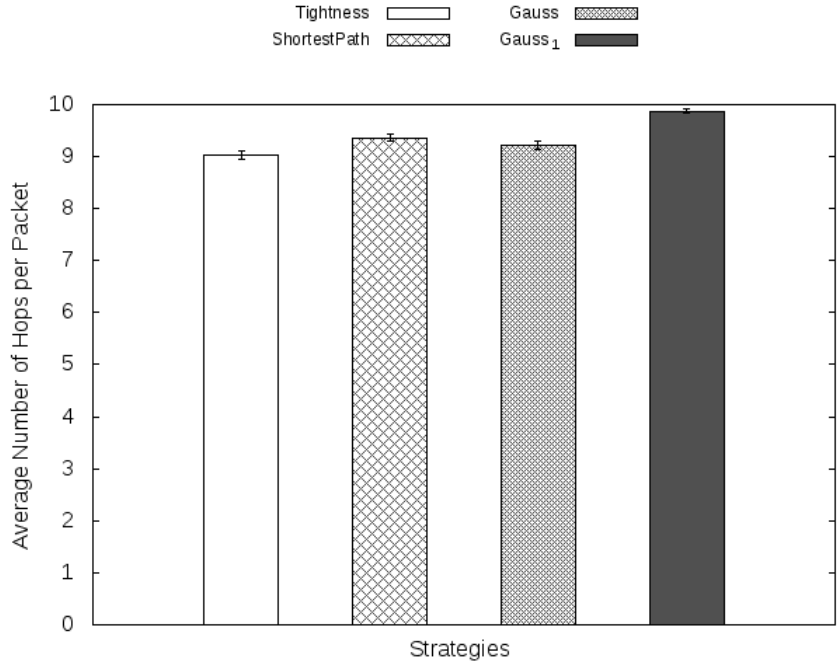

Figure 6.13: Average number of hops per packet of each strategy under mobility at 0.75 m/s.

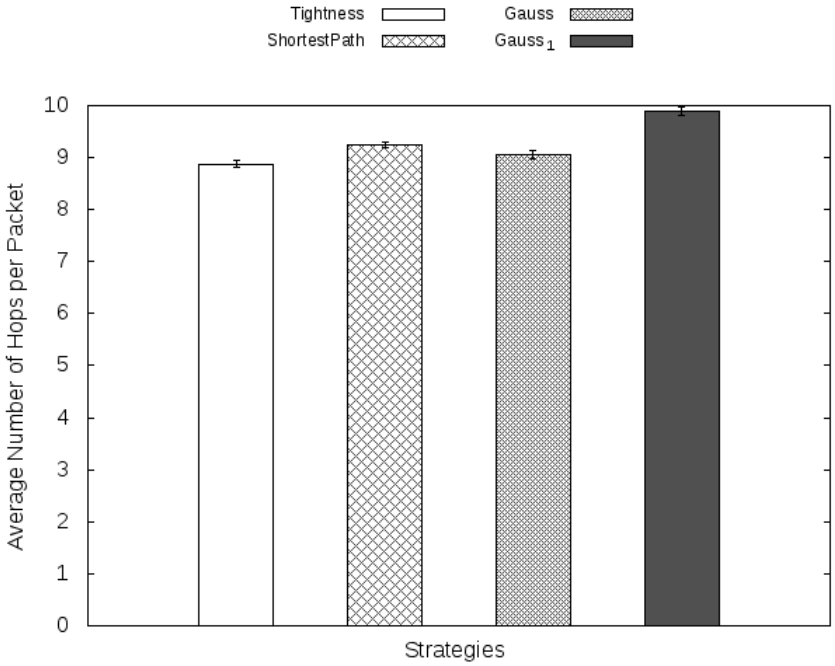

Figure 6.12: Average number of hops per packet of each strategy under mobility at 0.5 m/s.

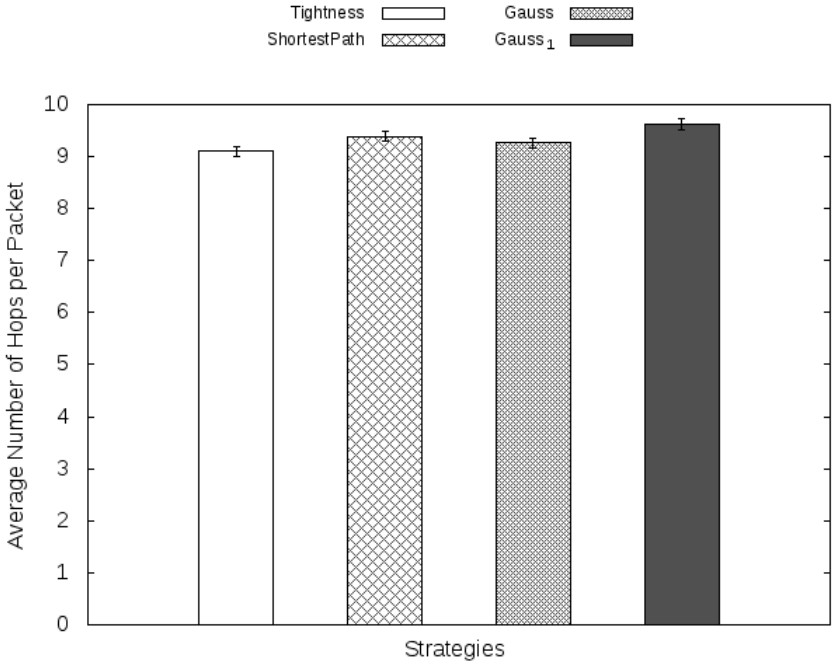

Figure 6.14: Average number of hops per packet of each strategy under mobility at 1.0 m/s.

Lastly, Figures 6.15, 6.16 and 6.17 presents the results on budget *fairness* among nodes under mobility. Following the results on the static scenario, *Shortest Path* has the best performance at speeds of 0.5 m/s and 0.75 m/s, where it achieves an average fairness of 0.73 and 0.74 respectively. This is roughly 20% better than *Gauss* and *Tightness* in both scenarios. This is because, when nodes follow the tightness strategies, the auction winners are those that bid values closer to the

announced fine F_u, as dictated by the bidding and decision strategies presented in Section 4.2. Thus, because the winner node keeps 5% of its bid before setting up its own budget and fine values (see the Budget-and-Fine set up strategy), the accumulated budget drops fast as a packet moves forward along a route (nodes that are closer to the AP gets more, since they assume a task of high risk early on, of unpredictable outcome towards destination, while nodes that are closer to destination have a much better idea of the possible success in the forwarding of a packet. Hence, they should be less rewarded, comparatively). Under *Shortest Path*, however, the auction participants offer random values within the interval $[F_u, B_u]$, and the winner node is always the one on the shortest path towards destination. This leads to a higher variation of accumulated budget along a route. Note that, nodes still obey the Budget-and-Fine set up strategy under *Shortest Path*, but the winner is no longer the one who bids a value close to the announced fine.

As far as resilience to mobility is concerned, *Tightness* presents the best performance, since its fairness varies by only 26.7% as speed changes from 0.5 m/s to 1.0 m/s. *Gauss* comes next, with a 38.3% variation, while *Shortest Path* has a variation of 54.8%, and $Gauss_1$ undergoes a 56.2% variation. In fact, the performance decay of *Shortest Path* and $Gauss_1$ is accentuated when speed changes from 0.75 m/s to 1.0 m/s. Such a significant drop in fairness is probably due to the low PDR of *Shortest Path* in this scenario. At the speed of 1.0 m/s, *Tightness* delivers the best performance, with an average fairness of 0.44, against 0.37 of *Gauss*, 0.33 of *Shortest Path*, and 0.21 of $Gauss_1$.

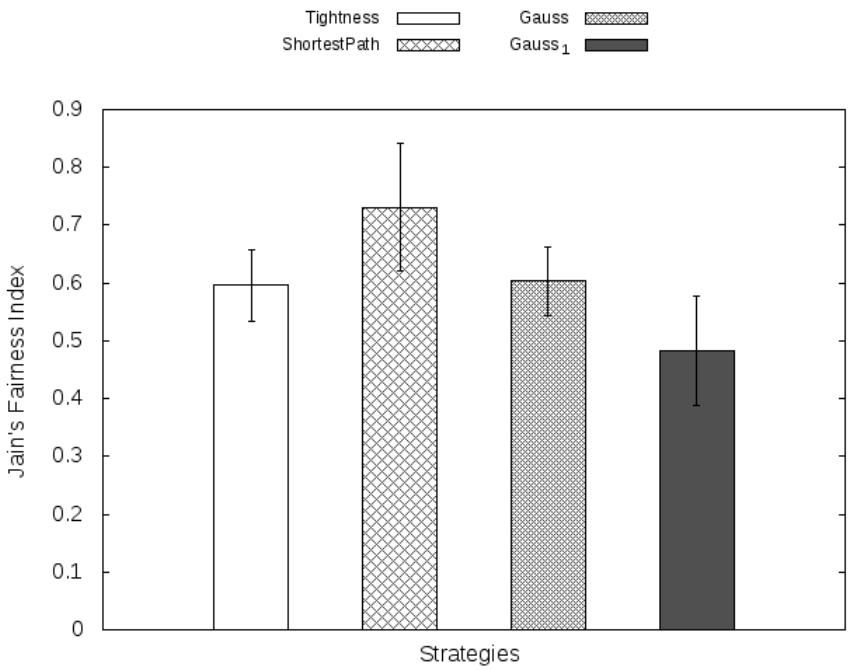

Figure 6.15: Budget fairness of each strategy under mobility at 0.5 m/s.

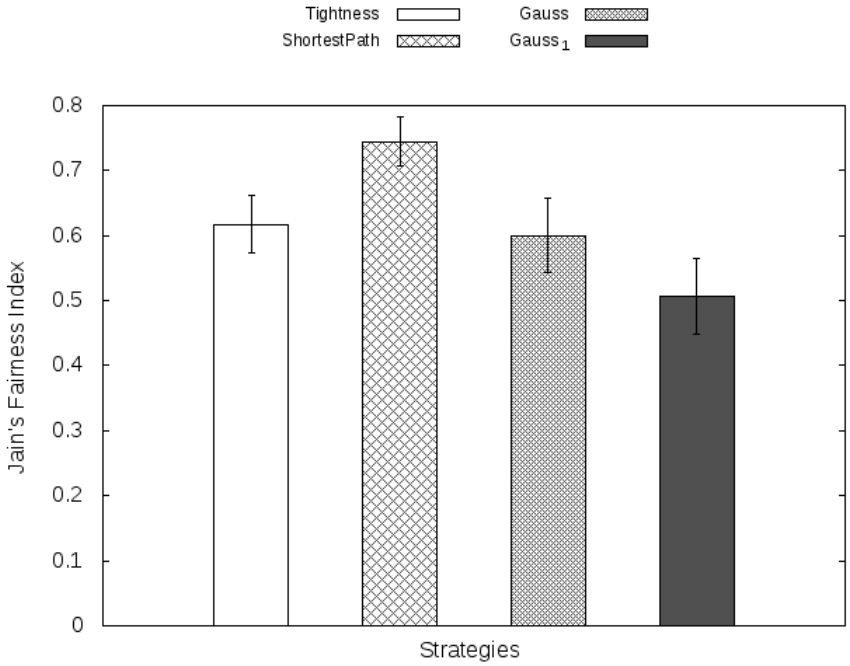

Figure 6.16: Budget fairness of each strategy under mobility at 0.75 m/s.

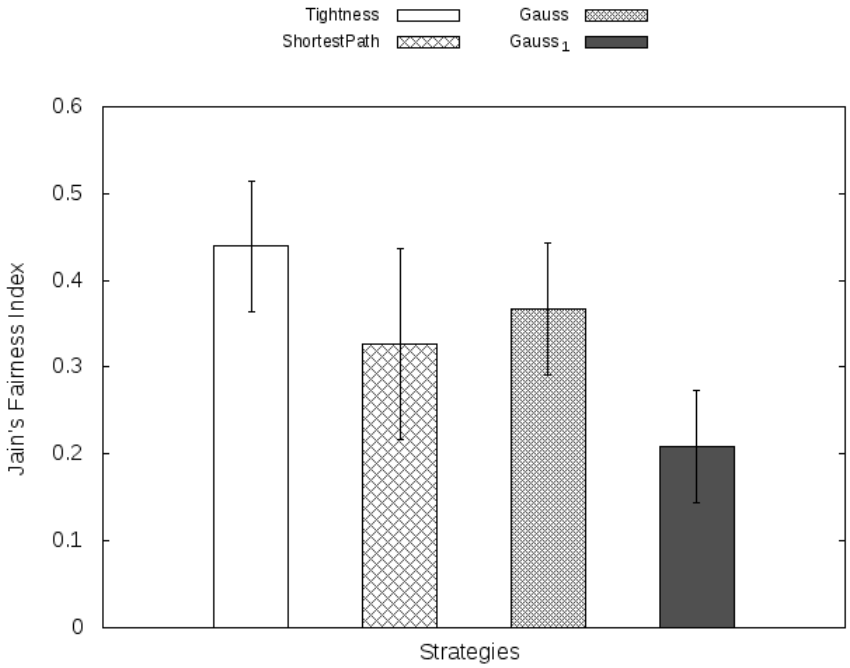

Figure 6.17: Budget fairness of each strategy under mobility at 1.0 m/s.

Chapter 7

Conclusions

This dissertation presented a comprehensive performance evaluation of the so-called *Tightness* strategy for device-to-device data offloading via recursive auctions. This strategy was designed for the offloading scenario where each packet announced by a source AP is associated to a maximum budget (to be shared by the clients) and a fine, which must be paid (recursively) by the nodes if the packet is not delivered to its target destination within the given deadline (translated to a maximum number of hops to destination). The *Tightness* strategy uses the idea of estimating how "tight" a node is to fullfill the job of delivering a packet to its destination within the announced deadline. In other words, a node estimates how much "room" it has (with respect to the deadline) to absorb eventual bad forwarding decisions resulted from the unpredictable outcomes of other downstream auctions. For tightness computations, the nodes in the D2D network rely on topology information (e.g., OLSR protocol). The tightness concept was used in the design of the bidding and decision-making sub-strategies, which take as input parameters the offered price and the relative tightness of the nodes.

The performance of the *Tightness* strategy was investigated for two specific preference functions used in the decision-making substrategy, according to different operating points (leading to three preference functions). Both static and mobile scenarios were investigated, for different node speeds, using discrete-event simulations. Two baseline strategies were also investigated for purposes of performance comparison: one that prioritizes packet delivery over budget gains, (using shortest-path routing), and a greedy one, that always pick the highest bid regardless of packet delivery within the deadline. The performance was carried out in scenarios where all nodes implement the same strategy (i.e., the homogeneous case). All strategies were evaluated for packet delivery ratio, average budget per node, budget fairness, and average number of hops to destination.

The presented results have shown that, apart from the *Lowest Bid* strategy, which delivered very low packet delivery ratios, all strategies proved to be very suitable for D2D data offloading under recursive auctions. Overall, they have provided consistent and positive results across all performance metrics, in both static and mobile scenarios. According to the results, two of the proposed variations of the *Tightness* strategy proved to be more effective than simply using shortest-path routing without taking into account the nodes' bids in the decision-making sub-strategy. In

particular, the preference functions *Hyperplane* and *Gaussian* (the one that prioritizes the lowest bid with the highest relative tightness) performed better than *Shortest Path* with respect to average budget per node, average number of hops to destination, and packet delivery ratio, especially under mobility. This result indicates that the use of the tightness concept in both bidding and decision-making sub-strategies is beneficial for cooperative behavior and better overall performance of D2D offloading under recursive auctions. This happens because the nodes who perceive a "tight" condition to deliver a packet within the announced deadline discourage the auctioneer from choosing them by bidding high values. As far as fairness in budget distribution among nodes is concerned, the *Tightness* strategies delivered slightly lower results than *Shortest Path* due to a higher variation of accumulated budget along a route. However, the *Tightness* strategies presented lower fairness variation across different mobility scenarios.

7.1 Future Work

The work developed in this dissertation can be extended in a number of ways, to address other issues and challenges to be explored in the future. The first of them could be to carry out a performance analysis with a mix of different strategies installed at different nodes. In this dissertation, we have evaluated a homogeneous scenario, where all nodes implement the same strategy. This is a likely scenario if, for instance, the MNOs could had the strategy deployed in all of its clients "as it is", by means of an app that would run on the background without intervention of the user. But, in the case where each client could customize (or implement) its own strategy, the end performance of the Tightness strategy could be totally different. This would reflect the scenario envisioned in the MANIAC Challenge 2013, where each team deployed its own strategy for competition on the challenge. But, as we have pointed out before, a key issue is how to promote cooperation among nodes. Therefore, the study of heterogeneous environments is important because it helps to understand the net result of different strategies deployed for the same task. Additionally, the study of the Tightness strategy itself could still be extended, since different preference functions can be used, as well as different choices for the bidding and budget-and-fine setup strategies. In this dissertation, no alternatives were investigated for these two sub-strategies.

From this work, it would be very simple and useful to observe the impact of changing some important parameters. The Budget-and-Fine set up strategy parameters make a difference at how the budget value decay along the packets' path. So by varying the current parameters (0.95 for the budget and 0.4 for the fine, as explained in Chapter 4) could probably lead to different budget distributions. Other interesting parameters to study would be the deadline, and the impact of many OLSR parameters according to different mobility scenarios. In fact, we believe that the OLSR parameters are strong candidates to increase the offloading performance if they are changed according to the mobility patterns of nodes. The time intervals in the OLSR operation may be too long for our D2D communications under mobility. Depending on the mobility of the nodes, the topology information may not be accurate, and the strategy will make use of unreliable information to make decisions. It would be also very interesting to *add* some new parameters at the strategies, like the *energy* of the participant nodes, the nodes' *reputation*, and other metrics. These new

parameters might be included, for example, at the "tightness" concept.

Another important aspect to consider in future work is the introduction of delay-tolerant techniques at each node for purposes of increasing the packet delivery ratio. For instance, in the current implementation, if a node does not receive any bids after issuing its RFB, it drops the packet without any retransmission of its RFB (no retries for auction the same packet). This situation could happen because of mobility or channel conditions. By using delay-tolerant techniques, the nodes could keep the packet until favorable conditions to execute their auction could be found.

Game Theory has been used to understand and devise efficient algorithms for reversed auctions (such as in our case). In the recursive auction scenario, we have a different game every time a node announces an auction. Both homogeneous and heterogeneous scenarios could be studied. Works such as [15] and [3] could serve as a starting point for that.

Studying other types of data traffic models are also important. In this work, we used a traffic with deterministic characteristics, i.e., with fixed time interval between packets. But, for more realistic scenarios, it would be good to simulate traffic with different statistical behavior. However, the code developed for this work does not allow multi-threading, and it does not handle multiple auctions at the same time. Therefore, each node can only handle one packet at a time. This is a limitation to the time interval between packets at data traffic generation. In the future, this code will also need to have the capacity of processing simultaneous packet auctions, so that the data traffic can be more intense (higher throughputs). Lastly, the impact of different mobility models can also be evaluated. In this work, it was used the Random Walk 2D mobility model of the ns-3. But, there are a lot of other mobility models, such as the Random Waypoint mobility model (ns-3) and some Social/Human walk mobility models [28].

REFERENCES

[1] CISCO. *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update 2014-2019 White Paper*. 2015.

[2] ASADI, A.; WANG, Q.; MANCUSO, V. A survey on device-to-device communication in cellular networks. *IEEE Communications Surveys Tutorials*, v. 16, n. 4, p. 1801–1819, Fourthquarter 2014. ISSN 1553-877X.

[3] IOSIFIDIS, G.; GAO, L.; HUANG, J.; TASSIULAS, L. A double-auction mechanism for mobile data-offloading markets. *Networking, IEEE/ACM Transactions on*, v. 23, n. 5, p. 1634–1647, Oct 2015.

[4] NS-3.17 Manual. https://www.nsnam.org/docs/release/3.17/manual/singlehtml/index.html.

[5] NS-3.17 Model Library. https://www.nsnam.org/docs/release/3.17/models/ns-3-model-library.pdf.

[6] ERICSSON. *Ericsson Mobility Report*. 2014.

[7] BALASUBRAMANIAN, A.; MAHAJAN, R.; VENKATARAMANI, A. Augmenting mobile 3G using WiFi. In: *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*. New York, NY, USA: ACM, 2010. (MobiSys '10), p. 209–222.

[8] DIMATTEO, S.; HUI, P.; HAN, B.; LI, V. Cellular traffic offloading through WiFi networks. In: *Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on*. [S.l.: s.n.], 2011. p. 192–201.

[9] LEE, K.; LEE, J.; YI, Y.; RHEE, I.; CHONG, S. Mobile data offloading: How much can WiFi deliver? *Networking, IEEE/ACM Transactions on*, v. 21, n. 2, p. 536–550, April 2013. ISSN 1063-6692.

[10] HAN, B.; HUI, P.; KUMAR, V.; MARATHE, M.; SHAO, J.; SRINIVASAN, A. Mobile data offloading through opportunistic communications and social participation. *Mobile Computing, IEEE Transactions on*, v. 11, n. 5, p. 821–834, May 2012.

[11] LI, Y.; SU, G.; HUI, P.; JIN, D.; SU, L.; ZENG, L. Multiple mobile data offloading through delay tolerant networks. In: *Proceedings of the 6th ACM Workshop on Challenged Networks*. New York, NY, USA: ACM, 2011. (CHANTS '11), p. 43–48.

[12] WHITBECK, J.; LOPEZ, Y.; LEGUAY, J.; CONAN, V.; AMORIM, M. D. de. Push-and-track: Saving infrastructure bandwidth through opportunistic forwarding. *Pervasive and Mobile Computing*, v. 8, n. 5, p. 682 – 697, 2012. ISSN 1574-1192.

[13] REBECCHI, F.; AMORIM, M. Dias de; CONAN, V.; PASSARELLA, A.; BRUNO, R.; CONTI, M. Data offloading techniques in cellular networks: A survey. *Communications Surveys Tutorials, IEEE*, v. 17, n. 2, p. 580–603, Secondquarter 2015.

[14] ZHUO, X.; GAO, W.; CAO, G.; DAI, Y. Win-coupon: An incentive framework for 3G traffic offloading. In: *Proc. ICNP*. [S.l.: s.n.], 2011. p. 206–215.

[15] GAO, L.; IOSIFIDIS, G.; HUANG, J.; TASSIULAS, L. Economics of mobile data offloading. In: IEEE. *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*. [S.l.], 2013. p. 351–356.

[16] BACCELLI, E.; JURASCHEK, F.; HAHM, O.; SCHMIDT, T. C.; WILL, H.; WAHLISCH, M. The MANIAC challenge at IETF 87. *The IETF Journal*, v. 9, n. 2, p. 27–29, Nov 2013.

[17] CLAUSEN, T.; JACQUET, P.; LAOUITI, A.; MUHLETHALER, P.; QAYYUM, A.; VIENNOT, L. Optimized link state routing protocol. In: *Proc. IEEE National Multi-Topic Conference (INMIC)*. [S.l.: s.n.], 2001.

[18] BACCELLI, E.; JURASCHEK, F.; HAHM, O.; SCHMIDT, T. C.; WILL, H.; WAHLISCH, M. Proceedings of the 3rd MANIAC challenge. In: . [s.n.], 2013. Available from Internet: <http://arxiv.org/html/1401.1163v2>.

[19] 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.

[20] The ns-3 Network Simulator. http://www.nsnam.org.

[21] BUTTYAN, L.; HUBAUX, J.-P. *Nuglets: a Virtual Currency to Stimulate Cooperation in Self-Organized Mobile Ad Hoc Networks*. Lausanne, January 2001.

[22] PARK, V. D.; CORSON, M. S. A highly adaptive distributed routing algorithm for mobile wireless networks. In: *Proc. IEEE INFOCOM*. [S.l.: s.n.], 1997. v. 3, p. 1405–1413.

[23] ANDEREGG, L.; EIDENBENZ, S. Ad hoc-VCG: A truthful and cost-efficient routing protocol for mobile ad hoc networks with selfish agents. In: *Proc. ACM MobiCom*. [S.l.: s.n.], 2003. p. 245–259.

[24] LUO, H.; MENG, X.; RAMJEE, R.; SINHA, P.; LI, L. The design and evaluation of unified cellular and ad-hoc networks. *Mobile Computing, IEEE Transactions on*, v. 6, n. 9, p. 1060–1074, Sept 2007.

[25] YU, T.; ZHOU, Z.; ZHANG, D.; WANG, X.; LIU, Y.; LU, S. INDAPSON: An incentive data plan sharing system based on self-organizing network. In: *Proc. INFOCOM*. [S.l.: s.n.], 2014. p. 1545–1553.

[26] KOUTSOPOULOS, I.; NOUTSI, E.; IOSIFIDIS, G. Dijkstra goes social: Social-graph-assisted routing in next generation wireless networks. In: *Proc. European Wireless 2014*. [S.l.: s.n.], 2014. p. 1–7.

[27] JAIN, R.; HAWE, W. R.; CHIU, D.-M. *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.

[28] RHEE, I.; SHIN, M.; HONG, S.; LEE, K.; KIM, S. J.; CHONG, S. On the levy-walk nature of human mobility. *IEEE/ACM Trans. Netw.*, IEEE Press, Piscataway, NJ, USA, v. 19, n. 3, p. 630–643, jun. 2011.

APPENDIX

I. NS-3 DEVELOPED CODE

I.1 Application Module

I.1.1 offloading.h

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright 2014 Universidade de BrasÃηlia, Brazil
 *
 * offloading.
 *
 *  Created on: 06/01/2014
 *      Author: Lucas Soares de Brito
 *
 * 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
 */

#ifndef OFFLOADING_H_
#define OFFLOADING_H_

#include "offloading-packet.h"
//#include "dijkstra.h"
#include "ns3/application.h"
#include "ns3/event-id.h"
#include "ns3/ptr.h"
#include "ns3/ipv4-address.h"
#include "ns3/traced-callback.h"
#include "ns3/address.h"

#include <vector>
#include <iostream>
#include <string>
#include <list>
#include <limits> // for numeric_limits
#include <set>
#include <utility> // for pair
#include <algorithm>
#include <iterator>

namespace ns3 {

class Socket;
class Packet;

enum StrategyType
{
  STRATEGYTYPE_DUMMYBID  = 1,   //!< STRATEGYTYPE_DUMMYBID
  STRATEGYTYPE_DUMMYPATH  = 2,   //!< STRATEGYTYPE_DUMMYPATH
  STRATEGYTYPE_TIGHTNESS  = 3,   //!< STRATEGYTYPE_TIGHTNESS
};

enum PreferenceFunctionType
{
PREFERENCEFUNCTION_PLANE = 1, //!< PREFERENCEFUNCTION_PLANE
PREFERENCEFUNCTION_GAUSS = 2, //!< PREFERENCEFUNCTION_GAUSS
PREFERENCEFUNCTION_GAUSS_1 = 3, //!< PREFERENCEFUNCTION_GAUSS_1 (with cn=1)
};

enum NodeType
{
```

```
  NODETYPE_AP  = 1,   //!< NODETYPE_AP
  NODETYPE_NETWORK  = 2,   //!< NODETYPE_NETWORK
};

typedef int vertex_t;
typedef double weight_t;

struct neighbor {
  vertex_t target;
  weight_t weight;
  neighbor(vertex_t arg_target, weight_t arg_weight)
       : target(arg_target), weight(arg_weight) { }
};

typedef std::vector<std::vector<neighbor> > adjacency_list_t;

/**
 * \ingroup offloading
 * \brief An Offloading app
 *
 * Every packet sent should be returned by the server and received here.
 */
class Offloading : public Application
{
public:
  static TypeId GetTypeId (void);
  Offloading ();
  virtual ~Offloading ();

  void SetRemote (Address ip, uint16_t port);
  void SetRemote (Ipv4Address ip, uint16_t port);
  void SetBackbone (std::vector<Ipv4Address > backbone);
  void SetTightnessParameters (double k1, double k2, double budget_percentage, double fine_percentage);
  void SetMapNodes (std::map<Ipv4Address, int> nodes_id);
  void SetTopologyName (std::string topologyName);
  void SetSeedIndex (int seedIndex);
  void SetParamName (std::string paramName);
  void SetExpIndex (std::string expIndex);

protected:
  virtual void DoDispose (void);

private:

  virtual void StartApplication (void);
  virtual void StopApplication (void);

  void SendData (Ipv4Address source, Ipv4Address destination, int dataEventNumber, bool toBackbone);
  void SendBid(Ptr<Socket> socket, uint32_t Bu, uint32_t Fu, uint8_t HO, int packetID);
  void BufferBid(Address from, uint32_t bid, Ipv4Address source, Ipv4Address destination);
  void SendRFB(Ipv4Address source, Ipv4Address destination, int packetID);

  void ScheduleSendFirstRFB (Time dt);
  void ScheduleSendData (Time dt);
  bool IsLastHop (Ipv4Address dest);
  //void SendLastHopData(void);
  bool isBackbone(Ipv4Address ipv4address);
  bool haveNeighborBackbone(Ipv4Address dest, Ipv4Address src);
  bool isOldPacket(Ipv4Address source, int packetID);
  //int FindBidNumber(Ipv4Address source, Ipv4Address destination);

  void HandleRead (Ptr<Socket> socket);
  void ImprimeTopologySet (void);
  void WritePacket(int sourceID, int pktID, int nextID, int status, double reserved);

  void HopCountComputation (Ipv4Address dest, int nodetype);
  void BidComputation (uint32_t Bu, uint32_t Fu, uint8_t HO);

  void PopulateArpCache (void);

  void MapNode (void);
  void BuildGraphDijkstra (adjacency_list_t &grafo_dijkstra);
  void ImprimeGrafo (bool sort, adjacency_list_t &grafo_dijkstra);
  bool FindGraphEdge (int a, int b, adjacency_list_t &grafo_dijkstra);
  Ipv4Address FindMapAddress (int a);
  void DijkstraComputePaths(vertex_t source,
                            const adjacency_list_t &adjacency_list,
                              std::vector<weight_t> &min_distance,
                              std::vector<vertex_t> &previous);
  std::list<vertex_t> DijkstraGetShortestPathTo(vertex_t vertex, const std::vector<vertex_t> &previous);
```

```
  double m_offerBid;
  double m_Bn;
  double m_Fn;
  double m_payFINE, m_winner_bid;
  uint8_t m_deadline;

  //Tightness strategy parameters:
  double m_k1, m_k2, m_budget_percentage, m_fine_percentage;

  //Hop count and delta:
  std::vector<int> m_hc;
  std::vector<int> m_delta, m_able_nodes;
  uint8_t m_pu;

  int m_nPackets;
  Ipv4Address m_dest, m_source;
  //std::vector<std::pair<Ipv4Address, Ipv4Address> > m_backbonePair;//Vetor de leilÃµes que estÃčo ocorrendo simultaneamente neste nÃş
  Ipv4Address m_nodeAddr;
  Mac48Address m_nodeMacAddr;
  Address m_upload;
  Mac48Address m_uploadMAC;
  Ipv4Address m_dropDest;

  double m_tempo_envio;
  double m_start;
  std::vector<std::pair<Ipv4Address, int> > m_loopControl;//Fila de leilÃµes que jÃą terminaram neste nÃş
  int m_pkt_count;
  double m_timeout;
  double m_timeBetweenSourcePacket;

  Ptr<Node> m_thisNode;
  std::vector<uint32_t > m_cn_buffer;
  std::vector<uint32_t > m_bid_buffer;
  std::vector<Address > m_address_buffer;
  std::vector<Ipv4Address > m_source_buffer;
  std::vector<Ipv4Address > m_destination_buffer;
  std::vector<Ipv4Address > m_backbone;
  std::vector<Ipv4Address > m_hc_address;


  std::map<Ipv4Address, int> m_map_address_graphnode, m_nodes_id;

  Ptr<Socket> m_socket;
  Address m_peerAddress;
  uint16_t m_peerPort;
  EventId m_send_AP_rfbEvent;
  EventId m_send_dataEvent;
  EventId m_send_bidEvent;

  //Callbacks for tracing the packet Tx and Rx events:
  TracedCallback<Ptr<const Packet> > m_txTrace;
  TracedCallback<Ptr<const Packet>, const Address &> m_rxTrace;

  //Lado "Servidor":
  uint16_t m_port;
  Address m_local;

  uint16_t node_type, strategy_type, preference_function_type;
  int number_nodes;

  //Create files to report auctions and budgets:
  FILE *report_user;
  std::string dir_myreport_user, filename_report_user, content, m_topologyName, m_paramName;
  int m_seedIndex;
  std::string m_expIndex;

  int m_wrong_order;

};

} // namespace ns3

#endif /* OFFLOADING_H_ */
```

I.1.2 offloading.cc

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright 2014 Universidade de BrasÃηlia, Brazil
```

```
 *
 * offloading.cc
 *
 *  Created on: 06/01/2014
 *      Author: Lucas Soares de Brito
 *
 * 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
 */

#include <iomanip>
#include <cmath>
#include "ns3/random-variable.h"
#include "ns3/log.h"
#include "ns3/ipv4.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/ipv4-address.h"
#include "ns3/mac48-address.h"
#include "ns3/wifi-net-device.h"
#include "ns3/nstime.h"
#include "ns3/inet-socket-address.h"
#include "ns3/socket.h"
#include "ns3/simulator.h"
#include "ns3/socket-factory.h"
#include "ns3/packet.h"
#include "ns3/uinteger.h"
#include "ns3/trace-source-accessor.h"
#include "ns3/address-utils.h"
#include "ns3/udp-socket.h"
#include "ns3/olsr-routing-protocol.h"
#include "ns3/object-vector.h"
#include "ns3/pointer.h"
#include "offloading.h"
#include <vector>
#include <iostream>
#include <map>
#define _USE_MATH_DEFINES
#include <math.h>
#include <string>
#include <fstream>
#include <boost/lexical_cast.hpp>
#include <sys/stat.h>


namespace ns3 {

NS_LOG_COMPONENT_DEFINE ("OffloadingApplication");
NS_OBJECT_ENSURE_REGISTERED (Offloading);

TypeId
Offloading::GetTypeId (void)
{
static TypeId tid = TypeId ("ns3::Offloading")
     .SetParent<Application> ()
     .AddConstructor<Offloading> ()
         .AddAttribute ("NPackets",
          "Number of packets that this node will generate (if APType = 1).",
              IntegerValue(0),
          MakeIntegerAccessor (&Offloading::m_nPackets),
          MakeIntegerChecker<int> ())
     .AddAttribute ("StrategyType",
     "The type of the strategy: "
     " STRATEGYTYPE_DUMMYBID (1) = Lowest Bid"
     " STRATEGYTYPE_DUMMYPATH (2) = Shortest Path Dummy"
     " STRATEGYTYPE_TIGHTNESS  (3) = Tightness",
     UintegerValue(STRATEGYTYPE_TIGHTNESS),
     MakeUintegerAccessor (&Offloading::strategy_type),
     MakeUintegerChecker<uint16_t> ())
          .AddAttribute ("PreferenceFunctionType",
          "The type of the Preference Function: "
```

```
          " PREFERENCEFUNCTION_PLANE (1) = A simple plane P = K1 - (K1/Bn)*bid + (K2/cn_max)*cn;"
          " PREFERENCEFUNCTION_GAUSS (2) = The same graph as the 2-dimensional Gaussian distribution, centered at [Fn,c_max]",
          UintegerValue(PREFERENCEFUNCTION_PLANE),
          MakeUintegerAccessor (&Offloading::preference_function_type),
          MakeUintegerChecker<uint16_t> ())
      .AddAttribute ("DestinationAddress",
      "The destination Address of the packets offloaded",
      Ipv4AddressValue ("10.0.0.25"),
      MakeIpv4AddressAccessor (&Offloading::m_dest),
      MakeIpv4AddressChecker ())
      .AddAttribute ("RemotePort",
      "The destination port of the outbound packets",
      UintegerValue (0),
      MakeUintegerAccessor (&Offloading::m_peerPort),
      MakeUintegerChecker<uint16_t> ())
      .AddAttribute ("Budget",
      "The initial Budget B0",
      DoubleValue (100.0),
      MakeDoubleAccessor (&Offloading::m_Bn),
      MakeDoubleChecker<double> ())
      .AddAttribute ("Fine",
      "The initial Fine F0",
      DoubleValue (50.0),
      MakeDoubleAccessor (&Offloading::m_Fn),
      MakeDoubleChecker<double> ())
      .AddAttribute ("Deadline",
      "The Deadline H0 to reach the destination",
      UintegerValue (10),
      MakeUintegerAccessor (&Offloading::m_deadline),
      MakeUintegerChecker<uint8_t> ())
      .AddAttribute ("StartOffloading",
      "Time (seconds) to initiate packet generation.",
      DoubleValue (0.),
      MakeDoubleAccessor (&Offloading::m_start),
      MakeDoubleChecker<double> ())
      .AddAttribute ("NumberNodes",
      "Number of nodes using OLSR. Used to build the Topology Graph in the Tightness strategy.",
      IntegerValue(100),
      MakeIntegerAccessor (&Offloading::number_nodes),
      MakeIntegerChecker<int> ())
      .AddTraceSource ("Tx", "A new packet is created and is sent",
      MakeTraceSourceAccessor (&Offloading::m_txTrace))
     .AddTraceSource ("Rx", "A packet has been received",
      MakeTraceSourceAccessor (&Offloading::m_rxTrace))

      ;
return tid;
}

Offloading::Offloading ()
{
NS_LOG_FUNCTION (this);
m_socket = 0;
m_send_AP_rfbEvent = EventId ();
m_send_dataEvent = EventId ();
m_send_bidEvent = EventId ();
m_tempo_envio = 0.0;
m_timeout = 0.050;
m_timeBetweenSourcePacket = 3.0;
m_pkt_count = 0;
m_pu = 0;
m_offerBid = 0.0;
m_winner_bid = 0.0;
m_peerAddress = Ipv4Address("255.255.255.255");
filename_report_user = "report_user";
m_wrong_order = 0;

}

Offloading::~Offloading()
{
NS_LOG_FUNCTION (this);
m_socket = 0;
}

void
Offloading::SetRemote (Address ip, uint16_t port)
{
NS_LOG_FUNCTION (this << ip << port);
m_peerAddress = ip;
m_peerPort = port;
```

```
}

void
Offloading::SetRemote (Ipv4Address ip, uint16_t port)
{
NS_LOG_FUNCTION (this << ip << port);
m_peerAddress = Address (ip);
m_peerPort = port;
}

void
Offloading::SetBackbone (std::vector<Ipv4Address > backbone)
{
m_backbone = backbone;
}

void
Offloading::SetTightnessParameters (double k1, double k2, double budget_percentage, double fine_percentage)
{
m_k1 = k1;
m_k2 = k2;
m_budget_percentage = budget_percentage;
m_fine_percentage = fine_percentage;
}

void
Offloading::SetMapNodes (std::map<Ipv4Address, int> nodes_id)
{
m_nodes_id = nodes_id;
}

void
Offloading::SetTopologyName (std::string topologyName)
{
m_topologyName = topologyName;
}

void
Offloading::SetSeedIndex (int seedIndex)
{
m_seedIndex = seedIndex;
}

void
Offloading::SetParamName (std::string paramName)
{
m_paramName = paramName;
}

void
Offloading::SetExpIndex (std::string expIndex)
{
m_expIndex = expIndex;
}

void
Offloading::DoDispose (void)
{
NS_LOG_FUNCTION (this);
Application::DoDispose ();
}

void
Offloading::StartApplication (void)
{
NS_LOG_FUNCTION (this);


// Saving this node address:
m_thisNode = this->GetNode();
Ptr<WifiNetDevice> netDevice = DynamicCast<WifiNetDevice> (m_thisNode->GetDevice(0));
m_nodeMacAddr = Mac48Address::ConvertFrom (netDevice->GetAddress());
Ptr<Ipv4> ipv4 = m_thisNode->GetObject<Ipv4>();
Ipv4InterfaceAddress iaddr = ipv4->GetAddress(1,0);
m_nodeAddr = iaddr.GetLocal();
NS_LOG_INFO("ADDRESS = " << m_nodeAddr);

//Accountability report file name and directory:
if(m_thisNode->GetId() < 10)
filename_report_user = filename_report_user + "00" + boost::lexical_cast<std::string>(m_thisNode->GetId());
else if(m_thisNode->GetId() < 100)
```

```
filename_report_user = filename_report_user + "0" + boost::lexical_cast<std::string>(m_thisNode->GetId());
else
filename_report_user = filename_report_user + boost::lexical_cast<std::string>(m_thisNode->GetId());

std::string home_dir(getenv("HOME"));
if((strategy_type == STRATEGYTYPE_TIGHTNESS) && (preference_function_type == PREFERENCEFUNCTION_PLANE)){
dir_myreport_user = home_dir + "/Dropbox/unb/mestrado/tese/simulations/topology_config/exp" + m_expIndex + "/" + m_paramName + "/seed"
 + boost::lexical_cast<std::string>(m_seedIndex) + "/tp" + m_topologyName + "/report_user";
}
else{
dir_myreport_user = home_dir + "/Dropbox/unb/mestrado/tese/simulations/topology_config/exp" + m_expIndex + "/seed"
 + boost::lexical_cast<std::string>(m_seedIndex) + "/tp" + m_topologyName + "/report_user";
}
mkdir (dir_myreport_user.c_str(),0777);
dir_myreport_user = dir_myreport_user + "/" + filename_report_user + ".txt";

//Report table header:
content = content + "Source" + "\t";
content = content + "pktID" + "\t";
content = content + "Next" + "\t";
content = content + "eBID" + "\t";
content = content + "pFINE" + "\t";
content = content + "pBID" + "\t";
content = content + "eFINE" + "\t";
content = content + "sBUDGET" + "\t";
content = content + "fBUDGET" + "\t";
content = content + "status" + "\t";
content = content + "BALANCE" + "\t";
content = content + "accumulative" + "\t";
content = content + "\n";

NS_LOG_INFO("->" << m_nodeAddr);
report_user = fopen(dir_myreport_user.c_str(), "w+");
fputs(content.c_str(), report_user);
fclose(report_user);

//Identify if this node is an AP or not:
node_type = NODETYPE_NETWORK;
if(isBackbone(m_nodeAddr))
node_type = NODETYPE_AP;

//Configure socket:
TypeId tid = TypeId::LookupByName ("ns3::UdpSocketFactory");
if (m_socket == 0){
m_socket = Socket::CreateSocket (GetNode (), tid);

if (Ipv4Address::IsMatchingType(m_peerAddress) == true)
{
int status;
InetSocketAddress src = InetSocketAddress (Ipv4Address::GetAny (), m_peerPort);
status = m_socket->Bind (src);
NS_ASSERT (status != -1);
InetSocketAddress dst = InetSocketAddress (Ipv4Address::ConvertFrom(m_peerAddress), m_peerPort);
status = m_socket->Connect (dst);
NS_ASSERT (status != -1);

m_socket->SetAllowBroadcast(true);
}
}

m_socket->SetRecvCallback (MakeCallback (&Offloading::HandleRead, this));

//Graph Mapping: node number --> address
MapNode();

if((node_type == NODETYPE_AP)&&(m_nPackets > 0)){
m_pu = 0;
m_send_AP_rfbEvent = Simulator::Schedule (Seconds (m_start), &Offloading::SendRFB, this, m_nodeAddr, m_dest, 0);
}
}

void
Offloading::StopApplication ()
{
NS_LOG_FUNCTION (this);

if (m_socket != 0)
{
m_socket->Close ();
m_socket->SetRecvCallback (MakeNullCallback<void, Ptr<Socket> > ());
m_socket = 0;
```

```
}

Simulator::Cancel (m_send_AP_rfbEvent);
Simulator::Cancel (m_send_dataEvent);
Simulator::Cancel (m_send_bidEvent);
}

void
Offloading::HandleRead (Ptr<Socket> socket)
{
NS_LOG_FUNCTION (this << socket);

NS_LOG_INFO("--> PACKET RECEIVED BY: " << m_nodeAddr);

RFBHeader rfbHeader;
BidHeader bidHeader;
DataHeader dataHeader;

Ptr<Packet> packet;
Address from;

double getOfferbid;
int packetID;
bool toBackbone;
//int bid_index = 0;

while ((packet = socket->RecvFrom (from)))
{
TypeHeader tHeader (OFFLOADINGTYPE_RFB);
packet->RemoveHeader (tHeader);

if (!tHeader.IsValid ())
{
NS_LOG_INFO ("Offloading message " << packet->GetUid () << " with unknown type received: " << tHeader.Get () << ". Drop");
return; // drop
}

if (InetSocketAddress::IsMatchingType (from)){
switch(tHeader.Get()){

case OFFLOADINGTYPE_RFB:
//If this node is an AP, so don't process the RFB (backbone cannot participate on the biddings):
if(node_type == NODETYPE_NETWORK){
//Extract packet:
packet->RemoveHeader (rfbHeader);
m_dest = rfbHeader.GetDst();
m_source = rfbHeader.GetSrc();
packetID = rfbHeader.GetPacketID();
m_pu = rfbHeader.GetHopcount();
m_deadline = rfbHeader.GetDeadline();
m_upload = from;
m_uploadMAC = rfbHeader.GetSrcRFB();
m_payFINE = rfbHeader.GetFine()/pow(10,2); //In case of packet delivery failure, paid this Fine;

//If this packet was already auctioned, do nothing. Otherwise, proceed with the bidding process:
if(!isOldPacket(m_source, packetID)){
//Save the bids that this node is participating at this moment:
//m_backbonePair.push_back(std::make_pair(rfbHeader.GetSrc(),rfbHeader.GetDst()));
//bid_index = FindBidNumber(rfbHeader.GetSrc(), rfbHeader.GetDst());

NS_LOG_INFO ("["<< m_source <<"->"<< m_dest <<"]["<<packetID<<"]At " << Simulator::Now ().GetSeconds () << "s "<< Ipv4Address::ConvertFrom (m_nodeAddr) <<
" received a RFB from " << InetSocketAddress::ConvertFrom (from).GetIpv4 () );

m_send_bidEvent = Simulator::Schedule (Seconds (0.), &Offloading::SendBid, this,
socket, rfbHeader.GetBO(), rfbHeader.GetFine(), rfbHeader.GetDeadline(), packetID);

//SendBid(socket, rfbHeader.GetBO(), rfbHeader.GetFine(), rfbHeader.GetDeadline());
}
}
break;

case OFFLOADINGTYPE_BID:
packet->RemoveHeader (bidHeader);

getOfferbid = bidHeader.GetOfferedBid()/pow(10,2);
packetID = bidHeader.GetPacketID();

//bid_index = FindBidNumber(bidHeader.GetSrc(), bidHeader.GetDst());
NS_LOG_INFO ("["<< bidHeader.GetSrc() <<"->"<< bidHeader.GetDst() <<"]["<<packetID<<"]At " << Simulator::Now ().GetSeconds () << "s "<<
 Ipv4Address::ConvertFrom (m_nodeAddr) <<
" received a BID from " << InetSocketAddress::ConvertFrom (from).GetIpv4 () <<
```

```
" Offered Bid = " << getOfferbid);
BufferBid(from, bidHeader.GetOfferedBid(), bidHeader.GetSrc(), bidHeader.GetDst());

break;

case OFFLOADINGTYPE_DATA:
//DATA received.
packet->RemoveHeader (dataHeader);
packetID = dataHeader.GetPacketID();
m_pu = dataHeader.GetHopcount() + 1; //Update 'pu' (hop count until here)
//bid_index = FindBidNumber(dataHeader.GetSrc(), dataHeader.GetDst());

NS_LOG_INFO ("["<< dataHeader.GetSrc() <<"->"<< dataHeader.GetDst() <<"]["<<packetID<<"]At "
<< Simulator::Now ().GetSeconds () << "s "<< Ipv4Address::ConvertFrom (m_nodeAddr) <<
" received a DATA from " << InetSocketAddress::ConvertFrom (from).GetIpv4 ());

if(isBackbone(m_nodeAddr)){//If this node is an AP:
if(m_nodeAddr == dataHeader.GetDst()){
if(m_pu <= m_deadline){
//If the packet arrived to the correct destination within deadline:
NS_LOG_INFO("-----------------------------------------------------------------------------------------");
NS_LOG_INFO("SUCCESSFULL PACKET [" << packetID << "] RECEIVED AT AP[" << dataHeader.GetDst() << "] FROM AP[" << dataHeader.GetSrc() << "]
 - Hop Count 'pu' = " << static_cast<int>(m_pu) << ".");
NS_LOG_INFO("-----------------------------------------------------------------------------------------");

//WRITE SUCCESSFULL PACKET:
int pktStatus = 1;
int sourceID = m_nodes_id[Ipv4Address::ConvertFrom(dataHeader.GetSrc())];
int nextID = -1;
WritePacket(sourceID, packetID, nextID, pktStatus, 0.0);
}
else{
//If the packet arrived to the correct destination, but after the deadline:
NS_LOG_INFO("-----------------------------------------------------------------------------------------");
NS_LOG_INFO("(ALMOST) SUCCESSFULL PACKET [" << packetID << "] RECEIVED AT AP[" << dataHeader.GetDst() << "] FROM AP[" << dataHeader.GetSrc() << "]
 - Hop Count 'pu' = " << static_cast<int>(m_pu) << ".");
NS_LOG_INFO("-----------------------------------------------------------------------------------------");

//WRITE ALMOST SUCCESSFULL PACKET:
int pktStatus = 3;
int sourceID = m_nodes_id[Ipv4Address::ConvertFrom(dataHeader.GetSrc())];
int nextID = -1;
WritePacket(sourceID, packetID, nextID, pktStatus, 0.0);
}
}
else{
if(m_pu <= m_deadline){
NS_LOG_INFO("-----------------------------------------------------------------------------");
NS_LOG_INFO("SUPER FINE [" << packetID << "] RECEIVED AT AP[" << dataHeader.GetDst() << "] FROM AP[" << dataHeader.GetSrc() << "]
 - Hop Count 'pu' = " << static_cast<int>(m_pu) << ".");
NS_LOG_INFO("-----------------------------------------------------------------------------");

//WRITE BACKBONE SUPER FINE PACKET:
int pktStatus = 2;
int sourceID = m_nodes_id[Ipv4Address::ConvertFrom(dataHeader.GetSrc())];
int nextID = -1;
WritePacket(sourceID, packetID, nextID, pktStatus, 0.0);
}
else{
//If the packet arrived to the correct destination, but after the deadline:
NS_LOG_INFO("-----------------------------------------------------------------------------------------");
NS_LOG_INFO("(ALMOST) SUCCESSFULL PACKET [" << packetID << "] RECEIVED AT AP[" << dataHeader.GetDst() << "] FROM AP[" << dataHeader.GetSrc() << "]
 - Hop Count 'pu' = " << static_cast<int>(m_pu) << ".");
NS_LOG_INFO("-----------------------------------------------------------------------------------------");

//WRITE ALMOST SUCCESSFULL PACKET:
int pktStatus = 3;
int sourceID = m_nodes_id[Ipv4Address::ConvertFrom(dataHeader.GetSrc())];
int nextID = -1;
WritePacket(sourceID, packetID, nextID, pktStatus, 0.0);
}
}
}
else{
if(m_pu < m_deadline){//Caso esteja dentro do prazo, enviar RFB (ou DATA, caso seja Ãźltimo salto):
if(IsLastHop(dataHeader.GetDst())){
toBackbone = true;
SendData(dataHeader.GetSrc(),dataHeader.GetDst(), packetID, toBackbone);
}
else{
//std::cout <<"\n";
```

```
//std::cout << " NEW BID " << std::endl;
NS_LOG_INFO("BIDDER = " << m_nodeAddr);
//TODO Analyze a backbone delivery (the Budget X Throughput tradeoff). Not used in this strategy, but can be used at other strategies.
SendRFB(dataHeader.GetSrc(),dataHeader.GetDst(), packetID);
}
}
else
{
NS_LOG_INFO("Prazo (Deadline) estourado. Entregando pacote para o Backbone...");
if(haveNeighborBackbone(dataHeader.GetDst(), dataHeader.GetSrc())){
//If the backbone is near,send to the AP neighbor (ALMOST SUCCESSFULL case):
toBackbone = true;
//TODO If m_dropDest is the source, report this as a loop...
SendData(dataHeader.GetSrc(),m_dropDest, packetID, toBackbone);
}
//Caso backbone esteja distante, tratar pacote como perdido:
else{
NS_LOG_INFO("--------------------------------------------------------------------------------");
NS_LOG_INFO("DROP PACKET [" << packetID << "] AT NODE[" << m_nodeAddr << "] FROM AP[" << dataHeader.GetSrc() << "]
 - Hop Count 'pu' = " << static_cast<int>(m_pu) << ".");
NS_LOG_INFO("--------------------------------------------------------------------------------");

//WRITE DROP PACKET:
int pktStatus = 4;
int sourceID = m_nodes_id[Ipv4Address::ConvertFrom(dataHeader.GetSrc())];
int nextID = -1;
WritePacket(sourceID, packetID, nextID, pktStatus, 0.0);
}
}
}
break;
}
}
m_rxTrace (packet, from);
}
}

// Se receber Dados, enviar RFB
void
Offloading::SendRFB(Ipv4Address source, Ipv4Address destination, int packetID)
{
NS_LOG_FUNCTION (this);
NS_ASSERT (m_send_AP_rfbEvent.IsExpired ());

//int bid_index = 0;

RFBHeader rfbHeader;
rfbHeader.SetDeadline(m_deadline);
rfbHeader.SetHopcount(m_pu);
rfbHeader.SetPacketID(packetID);
rfbHeader.SetSrcRFB(m_nodeMacAddr);
rfbHeader.SetSrc(source);
rfbHeader.SetDst(destination);

if(node_type == NODETYPE_AP){
//m_backbonePair.push_back(std::make_pair(source,destination));
//bid_index = FindBidNumber(source, destination);

//ImprimeTopologySet ();

//std::cout << std::endl;
//std::cout << " NEW PACKET -> ID["<< packetID <<"] SOURCE["<< source <<"]" << std::endl;
//std::cout <<"\n";

}
else{
//bid_index = FindBidNumber(source, destination);
//rfbHeader.SetSrc(source);
//rfbHeader.SetDst(destination);

//if(strategy_type == STRATEGYTYPE_TIGHTNESS){
m_Bn = m_budget_percentage * m_offerBid;
m_Fn = m_fine_percentage * m_Bn;
NS_LOG_INFO("Budget percentage = " << m_budget_percentage << ", Fine percentage = " << m_fine_percentage);
//}
}

NS_LOG_INFO("Bn = " << m_Bn);
rfbHeader.SetBO(m_Bn*pow(10,2));
rfbHeader.SetFine(m_Fn*pow(10,2));
```

```
Ptr<Packet> packet = Create<Packet> ();
packet->AddHeader (rfbHeader);
TypeHeader tHeader (OFFLOADINGTYPE_RFB);
packet->AddHeader (tHeader);

// call to the trace sinks before the packet is actually sent,
// so that tags added to the packet can be sent as well
m_txTrace (packet);

SetRemote(Ipv4Address("255.255.255.255"),m_peerPort);
InetSocketAddress dst = InetSocketAddress (Ipv4Address::ConvertFrom(m_peerAddress), m_peerPort);
m_socket->Connect(dst);
m_socket->Send (packet);

m_tempo_envio = Simulator::Now ().GetSeconds ();

double getBn = rfbHeader.GetB0()/pow(10,2);
double getFn = rfbHeader.GetFine()/pow(10,2);

if (Ipv4Address::IsMatchingType (m_peerAddress))
{
NS_LOG_INFO ("["<< source <<"->"<< destination <<"]["<<packetID<<"]At "
<< Simulator::Now ().GetSeconds () << "s "
<< Ipv4Address::ConvertFrom (m_nodeAddr) <<
" sent a RFB."
" Budget B0 = "      << getBn <<
" Fine F0 = "        << getFn <<
" Deadline H0 = "    << static_cast<int>(rfbHeader.GetDeadline()) <<
" Hop Count 'pu' = " << static_cast<int>(rfbHeader.GetHopcount()));
}
else
NS_LOG_INFO ("["<< source <<"->"<< destination <<"]["<<packetID<<"]At " << Simulator::Now ().GetSeconds () << "s "
<< Ipv4Address::ConvertFrom (m_nodeAddr) <<
" sent a RFB."
" Budget B0 = "      << getBn <<
" Fine F0 = "        << getFn <<
" Deadline H0 = "    << static_cast<int>(rfbHeader.GetDeadline()) <<
" Hop Count 'pu' = " << static_cast<int>(rfbHeader.GetHopcount()));

m_send_dataEvent = Simulator::Schedule (Seconds (m_timeout), &Offloading::SendData, this, source, destination, packetID, false);

if((packetID < (m_nPackets - 1))&&(node_type==NODETYPE_AP)){
m_pu = 0;
packetID++;
m_send_AP_rfbEvent = Simulator::Schedule (Seconds (m_timeBetweenSourcePacket), &Offloading::SendRFB, this, m_nodeAddr, m_dest, packetID);
}
}

void
Offloading::SendData (Ipv4Address source, Ipv4Address destination, int packetID, bool toBackbone)
{
NS_LOG_FUNCTION (this);

NS_ASSERT (m_send_dataEvent.IsExpired ());

//Send "DATA" to the Winner node (or to the Backbone):
Ptr<Packet> packet = Create<Packet> (100);

DataHeader dataHeader;
dataHeader.SetSrc(source);
dataHeader.SetDst(destination);
dataHeader.SetPacketID(packetID);
dataHeader.SetHopcount(m_pu);
packet->AddHeader (dataHeader);

NS_LOG_DEBUG("");
NS_LOG_DEBUG("**********************************************************************");
NS_LOG_DEBUG("BIDDER: "<< m_nodeAddr << " [SOURCE: " << source << ", PACKET_ID: " << packetID << "]");

//int bid_index = FindBidNumber(source, destination);
int winner_index = 0;
double bid(0.0), power(0.0);
m_winner_bid = 0.0;//Helps to calculate the preference function at the Gauss strategy.
int pktStatus, sourceID, nextID;//Used to write packet
int tight = 0;
int count = 0;
double optimal_cn(0.0), optimal_bid(0.0);
double var_opi(0.0), var_cn(0.0), sd_opi(0.0), sd_cn(0.0), cov(0.0), corr(0.0);
double cn, wrong_cn, cn_max, delta_avg;
int able_nodes_count = 0;
```

```
//If it's a DATA send because timeout is over, choice the winner BID:
if(toBackbone == false){
//Choice winner if has received some BIDs:
if(m_bid_buffer.empty() == false){
//If it's the first bid, AP node does not use strategy. Choice the lowest BID:
if(node_type == NODETYPE_AP){

NS_LOG_DEBUG("BO = " << m_Bn );
NS_LOG_DEBUG("");

count = 0;
for(std::vector<uint32_t >::iterator it = m_bid_buffer.begin(); it != m_bid_buffer.end(); it++, count++){
NS_LOG_DEBUG("");
NS_LOG_DEBUG("Competitor = " << InetSocketAddress::ConvertFrom (m_address_buffer.at(count)).GetIpv4());
bid = ((*it)/pow(10,2));
NS_LOG_DEBUG("Bid = " << bid);
}

int min = 0;
for(int i = 0;i < m_pkt_count;i++){
bid = m_bid_buffer.at(i)/pow(10,2);
if(bid < m_bid_buffer.at(min)/pow(10,2)){
min = i;
}
}
//Winner Choice:
SetRemote(m_address_buffer.at(min),m_peerPort);
m_winner_bid = m_bid_buffer.at(min)/pow(10,2);

//TODO Deal with the case that the competitors are all negative or zero in the first RFB.
}
else if(node_type == NODETYPE_NETWORK){

//Calculate hop count (m_hc)
HopCountComputation (destination,1);
Ptr<RoutingProtocol> routing = this->GetNode()->GetObject<RoutingProtocol>();
OlsrState olsrState = routing->GetOlsrState();

std::vector<uint32_t >::iterator winner_dummy_bid;
std::vector<int>::iterator winner_dummy_path_hc;
std::vector<Address >::iterator winner_dummy_path;
int winner_index_hc = std::distance(m_hc.begin(), winner_dummy_path_hc);
double bid_dummy = 0.0;

switch(strategy_type){

case STRATEGYTYPE_DUMMYBID:

count = 0;
for(std::vector<uint32_t >::iterator it = m_bid_buffer.begin(); it != m_bid_buffer.end(); it++, count++){
NS_LOG_DEBUG("");
NS_LOG_DEBUG("Competitor = " << InetSocketAddress::ConvertFrom (m_address_buffer.at(count)).GetIpv4());
bid_dummy = ((*it)/pow(10,2));
NS_LOG_DEBUG("Bid = " << bid_dummy);
}

//Winner choice:
winner_dummy_bid = std::min_element(m_bid_buffer.begin(), m_bid_buffer.end());//Choose the Lowest Bid.
winner_index = std::distance(m_bid_buffer.begin(), winner_dummy_bid);
bid_dummy = ((m_bid_buffer.at(winner_index))/pow(10,2));
NS_LOG_DEBUG("Minimum Bid = " << bid_dummy);

NS_LOG_DEBUG("----------------------------------------------------------------------");
NS_LOG_DEBUG("WINNER: "<< InetSocketAddress::ConvertFrom (m_address_buffer.at(winner_index)).GetIpv4() );
SetRemote(m_address_buffer.at(winner_index),m_peerPort);
m_winner_bid = m_bid_buffer.at(winner_index)/pow(10,2);

break;

case STRATEGYTYPE_DUMMYPATH:

count = 0;
for(std::vector<int >::iterator it = m_hc.begin(); it != m_hc.end(); it++, count++){
NS_LOG_DEBUG("");
NS_LOG_DEBUG("Competitor = " << m_hc_address.at(count));
NS_LOG_DEBUG("Hop Count = " << *it);
}

//Winner choice:
winner_dummy_path_hc = std::min_element(m_hc.begin(), m_hc.end());//Choose the Shortest Path.
winner_index_hc = std::distance(m_hc.begin(), winner_dummy_path_hc);
```

```
//Find bid of the node "m_hc_address.at(winner_index_hc)":
winner_index = 0;
for(winner_dummy_path = m_address_buffer.begin(); winner_dummy_path != m_address_buffer.end(); winner_dummy_path++, winner_index++){
NS_LOG_INFO("Buffer Address = " << InetSocketAddress::ConvertFrom (*winner_dummy_path).GetIpv4());
bid_dummy = m_bid_buffer.at(winner_index)/pow(10,2);
NS_LOG_INFO("Buffer Bid = " << bid_dummy);
if((InetSocketAddress::ConvertFrom (*winner_dummy_path).GetIpv4()) == m_hc_address.at(winner_index_hc))
break;
}
//winner_dummy_path = std::find(m_address_buffer.begin(), m_address_buffer.begin(), m_hc_address.at(winner_index_hc));
//winner_index = std::distance(m_address_buffer.begin(), winner_dummy_path);
m_winner_bid = m_bid_buffer.at(winner_index)/pow(10,2);
SetRemote(m_address_buffer.at(winner_index) , m_peerPort);

NS_LOG_DEBUG("Minimum Hop Count = " << m_hc.at(winner_index_hc));
NS_LOG_DEBUG("----------------------------------------------------------------------");
NS_LOG_DEBUG("WINNER: "<< InetSocketAddress::ConvertFrom (m_address_buffer.at(winner_index)).GetIpv4() );

NS_LOG_DEBUG("Winner index = " << winner_index);
NS_LOG_DEBUG("Winner Bid = " << m_winner_bid);

break;

case STRATEGYTYPE_TIGHTNESS:

int N_competitors = 0;
int hc_buffer_equivalent_index = 0;
std::vector<Ipv4Address >::iterator hc_buffer_equivalent;
for(std::vector<Address>::const_iterator itCompetitors = m_address_buffer.begin ();
itCompetitors != m_address_buffer.end (); itCompetitors++){
//Find and print the equivalent Hop Count for each buffer address (because m_hc and Buffer are at different orders):
hc_buffer_equivalent = std::find(m_hc_address.begin(), m_hc_address.end(), InetSocketAddress::ConvertFrom (*itCompetitors).GetIpv4());
hc_buffer_equivalent_index = std::distance(m_hc_address.begin(), hc_buffer_equivalent);
NS_LOG_INFO("m_hc["<< InetSocketAddress::ConvertFrom (*itCompetitors).GetIpv4() <<"] = "
<< m_hc.at(hc_buffer_equivalent_index) <<".");
}

//Calculate delta_i:
N_competitors = 0;
for(std::vector<int>::const_iterator it_hc = m_hc.begin(); it_hc != m_hc.end(); it_hc++, N_competitors++){
m_delta.push_back((m_deadline - m_pu - 1) - *it_hc);
if(((m_deadline - m_pu - 1) - *it_hc) <= 0)
tight++;
NS_LOG_INFO("m_delta["<< m_nodeAddr <<"][" << *it_hc << "] = (H0 - pu - 1)- hci = (" << static_cast<int>(m_deadline) << " - " <<
 static_cast<int>(m_pu) << " - 1) - " << *it_hc << " = " << m_delta.at(N_competitors));
}

//When all competitors are "Tight", do a special winner choice (*not in the original strategy):
if(tight == N_competitors){
int max = 0;
bool equal = true;

for(int delta_index = 0;delta_index < (N_competitors-1);delta_index++)
{
if(m_delta.at(delta_index) != m_delta.at(delta_index+1)){
equal = false;
break;
}
}
//If all deltas are equal:
if(equal){
winner_index = 0;//Choice the first bid as the winner.
NS_LOG_INFO("IT WORKS! CASE 1 -> winner_index = 0");
}
//If there are different values of deltas:
else{
max = *std::max_element(m_delta.begin(), m_delta.end());//Choice the maximum value of deltas.
for(int delta_index = 0;delta_index < N_competitors;delta_index++)
{
if(m_delta.at(delta_index) == max){
winner_index = delta_index;//Choice the first maximum competitor.
NS_LOG_INFO("IT WORKS! CASE 2 -> winner_index = " << winner_index);
break;
}
}
}
}
//'Normal' cases:
else{
//Calculating average Delta:
```

```
delta_avg = 0;
able_nodes_count = 0;
for(std::size_t i = 0; i < m_delta.size(); i++){
if(m_delta.at(i) >= 0){
able_nodes_count++;
delta_avg += m_delta.at(i);
NS_LOG_DEBUG("delta_avg += m_delta.at(" << i << ") --> delta_avg += " << m_delta.at(i) );
}
}
delta_avg /= able_nodes_count;
NS_LOG_DEBUG("delta_avg /= able_nodes_count = " << delta_avg);

//Calculating maximum 'cn':
if(delta_avg != 0)
cn = m_delta[0]/delta_avg;
else if(m_delta[0] >= 0)
cn = m_delta[0] + 1;// We add "+ 1" to force this "cn" to be greater than when m_delta[0] = 0
else
cn = m_delta[0];
cn_max = cn;
for(int i = 0;i < m_pkt_count;i++){
if(delta_avg != 0)
cn = m_delta.at(i)/delta_avg;
else if(m_delta.at(i) >= 0)
cn = m_delta.at(i) + 1;// We add "+ 1" to force this "cn" to be greater than when m_delta[0] = 0
else
cn = m_delta.at(i);

cn_max = std::max(cn_max, cn);
}

if( (preference_function_type ==  PREFERENCEFUNCTION_GAUSS)||(preference_function_type ==  PREFERENCEFUNCTION_GAUSS_1) ){

if(preference_function_type ==  PREFERENCEFUNCTION_GAUSS_1)
optimal_cn = 1;
else
optimal_cn = cn_max;
optimal_bid = m_Fn;
NS_LOG_DEBUG("OPTIMAL VALUES:");
NS_LOG_DEBUG("Offerede Price = " << optimal_bid << "; Relative Tightness = " << optimal_cn);

//Calculate Offered Price Variance:
var_opi = 0;
for(int i = 0;i < m_pkt_count;i++){
bid = (m_bid_buffer.at(i))/pow(10,2);
NS_LOG_DEBUG("bid = " << bid );
power = pow((bid - optimal_bid),2);
NS_LOG_DEBUG("pow((bid - optimal_bid),2) = pow((" << bid << " - " << optimal_bid << "), 2) = " << power );
var_opi += power;
NS_LOG_DEBUG("sum(var_opi) = " << var_opi);
}
var_opi /= m_pkt_count;
NS_LOG_DEBUG("var_opi = " << var_opi);

//Calculate Relative Tightness Variance:
var_cn = 0;
for(int i = 0;i < m_pkt_count;i++){
if(delta_avg != 0)
cn = m_delta.at(i)/delta_avg;
else if(m_delta.at(i) >= 0)
cn = m_delta.at(i) + 1;// We add "+ 1" to force this "cn" to be greater than when m_delta[0] = 0
else
cn = m_delta.at(i);

NS_LOG_DEBUG("cn = m_delta.at(" << i << ")/delta_avg = " << cn);
power = pow((cn - optimal_cn),2);
NS_LOG_DEBUG("pow((cn - optimal_cn),2) = pow((" << cn << " - " << optimal_cn << "), 2) = " << power );
var_cn += power;
NS_LOG_DEBUG("sum(var_cn) = " << var_cn);
}
var_cn /= m_pkt_count;
NS_LOG_DEBUG("var_cn = " << var_cn);

//Calculate Covariance between Offered Price and Relative Tightness:
cov = 0;
for(int i = 0;i < m_pkt_count;i++){
if(delta_avg != 0)
cn = m_delta.at(i)/delta_avg;
else if(m_delta.at(i) >= 0)
cn = m_delta.at(i) + 1;// We add "+ 1" to force this "cn" to be greater than when m_delta[0] = 0
else
```

```
cn = m_delta.at(i);

cov += ((m_bid_buffer.at(i)/pow(10,2)) - optimal_bid)*(cn - optimal_cn);
}
cov /= m_pkt_count;

//Calculate standard deviation:
sd_opi = sqrt(var_opi);
sd_cn = sqrt(var_cn);

//Calculate correlation:
corr = cov/(var_opi*var_cn);
}

//Calculating the Preference Function:
std::vector<double > P;
double preference = 0.0;

NS_LOG_DEBUG("k1 = " << m_k1 << ", k2 = " << m_k2);
NS_LOG_DEBUG("cn_max = " << cn_max);
NS_LOG_DEBUG("");


//Show m_hc_address addresses:
int hc_addr_index = 0;
for(std::vector<Ipv4Address >::iterator it = m_hc_address.begin(); it != m_hc_address.end(); it++, hc_addr_index++){
NS_LOG_INFO("m_hc_address[" << hc_addr_index << "] = " << *it);
}

//Calculate Preference Function values:
for(int i = 0;i < m_pkt_count;i++){
bid = m_bid_buffer.at(i)/pow(10,2);

//Find the equivalent Hop Count for each buffer address (because m_hc and the Buffer are at different orders, and m_delta is at the same order as m_hc):
hc_buffer_equivalent = std::find(m_hc_address.begin(), m_hc_address.end(), InetSocketAddress::ConvertFrom (m_address_buffer.at(i)).GetIpv4());
hc_buffer_equivalent_index = std::distance(m_hc_address.begin(), hc_buffer_equivalent);
if(delta_avg != 0)
cn = m_delta.at(i)/delta_avg;
else if(m_delta.at(i) >= 0)
cn = m_delta.at(i) + 1;// We add "+ 1" to force this "cn" to be greater than when m_delta[0] = 0
else
cn = m_delta.at(i);

//Select Preference Fuction:
switch(preference_function_type){

case PREFERENCEFUNCTION_PLANE:
preference = m_k1 - (m_k1/m_Bn)*bid + (m_k2/cn_max)*cn;
break;

case PREFERENCEFUNCTION_GAUSS:
//Calculate 2-dimensional Gauss centered at [Fn,c_max]:
preference = (1/(2*M_PI*sd_opi*sd_cn*sqrt(1 - pow(corr,2)))) * exp( (-1/(2*(1 - pow(corr,2)))) * ( (pow((bid - optimal_bid),2)/var_opi) +
 (pow((cn - optimal_cn),2)/var_cn) - ((2*corr*(bid - optimal_bid)*(cn - optimal_cn))/(sd_opi*sd_cn)) ) );
break;

case PREFERENCEFUNCTION_GAUSS_1:
//Calculate 2-dimensional Gauss centered at [Fn,c_max]:
preference = (1/(2*M_PI*sd_opi*sd_cn*sqrt(1 - pow(corr,2)))) * exp( (-1/(2*(1 - pow(corr,2)))) * ( (pow((bid - optimal_bid),2)/var_opi) +
 (pow((cn - optimal_cn),2)/var_cn) - ((2*corr*(bid - optimal_bid)*(cn - optimal_cn))/(sd_opi*sd_cn)) ) );
break;

}

NS_LOG_DEBUG("COMPETITOR " << InetSocketAddress::ConvertFrom (m_address_buffer.at(i)).GetIpv4() << ":");

//Show the problem of order effect:
if(hc_buffer_equivalent_index != i){
m_wrong_order++;
NS_LOG_INFO("");
NS_LOG_DEBUG("WRONG ORDER!! Total order errors = " << m_wrong_order);
NS_LOG_INFO("Buffer index (wrong Delta) = " << i << " <--> HC index (correct Delta) = " << hc_buffer_equivalent_index);
NS_LOG_INFO("m_delta[" << i << "] = " << m_delta.at(i) << " <--> m_delta[" << hc_buffer_equivalent_index << "] = " << m_delta.at(hc_buffer_equivalent_index) );
wrong_cn = m_delta.at(i)/delta_avg;
NS_LOG_INFO("cn[" << i << "] = " << wrong_cn << " <--> cn[" << hc_buffer_equivalent_index << "] = " << cn );
NS_LOG_INFO("");
}

//Show preference values:
NS_LOG_DEBUG("Bid = " << bid << " Relative Tight (cn) = " << cn );
NS_LOG_DEBUG("Preference value = " << preference);
```

```
P.push_back(preference);
}

//Winner choice:
std::vector<double >::iterator winner;
winner = std::max_element(P.begin(), P.end());//Choose the largest preference function.
winner_index = std::distance(P.begin(), winner);
}

if( (preference_function_type ==  PREFERENCEFUNCTION_GAUSS)||(preference_function_type ==  PREFERENCEFUNCTION_GAUSS_1) ){
//Statistical data of competitors:
NS_LOG_DEBUG("");
NS_LOG_DEBUG("###### Statistical data of competitors:");
NS_LOG_DEBUG("Offered Price variance = " << var_opi);
NS_LOG_DEBUG("Relative Tightness variance = " << var_cn);
NS_LOG_DEBUG("Covariance between Relative Tightness and Offered Price = " << cov);
NS_LOG_DEBUG("Offered Price standard deviation = " << sd_opi);
NS_LOG_DEBUG("Relative Tightness standard deviation = " << sd_cn);
NS_LOG_DEBUG("Correlation between Relative Tightness and Offered Price = " << corr);
}

NS_LOG_DEBUG("-----------------------------------------------------------------");
NS_LOG_DEBUG("WINNER: "<< InetSocketAddress::ConvertFrom (m_address_buffer.at(winner_index)).GetIpv4() );
SetRemote(m_address_buffer.at(winner_index),m_peerPort);
m_winner_bid = m_bid_buffer.at(winner_index)/pow(10,2);

break;
}
}
}
//No BIDs received after timeout:
else{
NS_LOG_INFO("No BIDs received at "<< m_nodeAddr <<".");
if(node_type == NODETYPE_NETWORK){
//If there is a backbone nearby, sends to the neighbor AP:
if((haveNeighborBackbone(destination, source))&&(node_type == NODETYPE_NETWORK)){
//Here is a case of dead end, with an AP nearby (even within deadline).
toBackbone = true;
//TODO If m_dropDest == source, report this as a loop?
SetRemote(m_dropDest,m_peerPort);
}
else{//No backbone nearby:
//Here is a case of dead end, with no neighbor (or AP) nearby (even within deadline).
NS_LOG_INFO("-------------------------------------------------------------------------------");
NS_LOG_INFO("DROP PACKET [" << packetID << "] AT NODE[" << m_nodeAddr << "] FROM AP[" << dataHeader.GetSrc() << "] - Hop Count 'pu' = "
<< static_cast<int>(m_pu) << ".");
NS_LOG_INFO("-------------------------------------------------------------------------------");

//WRITE DROP PACKET:
pktStatus = 4;
sourceID = m_nodes_id[Ipv4Address::ConvertFrom(source)];
nextID = -1;
WritePacket(sourceID, packetID, nextID, pktStatus, 0.0);
return;
}
}
else{
//If this node is AP source, so it could not send any packet yet:
NS_LOG_INFO("THIS AP SOURCE CANNOT START A BID. NO BIDs RECEIVED YET...");

//TODO Retransmit an RFB packet?

return;
}
}
}
else
SetRemote(destination,m_peerPort);

TypeHeader tHeader (OFFLOADINGTYPE_DATA);
packet->AddHeader(tHeader);

if(toBackbone == false){
m_socket->SendTo (packet, 0, m_peerAddress);

if (Ipv4Address::IsMatchingType (m_peerAddress))
{
NS_LOG_INFO ("["<< source <<"->"<< destination <<"]["<<packetID<<"]At " << Simulator::Now ().GetSeconds () << "s "
<< Ipv4Address::ConvertFrom (m_nodeAddr) <<
" sent a DATA to " << Ipv4Address::ConvertFrom (m_peerAddress));
}
```

```
else{
NS_LOG_INFO ("["<< source <<"->"<< destination <<"]["<<packetID<<"]At " << Simulator::Now ().GetSeconds () << "s "
<< Ipv4Address::ConvertFrom (m_nodeAddr) <<
" sent a DATA to " <<
InetSocketAddress::ConvertFrom (m_peerAddress).GetIpv4 ());
}
}
else{
InetSocketAddress dst = InetSocketAddress (Ipv4Address::ConvertFrom(m_peerAddress), m_peerPort);
m_socket->Connect(dst);
m_socket->Send (packet);

if (Ipv4Address::IsMatchingType (m_peerAddress))
{
NS_LOG_INFO ("["<< source <<"->"<< destination <<"]["<<packetID<<"]At " << Simulator::Now ().GetSeconds () << "s "
<< Ipv4Address::ConvertFrom (m_nodeAddr) <<
" sent a DATA to " << Ipv4Address::ConvertFrom (m_peerAddress) <<
"Hop Count 'pu' = " << static_cast<int>(m_pu));
}
else{
NS_LOG_INFO ("["<< source <<"->"<< destination <<"]["<<packetID<<"]At " << Simulator::Now ().GetSeconds () << "s "
<< Ipv4Address::ConvertFrom (m_nodeAddr) <<
" sent a DATA to " <<
InetSocketAddress::ConvertFrom (m_peerAddress).GetIpv4 () <<
" Hop Count 'pu' = " << static_cast<int>(m_pu));
}
}

//WRITE OFFLOADED PACKET:
sourceID = m_nodes_id[Ipv4Address::ConvertFrom(source)];
if(toBackbone == false){
nextID = m_nodes_id[InetSocketAddress::ConvertFrom (m_peerAddress).GetIpv4 ()];
pktStatus = 5;
}
else{
nextID = m_nodes_id[Ipv4Address::ConvertFrom(m_peerAddress)];
pktStatus = 6;
}
WritePacket(sourceID, packetID, nextID, pktStatus, m_winner_bid);

//Clean buffer:
m_bid_buffer.erase (m_bid_buffer.begin(),m_bid_buffer.end());
m_address_buffer.erase (m_address_buffer.begin(),m_address_buffer.end());
m_pkt_count = 0;

//Clean vectors:
m_hc.erase (m_hc.begin(),m_hc.end());
m_hc_address.erase (m_hc_address.begin(),m_hc_address.end());
m_delta.erase (m_delta.begin(),m_delta.end());

//m_backbonePair.erase(m_backbonePair.begin() + bid_index);

//Loop control:
m_loopControl.push_back(std::make_pair(source, packetID));
m_pu = 0;
}

// Guardar no buffer se receber proposta (bid) dentro do Timeout:
void
Offloading::BufferBid(Address from, uint32_t bid, Ipv4Address source, Ipv4Address destination)
{
if((Simulator::Now ().GetSeconds () - m_tempo_envio) < m_timeout){
m_bid_buffer.push_back(bid);
m_address_buffer.push_back(from);
//m_source_buffer.push_back(source);
//m_destination_buffer.push_back(destination);
m_pkt_count++;
}
}

// Se receber RFB, enviar lance
void
Offloading::SendBid(Ptr<Socket> socket, uint32_t Bu, uint32_t Fu, uint8_t H0, int packetID)
{
NS_LOG_LOGIC ("Sending bid...");

NS_ASSERT (m_send_bidEvent.IsExpired ());

//ImprimeTopologySet ();
HopCountComputation (m_dest,0);
BidComputation (Bu, Fu, H0);
```

```
BidHeader bidHeader;
bidHeader.SetOfferedBid(m_offerBid*pow(10,2));
bidHeader.SetSrc(m_source);
bidHeader.SetDst(m_dest);
bidHeader.SetPacketID(packetID);

Ptr<Packet> packet = Create<Packet> ();
packet->AddHeader (bidHeader);
TypeHeader tHeader (OFFLOADINGTYPE_BID);
packet->AddHeader (tHeader);


//Insere endereÃğo MAC do nÃş leiloeiro (upload) na tabela ARP:
PopulateArpCache();

socket->SendTo(packet, 0, m_upload);

double getOfferbid = bidHeader.GetOfferedBid()/pow(10,2);

if (InetSocketAddress::IsMatchingType (m_upload))
{
NS_LOG_INFO ("["<< m_source <<"->"<< m_dest <<"]["<<packetID<<"]At " << Simulator::Now ().GetSeconds () << "s "
<< Ipv4Address::ConvertFrom (m_nodeAddr) <<
" sent a BID to " <<
InetSocketAddress::ConvertFrom (m_upload).GetIpv4 () <<
" Offered Bid = " << getOfferbid);
}

//Limpa vetores:
m_hc.erase (m_hc.begin(),m_hc.end());
m_hc_address.erase (m_hc_address.begin(),m_hc_address.end());
m_delta.erase (m_delta.begin(),m_delta.end());
m_able_nodes.erase (m_able_nodes.begin(),m_able_nodes.end());
//grafo_dijkstra.erase(grafo_dijkstra.begin(),grafo_dijkstra.end());
}

bool
Offloading::IsLastHop (Ipv4Address dest){

Ptr<RoutingProtocol> routing = this->GetNode()->GetObject<RoutingProtocol>();
std::vector<RoutingTableEntry> entry = routing->GetRoutingTableEntries();

for (std::vector< RoutingTableEntry, std::allocator<RoutingTableEntry> >::iterator i = entry.begin(); i!=entry.end(); i++)
{
if((i->destAddr == dest) && (i->distance == 1))
return true;
}

return false;
}

//Store the hop counts of this node competitors in the array "m_hc".
//If this node is the Bidder, nodetype=1. If this node received a RFB, nodetype=0:
void
Offloading::HopCountComputation (Ipv4Address dest, int nodetype)
{
/*
 * Hop counts:
 * - m_hc[0] representa o hci deste nÃş
 * - m_hc[i] p/ i != 0 representa o hci dos competidores deste nÃş
 */

Ptr<RoutingProtocol> routing = this->GetNode()->GetObject<RoutingProtocol>();
OlsrState olsrState = routing->GetOlsrState();

//Construir Grafo via repositÃşrio OLSR. Grafo(grafo_dijkstra) ÃĬ usado no algoritmo Dijkstra:
adjacency_list_t grafo_dijkstra(number_nodes);
BuildGraphDijkstra(grafo_dijkstra);

//AplicaÃğÃčo de Dijkstra sobre o Grafo:
std::vector<weight_t> min_distance;
std::vector<vertex_t> previous;

int delta = 0;
//CÃąlculo do Hop Count deste nÃş (m_hc[0]):
if(nodetype == 0){
DijkstraComputePaths(m_map_address_graphnode[m_nodeAddr], grafo_dijkstra, min_distance, previous);
m_hc.push_back(min_distance[m_map_address_graphnode[dest]]);
m_hc_address.push_back(m_nodeAddr);
```

```
NS_LOG_DEBUG("Calculo de HopCounts pelo no " << m_nodeAddr << " para calcular Offered Bid:");
delta = (m_deadline - m_pu - 1) - min_distance[m_map_address_graphnode[dest]];
NS_LOG_DEBUG("m_hc["<< m_nodeAddr <<"]["<< m_nodeAddr <<" -> " << dest << "] = "
<< min_distance[m_map_address_graphnode[dest]]
<< " Delta = (H0 - pu - 1)- hci = ("
<< static_cast<int>(m_deadline) << " - " << static_cast<int>(m_pu) << " - 1) - "
<< min_distance[m_map_address_graphnode[dest]] << " = " << delta);
}

if(nodetype == 0)//TODO Problem of incomplete network (OLSR). Pass 'hc' of competitors via RFB?
for ( TwoHopNeighborSet::const_iterator itTwohopNeighbor = olsrState.GetTwoHopNeighbors ().begin ();
itTwohopNeighbor != olsrState.GetTwoHopNeighbors ().end (); itTwohopNeighbor++)
{
//Verifica somente os vizinhos do nÃş leiloeiro (upload node) e que nÃčo sejam APs (nÃčo pertenÃğam ao backbone):
if((itTwohopNeighbor->neighborMainAddr == InetSocketAddress::ConvertFrom (m_upload).GetIpv4 ())&&(!isBackbone(itTwohopNeighbor->twoHopNeighborAddr))){
DijkstraComputePaths(m_map_address_graphnode[itTwohopNeighbor->twoHopNeighborAddr], grafo_dijkstra, min_distance, previous);
m_hc.push_back(min_distance[m_map_address_graphnode[dest]]);
m_hc_address.push_back(itTwohopNeighbor->twoHopNeighborAddr);
delta = (m_deadline - m_pu - 1) - min_distance[m_map_address_graphnode[dest]];
NS_LOG_DEBUG("m_hc["<< m_nodeAddr <<"]["<< itTwohopNeighbor->twoHopNeighborAddr <<" -> " <<
    dest << "] = " << min_distance[m_map_address_graphnode[dest]]
         << " Delta = (H0 - pu - 1)- hci = ("
     << static_cast<int>(m_deadline) << " - " << static_cast<int>(m_pu) << " - 1) - "
     << min_distance[m_map_address_graphnode[dest]] << " = " << delta);
}
}
else
for ( std::vector<Address>::const_iterator itCompetitors = m_address_buffer.begin ();
itCompetitors != m_address_buffer.end (); itCompetitors++)
{
DijkstraComputePaths(m_map_address_graphnode[InetSocketAddress::ConvertFrom (*itCompetitors).GetIpv4()], grafo_dijkstra, min_distance, previous);
m_hc.push_back(min_distance[m_map_address_graphnode[dest]]);
m_hc_address.push_back(InetSocketAddress::ConvertFrom (*itCompetitors).GetIpv4());
delta = (m_deadline - m_pu - 1) - min_distance[m_map_address_graphnode[dest]];
NS_LOG_INFO("m_hc["<< m_nodeAddr <<"]["<< InetSocketAddress::ConvertFrom (*itCompetitors).GetIpv4() <<" -> " <<
    dest << "] = " << min_distance[m_map_address_graphnode[dest]]
                  << " Delta = (H0 - pu - 1)- hci = ("
         << static_cast<int>(m_deadline) << " - " << static_cast<int>(m_pu) << " - 1) - "
         << min_distance[m_map_address_graphnode[dest]] << " = " << delta);
}
}


void
Offloading::BidComputation (uint32_t Bu, uint32_t Fu, uint8_t H0){

UniformVariable uv;
if((strategy_type == STRATEGYTYPE_DUMMYBID) || (strategy_type == STRATEGYTYPE_DUMMYPATH)){
// Calculate the Random offered Bid for the Dummy cases:
m_offerBid = (uv.GetValue(Fu/pow(10,2), Bu/pow(10,2)));
NS_LOG_INFO("* Offered Bid [" << m_nodeAddr << "] = " << m_offerBid);
}
else if(strategy_type == STRATEGYTYPE_TIGHTNESS){
//Calculate delta_i
int i = 0;
for(std::vector<int>::const_iterator it_hc = m_hc.begin(); it_hc != m_hc.end(); it_hc++, i++){
m_delta.push_back((H0 - m_pu - 1) - *it_hc);
NS_LOG_INFO("m_delta["<< m_nodeAddr <<"][" << i << "] = (H0 - pu - 1)- hci = (" << static_cast<int>(H0) << " - " << static_cast<int>(m_pu) << " - 1) - " <<
 *it_hc << " = " << m_delta[i]);
}
NS_LOG_INFO("m_delta["<< m_nodeAddr <<"][0] = (H0 - pu - 1)- hci = (" << static_cast<int>(H0) << " - " << static_cast<int>(m_pu) << " - 1) - " <<
 m_hc.at(0) << " = " << m_delta[0]);

//Calculate parameters:
double delta_avg = 0.0;
int delta_max = 0;
double cn = 0.0;
double an;

//Set S(u)[m_able_nodes]:
for(std::vector<int>::const_iterator it = m_delta.begin(); it != m_delta.end(); it++){
if(*it >= 0){
m_able_nodes.push_back(*it);
NS_LOG_INFO("m_able_nodes.push_back("<< *it <<")");
}
}

//If exist able nodes to compete with this node AND this node is able to compete, cn = delta/delta_avg:
if((m_delta[0] > 0)&&(m_able_nodes.size() > 1)){
delta_max = *std::max_element(m_able_nodes.begin()+1, m_able_nodes.end());
NS_LOG_INFO("delta_max = " << delta_max);
```

```
for(std::size_t i = 0; i < m_able_nodes.size(); i++)
delta_avg += m_able_nodes.at(i);
delta_avg /= m_able_nodes.size();
NS_LOG_INFO("delta_avg = " << delta_avg);

cn = (m_delta[0])/delta_avg;
NS_LOG_INFO("cn = " << cn);
an = m_delta[0];
if(delta_max!=0)// If all able competitors are zero, an = this node Delta. Not specified at original strategy.
an /= delta_max;
NS_LOG_INFO("an = " << an);

m_offerBid = (Bu/pow(10,2) - Fu/pow(10,2))*(1-(1/(1+ exp(-(an*(cn - 1))) ))) + Fu/pow(10,2);
}
else
m_offerBid = Bu/pow(10,2);//If this node is not an able competitor OR it is the unique able competitor OR ther is no competitors

NS_LOG_INFO("m_offerBid = " << m_offerBid);
}
}

void
Offloading::BuildGraphDijkstra(adjacency_list_t &grafo_dijkstra){

Ptr<RoutingProtocol> routing = this->GetNode()->GetObject<RoutingProtocol>();
OlsrState olsrState = routing->GetOlsrState();
const TopologySet &topology = olsrState.GetTopologySet ();

// ConstruÃğÃčo do Grafo (adjacency_list):
// Parte 1: TopologySet - OLSR:
for (std::map<Ipv4Address, int>::const_iterator mapGraphNode =
m_map_address_graphnode.begin (); mapGraphNode != m_map_address_graphnode.end (); mapGraphNode++)
{
for(TopologySet::const_iterator tuple = topology.begin ();
tuple != topology.end (); tuple++)
{
if(mapGraphNode->first == tuple->lastAddr){

if(!FindGraphEdge(mapGraphNode->second, m_map_address_graphnode[tuple->destAddr], grafo_dijkstra)){
grafo_dijkstra[mapGraphNode->second].push_back(neighbor(m_map_address_graphnode[tuple->destAddr], 1));//ida
//NS_LOG_INFO(mapGraphNode->first << " --> " << tuple->destAddr);
}
if(!FindGraphEdge(m_map_address_graphnode[tuple->destAddr], mapGraphNode->second, grafo_dijkstra)){
grafo_dijkstra[m_map_address_graphnode[tuple->destAddr]].push_back(neighbor(mapGraphNode->second, 1));//volta
//NS_LOG_INFO(tuple->destAddr << " --> " << mapGraphNode->first);
}

}
if(mapGraphNode->first == tuple->destAddr){

if(!FindGraphEdge(mapGraphNode->second, m_map_address_graphnode[tuple->lastAddr], grafo_dijkstra)){
grafo_dijkstra[mapGraphNode->second].push_back(neighbor(m_map_address_graphnode[tuple->lastAddr], 1));//ida
//NS_LOG_INFO(mapGraphNode->first << " --> " << tuple->lastAddr);
}
if(!FindGraphEdge(m_map_address_graphnode[tuple->lastAddr], mapGraphNode->second, grafo_dijkstra)){
grafo_dijkstra[m_map_address_graphnode[tuple->lastAddr]].push_back(neighbor(mapGraphNode->second, 1));//volta
//NS_LOG_INFO(tuple->lastAddr << " --> " << mapGraphNode->first);
}
}
}
}

//Parte 2: NeighborSet - OLSR:
for ( NeighborSet::const_iterator itNeighbor = olsrState.GetNeighbors ().begin ();
itNeighbor != olsrState.GetNeighbors ().end (); itNeighbor++)
{
if( (!FindGraphEdge(m_map_address_graphnode[m_nodeAddr], m_map_address_graphnode[itNeighbor->neighborMainAddr], grafo_dijkstra) )){
grafo_dijkstra[m_map_address_graphnode[m_nodeAddr]].push_back(neighbor(m_map_address_graphnode[itNeighbor->neighborMainAddr], 1));
grafo_dijkstra[m_map_address_graphnode[itNeighbor->neighborMainAddr]].push_back(neighbor(m_map_address_graphnode[m_nodeAddr], 1));
}
}

//Parte 3: TwoHopNeighborSet - OLSR:
for ( TwoHopNeighborSet::const_iterator itTwohopNeighbor = olsrState.GetTwoHopNeighbors ().begin ();
itTwohopNeighbor != olsrState.GetTwoHopNeighbors ().end (); itTwohopNeighbor++)
{
//NS_LOG_INFO("Neighbor: " << itTwohopNeighbor->neighborMainAddr << " --> TwoHopNeighbor: " << itTwohopNeighbor->twoHopNeighborAddr);
if( (!FindGraphEdge(m_map_address_graphnode[itTwohopNeighbor->neighborMainAddr], m_map_address_graphnode[itTwohopNeighbor->twoHopNeighborAddr], grafo_dijkstra)) &&
(!FindGraphEdge(m_map_address_graphnode[itTwohopNeighbor->twoHopNeighborAddr], m_map_address_graphnode[itTwohopNeighbor->neighborMainAddr], grafo_dijkstra)) ){
grafo_dijkstra[m_map_address_graphnode[itTwohopNeighbor->neighborMainAddr]].push_back(neighbor(m_map_address_graphnode[itTwohopNeighbor->twoHopNeighborAddr], 1));
```

```
grafo_dijkstra[m_map_address_graphnode[itTwohopNeighbor->twoHopNeighborAddr]].push_back(neighbor(m_map_address_graphnode[itTwohopNeighbor->neighborMainAddr], 1));
}
}
//ImprimeGrafo (true, grafo_dijkstra);
}

//Verifica se existe determinada aresta direcionada (a --> b) no Grafo:
bool
Offloading::FindGraphEdge (int a, int b, adjacency_list_t &grafo_dijkstra){

int i = 0;
for(std::vector<std::vector<neighbor> >::const_iterator it_a = grafo_dijkstra.begin(); it_a != grafo_dijkstra.end(); it_a++, i++)
for(std::vector<neighbor>::const_iterator it_b = it_a->begin(); it_b != it_a->end(); it_b++)
if((i == a) && (it_b->target == b))
return true;

return false;
}

void
Offloading::ImprimeTopologySet ()
{
Ptr<RoutingProtocol> routing = this->GetNode()->GetObject<RoutingProtocol>();

//Imprime TopologySet:
OlsrState olsrState = routing->GetOlsrState();
const TopologySet &topology = olsrState.GetTopologySet ();
NS_LOG_INFO (Simulator::Now ().GetSeconds ()
<< "s ** BEGIN dump TopologySet for OLSR Node " << m_nodeAddr);
for (TopologySet::const_iterator tuple = topology.begin ();
tuple != topology.end (); tuple++)
{
NS_LOG_INFO (*tuple);
}
NS_LOG_INFO ("** END dump TopologySet Set for OLSR Node " << m_nodeAddr);

}

void
Offloading::ImprimeGrafo (bool sort, adjacency_list_t &grafo_dijkstra)
{
int node = 0;
int edge = 0;
if(sort){
for(std::map<Ipv4Address, int>::const_iterator mapGraphNode = m_map_address_graphnode.begin ();
mapGraphNode != m_map_address_graphnode.end (); mapGraphNode++, node++)
for(std::vector<neighbor>::const_iterator it = grafo_dijkstra[mapGraphNode->second].begin(); it != grafo_dijkstra[mapGraphNode->second].end(); it++){
NS_LOG_INFO (mapGraphNode->first << " <--> " << FindMapAddress (it->target));
edge++;
}
}
else{
for(std::vector<std::vector<neighbor> >::const_iterator it_a = grafo_dijkstra.begin(); it_a != grafo_dijkstra.end(); it_a++, node++)
for(std::vector<neighbor>::const_iterator it_b = it_a->begin(); it_b != it_a->end(); it_b++){
NS_LOG_INFO (FindMapAddress (node) << " <--> " << FindMapAddress (it_b->target));
edge++;
}
}

NS_LOG_INFO("Este Grafo possui " << node << " no's e " << edge << " arestas.");
}

//Verifica endereÃ§o IPv4 do nÃ§ 'a' no Grafo:
Ipv4Address
Offloading::FindMapAddress (int a){
Ipv4Address mapAddr;

for(std::map<Ipv4Address, int>::const_iterator mapGraphNode = m_map_address_graphnode.begin ();
mapGraphNode != m_map_address_graphnode.end (); mapGraphNode++){
if(mapGraphNode->second == a)
mapAddr = mapGraphNode->first;
}
return mapAddr;
}

void
Offloading::DijkstraComputePaths(vertex_t source,
const adjacency_list_t &adjacency_list,
std::vector<weight_t> &min_distance,
std::vector<vertex_t> &previous)
{
```

```
int n = adjacency_list.size();
min_distance.clear();
min_distance.resize(n, std::numeric_limits<double>::infinity());
min_distance[source] = 0;
previous.clear();
previous.resize(n, -1);
std::set<std::pair<weight_t, vertex_t> > vertex_queue;
vertex_queue.insert(std::make_pair(min_distance[source], source));

while (!vertex_queue.empty())
{
weight_t dist = vertex_queue.begin()->first;
vertex_t u = vertex_queue.begin()->second;
vertex_queue.erase(vertex_queue.begin());

// Visit each edge exiting u
const std::vector<neighbor> &neighbors = adjacency_list[u];
for (std::vector<neighbor>::const_iterator neighbor_iter = neighbors.begin();
neighbor_iter != neighbors.end();
neighbor_iter++)
{
vertex_t v = neighbor_iter->target;
weight_t weight = neighbor_iter->weight;
weight_t distance_through_u = dist + weight;
if (distance_through_u < min_distance[v]) {
vertex_queue.erase(std::make_pair(min_distance[v], v));

min_distance[v] = distance_through_u;
previous[v] = u;
vertex_queue.insert(std::make_pair(min_distance[v], v));

}

}
}
}

std::list<vertex_t>
Offloading::DijkstraGetShortestPathTo(vertex_t vertex, const std::vector<vertex_t> &previous)
{
std::list<vertex_t> path;
for ( ; vertex != -1; vertex = previous[vertex])
path.push_front(vertex);
return path;
}

void
Offloading::MapNode(void){

Ptr<RoutingProtocol> routing = this->GetNode()->GetObject<RoutingProtocol>();
OlsrState olsrState = routing->GetOlsrState();
const TopologySet &topology = olsrState.GetTopologySet ();

//NS_LOG_INFO("Mapeamento de Grafo para o nÃş " << m_nodeAddr << " :");

int i = 0;
// Mapeamento via TopologySet- [EndereÃğo Ipv4 -> nÃžmero do nÃş no Grafo]:
for (TopologySet::const_iterator tuple = topology.begin (); tuple != topology.end (); tuple++)
{
if(m_map_address_graphnode.find(tuple->destAddr) == m_map_address_graphnode.end()){ //Se o endereÃğo nÃčo estiver mapeado...
m_map_address_graphnode[tuple->destAddr] = i++; //...adiciona nÃžmero do nÃş.
//NS_LOG_INFO("m_map_address_graphnode[" << tuple->destAddr << "] = " << m_map_address_graphnode[tuple->destAddr]);
}
if(m_map_address_graphnode.find(tuple->lastAddr) == m_map_address_graphnode.end()){
m_map_address_graphnode[tuple->lastAddr] = i++;
//NS_LOG_INFO("m_map_address_graphnode[" << tuple->lastAddr << "] = " << m_map_address_graphnode[tuple->lastAddr]);
}
}

// Mapeamento via NeighborSet- [EndereÃğo Ipv4 -> nÃžmero do nÃş no Grafo]:
for (NeighborSet::const_iterator tuple = olsrState.GetNeighbors().begin (); tuple != olsrState.GetNeighbors().end (); tuple++)
{
if(m_map_address_graphnode.find(tuple->neighborMainAddr) == m_map_address_graphnode.end()){ //Se o endereÃğo nÃčo estiver mapeado...
m_map_address_graphnode[tuple->neighborMainAddr] = i++; //...adiciona nÃžmero do nÃş.
//NS_LOG_INFO("m_map_address_graphnode[" << tuple->neighborMainAddr << "] = " << m_map_address_graphnode[tuple->neighborMainAddr]);
}
}

}

void
```

```
Offloading::PopulateArpCache (void)
{
Ptr <ArpL3Protocol> arpL3 = m_thisNode->GetObject <ArpL3Protocol> ();
Ptr <ArpCache> arp = arpL3->FindCache (m_thisNode->GetDevice(0));

arp->SetAliveTimeout (Seconds(3600 * 24 * 365));

ArpCache::Entry * have_entry = arp->Lookup(InetSocketAddress::ConvertFrom (m_upload).GetIpv4 ());
//Caso a entrada na tabela nÃ£o exista, adiciona entrada. Caso contrÃ¡rio nÃ£o faz nada:
if(have_entry == 0){
ArpCache::Entry * entry = arp->Add(InetSocketAddress::ConvertFrom (m_upload).GetIpv4 ());
entry->MarkWaitReply(0);
entry->MarkAlive(m_uploadMAC);

Ptr<Ipv4L3Protocol> ip = m_thisNode->GetObject<Ipv4L3Protocol> ();
NS_ASSERT(ip !=0);
ip->GetInterface(0)->SetAttribute("ArpCache", PointerValue(arp));
}
}

//Check if this node have some neighbor that is AP, and if the final destination 'dest' is a neighbor too.
//Store address in 'm_dropDest':
bool
Offloading::haveNeighborBackbone(Ipv4Address dest, Ipv4Address src)
{
Ptr<RoutingProtocol> routing = this->GetNode()->GetObject<RoutingProtocol>();
OlsrState olsrState = routing->GetOlsrState();
bool haveNeighborAP = false;

for (NeighborSet::const_iterator tuple = olsrState.GetNeighbors().begin (); tuple != olsrState.GetNeighbors().end (); tuple++){
if(isBackbone(tuple->neighborMainAddr)){
if(tuple->neighborMainAddr == src){
//Only choice 'source' if it's the only choice:
if(olsrState.GetNeighbors().size() == 1){
m_dropDest = tuple->neighborMainAddr;
haveNeighborAP = true;
}
else
continue;
}
else {
m_dropDest = tuple->neighborMainAddr;
haveNeighborAP = true;
if(tuple->neighborMainAddr == dest)
break;//The 'destination' is the priority.
}
}
}

return haveNeighborAP;
}

//Verifica se o endereÃ§o 'ipv4address' Ã© AP:
bool
Offloading::isBackbone(Ipv4Address ipv4address)
{
for (std::vector<Ipv4Address >::const_iterator backbone = m_backbone.begin(); backbone != m_backbone.end(); backbone++){
if(*backbone == ipv4address){
return true;
}
}

return false;
}

//Verifica se determinado par de APs jÃ¡ fez Offloading:
bool
Offloading::isOldPacket(Ipv4Address source, int packetID)
{
for(std::vector<std::pair<Ipv4Address, int> >::const_iterator it = m_loopControl.begin(); it != m_loopControl.end(); it++){
if((it->first == source) && (it->second == packetID))
return true;
}
return false;
}

void
Offloading::WritePacket(int sourceID, int pktID, int nextID, int status, double reserved)
{
m_winner_bid = reserved;
//Auxiliary strings. Used to make the file content.
```

```
std::stringstream str;
std::string s;

//BIDs and Fines (used to make accounts):
double earnBID;
double earnFINE;
double payBID;
double payFINE;

switch(status){
case 1://Success
earnBID = 0.0;
payFINE = 0.0;
payBID = 0.0;
earnFINE = 0.0;
break;
case 2://Super fine success
earnBID = 0.0;
payFINE = 0.0;
payBID = 0.0;
earnFINE = 0.0;
break;
case 3://Almost success
earnBID = 0.0;
payFINE = 0.0;
payBID = 0.0;
earnFINE = 0.0;
break;
case 4://Drop
earnBID = m_offerBid;
payFINE = m_payFINE;
payBID = 0.0;
earnFINE = 0.0;
break;
case 5://Normal auction(User only)
earnBID = m_offerBid;
payFINE = m_payFINE;
payBID = m_winner_bid;
earnFINE = m_Fn;
break;
case 6://Last hop(User only)
earnBID = m_offerBid;
payFINE = m_payFINE;
payBID = 0.0;
earnFINE = 0.0;
break;
default:
break;
}

//Fill the Source node id with zeros:
if(sourceID < 10)
content = content + "00" + boost::lexical_cast<std::string>(sourceID) + "\t";
else if(sourceID < 100)
content = content + "0" + boost::lexical_cast<std::string>(sourceID) + "\t";
else
content = content + boost::lexical_cast<std::string>(sourceID) + "\t";

content = content + boost::lexical_cast<std::string>(pktID) + "\t";

if(nextID != -1){
//Fill the Next node id with zeros:
if(nextID < 10)
content = content + "00" + boost::lexical_cast<std::string>(nextID) + "\t";
else if(nextID < 100)
content = content + "0" + boost::lexical_cast<std::string>(nextID) + "\t";
else
content = content + boost::lexical_cast<std::string>(nextID) + "\t";
}
else if((status == 1)||(status == 2)||(status == 3))
content = content + "end" + "\t";
else if(status == 4)
content = content + "drop" + "\t";
else
content = content + "error" + "\t";

//BIDs and Fines:
str << std::fixed << std::setprecision(2) << earnBID;
s = str.str();
content = content + s + "\t";
str.str(std::string());
```

```
str << std::fixed << std::setprecision(2) << payFINE;
s = str.str();
content = content + s + "\t";
str.str(std::string());
str << std::fixed << std::setprecision(2) << payBID;
s = str.str();
content = content + s + "\t";
str.str(std::string());
str << std::fixed << std::setprecision(2) << earnFINE;
s = str.str();
content = content + s + "\t";

//Successful packet balance:
str.str(std::string());
str << std::fixed << std::setprecision(2) << (earnBID - payBID);
s = str.str();
content = content + s + "\t";

//Fail packet balance:
str.str(std::string());
str << std::fixed << std::setprecision(2) << (earnBID - payFINE - payBID + earnFINE);
s = str.str();
content = content + s + "\t";

//Status:
if((status == 1)||(status == 2)||(status == 3)||(status == 4))
content = content + boost::lexical_cast<std::string>(status) + "\t";
else if((status == 5)||(status == 6))
content = content + " " + "\t";//Don't write. Used by the user only.

//Packet balance:
switch(status){
case 1:
str.str(std::string());
str << std::fixed << std::setprecision(2) << (earnBID - payBID);
s = str.str();
content = content + s + "\t";
break;
case 2:
content = content + "       " + "\t";
break;
case 3:
str.str(std::string());
str << std::fixed << std::setprecision(2) << (earnBID - payFINE - payBID + earnFINE);
s = str.str();
content = content + s + "\t";
break;
case 4:
str.str(std::string());
str << std::fixed << std::setprecision(2) << (earnBID - payFINE - payBID + earnFINE);
s = str.str();
content = content + s + "\t";
break;
case 5:
content = content + "       " + "\t";
break;
case 6:
content = content + "       " + "\t";
break;
default:
break;
}

//Accumulative balance (reserved to the telecommunications company):
content = content + "         " + "\t";
content = content + "\n";

report_user = fopen(dir_myreport_user.c_str(), "w+");
fputs(content.c_str(), report_user);
fclose(report_user);
}

} // Namespace ns3
```

I.1.3 offloading-packet.h

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2014 UnB, Departamento de Engenharia ElÃľtrica
```

```
 *
 * 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: Lucas Soares de Brito <lucasbrito573@gmail.com>
 */
#ifndef OFFLOADINGPACKET_H
#define OFFLOADINGPACKET_H

#include <iostream>
#include "ns3/header.h"
#include "ns3/enum.h"
#include "ns3/ipv4-address.h"
#include "ns3/mac48-address.h"
#include <map>
#include "ns3/nstime.h"

namespace ns3 {

enum MessageType
{
  OFFLOADINGTYPE_RFB  = 1,   //!< OFFLOADINGTYPE_RFB
  OFFLOADINGTYPE_BID  = 2,   //!< OFFLOADINGTYPE_BID
  OFFLOADINGTYPE_DATA  = 3,  //!< OFFLOADINGTYPE_DATA
};

/**
* \ingroup offloading
* \brief Offloading types
*/
class TypeHeader : public Header
{
public:
  TypeHeader (MessageType t = OFFLOADINGTYPE_RFB);

  ///\name Header serialization/deserialization
  //\{
  static TypeId GetTypeId ();
  TypeId GetInstanceTypeId () const;
  uint32_t GetSerializedSize () const;
  void Serialize (Buffer::Iterator start) const;
  uint32_t Deserialize (Buffer::Iterator start);
  void Print (std::ostream &os) const;
  //\}

  /// Return type
  MessageType Get () const { return m_type; }
  /// Check that type if valid
  bool IsValid () const { return m_valid; }
  bool operator== (TypeHeader const & o) const;
private:
  MessageType m_type;
  bool m_valid;
};

std::ostream & operator<< (std::ostream & os, TypeHeader const & h);

/**
* \ingroup offloading
* \brief   Request For Bid (RFB) Message Format
  \verbatim
  0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |     Type      |   Hop count   |    Deadline   |    Packet ID  |
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |                               B0                              |
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |                              Fine                             |
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |                       Source IP Address                       |
```

```
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |                     Destination IP Address                    |
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |                     RFB Source MAC Address...                 |
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |  ...RFB Source MAC Address    |
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  \endverbatim
*/

class RFBHeader : public Header
{
public:
  RFBHeader (uint8_t hopcount = 0,  uint8_t deadline = 0, uint8_t packet_id = 0,
     uint32_t b0 = 0.0, uint32_t fine = 0.0,
     Ipv4Address dst = Ipv4Address (),
     Ipv4Address src = Ipv4Address (),
     Mac48Address srcRFB = Mac48Address ());

  ///\name Header serialization/deserialization
  //\{
  static TypeId GetTypeId ();
  TypeId GetInstanceTypeId () const;
  uint32_t GetSerializedSize () const;
  void Serialize (Buffer::Iterator start) const;
  uint32_t Deserialize (Buffer::Iterator start);
  void Print (std::ostream &os) const;
  //\}

  ///\name Fields
  //\{
  void SetHopcount (uint8_t hopcount) { m_hopcount = hopcount; }
  uint8_t GetHopcount () const { return m_hopcount; }
  void SetDeadline (uint8_t deadline) { m_deadline = deadline; }
  uint8_t GetDeadline () const { return m_deadline; }
  void SetPacketID (uint8_t a) { m_packet_id = a; }
  uint16_t GetPacketID () const { return m_packet_id; }
  void SetB0 (uint32_t b0) { m_b0 = b0; }
  uint32_t GetB0 () const { return m_b0; }
  void SetFine (uint32_t fine) { m_fine = fine; }
  uint32_t GetFine () const { return m_fine; }
  void SetDst (Ipv4Address a) { m_dst = a; }
  Ipv4Address GetDst () const { return m_dst; }
  void SetSrc (Ipv4Address a) { m_src = a; }
  Ipv4Address GetSrc () const { return m_src; }
  void SetSrcRFB (Mac48Address a) { m_srcRFB = a; }
  Mac48Address GetSrcRFB () const { return m_srcRFB; }
  //\}

  bool operator== (RFBHeader const & o) const;
private:

  uint8_t        m_hopcount;    ///< Hop Count 'pu'
  uint8_t        m_deadline;    ///< Deadline H0
  uint8_t        m_packet_id;   ///< Packet ID
  uint32_t       m_b0;          ///< Budget B0
  uint32_t       m_fine;        ///< Fine F0
  Ipv4Address    m_dst;         ///< Destination IP Address
  Ipv4Address    m_src;         ///< Source IP Address
  Mac48Address   m_srcRFB;      ///< RFB Source MAC Address
};

std::ostream & operator<< (std::ostream & os, RFBHeader const &);

/**
* \ingroup offloading
* \brief Bid Message Format
  \verbatim
  0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |     Type      |            Reserved           |   Packet ID   |
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |                          Bid Offer                            |
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |                       Source IP Address                       |
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |                     Destination IP Address                    |
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

  \endverbatim
```

```
*/
class BidHeader : public Header
{
public:
  BidHeader (uint16_t reserved = 0, uint8_t packet_id = 0, uint32_t offeredBid = 0,
    Ipv4Address dst = Ipv4Address (),
   Ipv4Address src = Ipv4Address ());

  ///\name Header serialization/deserialization
  //\{
  static TypeId GetTypeId ();
  TypeId GetInstanceTypeId () const;
  uint32_t GetSerializedSize () const;
  void Serialize (Buffer::Iterator start) const;
  uint32_t Deserialize (Buffer::Iterator start);
  void Print (std::ostream &os) const;
  //\}

  ///\name Fields
  //\{
  void SetPacketID (uint8_t a) { m_packet_id = a; }
  uint16_t GetPacketID () const { return m_packet_id; }
  void SetOfferedBid (uint32_t offeredBid) { m_offeredBid = offeredBid; }
  uint32_t GetOfferedBid () const { return m_offeredBid; }
  void SetSrc (Ipv4Address a) { m_src = a; }
  Ipv4Address GetSrc () const { return m_src; }
  void SetDst (Ipv4Address a) { m_dst = a; }
  Ipv4Address GetDst () const { return m_dst; }
  //\}

  bool operator== (BidHeader const & o) const;
private:
  uint16_t       m_reserved;       ///< Not used
  uint8_t        m_packet_id;      ///< Packet ID
  uint32_t       m_offeredBid;     ///< Offered Bid 'O(cn)'
  Ipv4Address    m_src;            ///< Source IP Address
  Ipv4Address    m_dst;            ///< Destination IP Address
};

std::ostream & operator<< (std::ostream & os, BidHeader const &);

/*
 * O nÃş vencedor irÃą receber DADOS e seu endereÃğo de AP fonte.
 *
 * The winner node will receive DATA and its source AP address.
 *
 */

/**
* \ingroup aodv
* \brief Winner (DATA) Message Format
  \verbatim
  0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |     Type      |                     Source IP Address                         |
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |    Packet ID  |                   Destination IP Address                      |
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |    Hop Count  |
  +-+-+-+-+-+-+-+-+

  \endverbatim
*/

class DataHeader : public Header
{
public:
  DataHeader (Ipv4Address src = Ipv4Address (),
     uint8_t packet_id = 0,
     Ipv4Address dst = Ipv4Address (),
     uint8_t hopcount = 0
);

  ///\name Header serialization/deserialization
  //\{
  static TypeId GetTypeId ();
  TypeId GetInstanceTypeId () const;
  uint32_t GetSerializedSize () const;
  void Serialize (Buffer::Iterator i) const;
  uint32_t Deserialize (Buffer::Iterator start);
```

```
  void Print (std::ostream &os) const;
  //\}

  ///\name Fields
  void SetPacketID (uint8_t a) { m_packet_id = a; }
  uint8_t GetPacketID () const { return m_packet_id; }
  void SetDst (Ipv4Address a) { m_dst = a; }
  Ipv4Address GetDst () const { return m_dst; }
  void SetSrc (Ipv4Address a) { m_src = a; }
  Ipv4Address GetSrc () const { return m_src; }
  void SetHopcount (uint8_t hopcount) { m_hopcount = hopcount; }
  uint8_t GetHopcount () const { return m_hopcount; }

  bool operator== (DataHeader const & o) const;
private:

  Ipv4Address    m_src;         ///< Source IP Address
  uint8_t        m_packet_id;   ///< Packet sequence number
  Ipv4Address    m_dst;         ///< Destination IP Address
  uint8_t        m_hopcount;    ///< Number of hops at the moment of DATA send
};
std::ostream & operator<< (std::ostream & os, DataHeader const &);

}
#endif /* OFFLOADINGPACKET_H */
```

I.1.4 offloading-packet.cc

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2014 UnB, Departamento de Engenharia ElÃľtrica
 *
 * 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: Lucas Soares de Brito <lucasbrito573@gmail.com>
 */
#include "ns3/log.h"
#include "offloading-packet.h"
#include "ns3/address-utils.h"
#include "ns3/packet.h"

namespace ns3
{
NS_LOG_COMPONENT_DEFINE ("OffloadingPacket");

NS_OBJECT_ENSURE_REGISTERED (TypeHeader);

TypeHeader::TypeHeader (MessageType t) :
  m_type (t), m_valid (true)
{
}

TypeId
TypeHeader::GetTypeId ()
{
  static TypeId tid = TypeId ("ns3::TypeHeader")
    .SetParent<Header> ()
    .AddConstructor<TypeHeader> ()
  ;
  return tid;
}

TypeId
TypeHeader::GetInstanceTypeId () const
{
  return GetTypeId ();
}
```

```
uint32_t
TypeHeader::GetSerializedSize () const
{
  return 1;
}

void
TypeHeader::Serialize (Buffer::Iterator i) const
{
  i.WriteU8 ((uint8_t) m_type);
}

uint32_t
TypeHeader::Deserialize (Buffer::Iterator start)
{
  Buffer::Iterator i = start;
  uint8_t type = i.ReadU8 ();
  m_valid = true;
  switch (type)
    {
    case OFFLOADINGTYPE_RFB:
    case OFFLOADINGTYPE_BID:
    case OFFLOADINGTYPE_DATA:
      {
        m_type = (MessageType) type;
        break;
      }
    default:
      m_valid = false;
    }
  uint32_t dist = i.GetDistanceFrom (start);
  NS_ASSERT (dist == GetSerializedSize ());
  return dist;
}

void
TypeHeader::Print (std::ostream &os) const
{
  switch (m_type)
    {
    case OFFLOADINGTYPE_RFB:
      {
        os << "RFB";
        break;
      }
    case OFFLOADINGTYPE_BID:
      {
        os << "BID";
        break;
      }
    case OFFLOADINGTYPE_DATA:
      {
        os << "DATA";
        break;
      }
    default:
      os << "UNKNOWN_TYPE";
    }
}

bool
TypeHeader::operator== (TypeHeader const & o) const
{
  return (m_type == o.m_type && m_valid == o.m_valid);
}

std::ostream &
operator<< (std::ostream & os, TypeHeader const & h)
{
  h.Print (os);
  return os;
}

//-----------------------------------------------------------------------------
// RFB
//-----------------------------------------------------------------------------
RFBHeader::RFBHeader (uint8_t hopcount,  uint8_t deadline, uint8_t packet_id,
                  uint32_t b0, uint32_t fine,
                  Ipv4Address dst,
                  Ipv4Address src,
                  Mac48Address srcRFB) :
```

```
          m_hopcount (hopcount), m_deadline (deadline), m_packet_id (packet_id),
          m_b0 (b0), m_fine (fine),
          m_dst (dst),
          m_src (src),
          m_srcRFB (srcRFB)
{
}

NS_OBJECT_ENSURE_REGISTERED (RFBHeader);

TypeId
RFBHeader::GetTypeId ()
{
  static TypeId tid = TypeId ("ns3::RFBHeader")
    .SetParent<Header> ()
    .AddConstructor<RFBHeader> ()
  ;
  return tid;
}

TypeId
RFBHeader::GetInstanceTypeId () const
{
  return GetTypeId ();
}

uint32_t
RFBHeader::GetSerializedSize () const
{
  return 25;
}

void
RFBHeader::Serialize (Buffer::Iterator i) const
{
  i.WriteU8 (m_hopcount);
  i.WriteU8 (m_deadline);
  i.WriteU8 (m_packet_id);
  i.WriteHtonU32 (m_b0);
  i.WriteHtonU32 (m_fine);
  WriteTo (i, m_dst);
  WriteTo (i, m_src);
  WriteTo (i, m_srcRFB);
}

uint32_t
RFBHeader::Deserialize (Buffer::Iterator start)
{
  Buffer::Iterator i = start;
  m_hopcount = i.ReadU8 ();
  m_deadline = i.ReadU8 ();
  m_packet_id = i.ReadU8 ();
  m_b0 = i.ReadNtohU32 ();
  m_fine = i.ReadNtohU32 ();
  ReadFrom (i, m_dst);
  ReadFrom (i, m_src);
  ReadFrom (i, m_srcRFB);

  uint32_t dist = i.GetDistanceFrom (start);
  //NS_LOG_DEBUG("DIST=" << dist);
  NS_ASSERT (dist == GetSerializedSize ());
  return dist;
}

void
RFBHeader::Print (std::ostream &os) const
{
  os << " Budget B0 "          << m_b0
     << " Fine F0 "            << m_fine
     << " Deadline H0 "        << m_deadline
     << " Hop Count 'pu' "     << m_hopcount
     << " destination: ipv4 "  << m_dst
     << " source: ipv4 "       << m_src
     << " Packet ID "          << m_packet_id
     << " Source RFB : MAC48 " << m_srcRFB ;
}

std::ostream &
operator<< (std::ostream & os, RFBHeader const & h)
{
  h.Print (os);
```

```
  return os;
}

bool
RFBHeader::operator== (RFBHeader const & o) const
{
  return (m_b0 == o.m_b0 && m_fine == o.m_fine &&
  m_deadline == o.m_deadline && m_hopcount == o.m_hopcount
  && m_dst == o.m_dst
  && m_src == o.m_src
  && m_packet_id == o.m_packet_id
  && m_srcRFB == o.m_srcRFB);
}

//-----------------------------------------------------------------------------
// Bid
//-----------------------------------------------------------------------------

BidHeader::BidHeader (uint16_t reserved, uint8_t packet_id,
              uint32_t offeredBid,
               Ipv4Address src, Ipv4Address dst):
m_reserved (reserved), m_packet_id (packet_id),
  m_offeredBid (offeredBid),
  m_src(src), m_dst(dst)
{
}

NS_OBJECT_ENSURE_REGISTERED (BidHeader);

TypeId
BidHeader::GetTypeId ()
{
  static TypeId tid = TypeId ("ns3::BidHeader")
    .SetParent<Header> ()
    .AddConstructor<BidHeader> ()
  ;
  return tid;
}

TypeId
BidHeader::GetInstanceTypeId () const
{
  return GetTypeId ();
}

uint32_t
BidHeader::GetSerializedSize () const
{
  return 15;
}

void
BidHeader::Serialize (Buffer::Iterator i) const
{
  i.WriteU16 (m_reserved);
  i.WriteU8 (m_packet_id);
  i.WriteHtonU32 (m_offeredBid);
  WriteTo (i, m_src);
  WriteTo (i, m_dst);
}

uint32_t
BidHeader::Deserialize (Buffer::Iterator start)
{
  Buffer::Iterator i = start;
  m_reserved = i.ReadU16 ();
  m_packet_id = i.ReadU8 ();
  m_offeredBid = i.ReadNtohU32 ();
  ReadFrom (i, m_src);
  ReadFrom (i, m_dst);

  uint32_t dist = i.GetDistanceFrom (start);
  //NS_LOG_DEBUG("DIST=" << dist);
  NS_ASSERT (dist == GetSerializedSize ());
  return dist;
}

void
BidHeader::Print (std::ostream &os) const
{
  os << " Lance: "  << m_offeredBid
```

```
 << " Packet ID: "  << m_packet_id
     << " source: "  << m_src
     << " destination: "  << m_dst;
}

std::ostream &
operator<< (std::ostream & os, BidHeader const & h)
{
  h.Print (os);
  return os;
}

bool
BidHeader::operator== (BidHeader const & o) const
{
  return (m_offeredBid == o.m_offeredBid &&
          m_reserved == o.m_reserved && m_packet_id == o.m_packet_id
          && m_src == o.m_src && m_dst == o.m_dst);
}

//-----------------------------------------------------------------------------
// Data (Winner)
//-----------------------------------------------------------------------------

/*
 * O nÃş vencedor irÃą receber somente DADOS.
 * Se quiser adicionar algum cabeÃğalho, remova os comentÃąrios abaixo e faÃğa suas alteraÃğÃţes.
 *
 * The winner node will only receive DATA.
 * If you want to add some header, remove comments below and make your changes.
 *
 */

DataHeader::DataHeader (Ipv4Address src,
uint8_t packet_id,
        Ipv4Address dst,
        uint8_t hopcount
        ) :
m_src (src),
     m_packet_id (packet_id),
     m_dst (dst),
     m_hopcount (hopcount)
{
}

NS_OBJECT_ENSURE_REGISTERED (DataHeader);
TypeId
DataHeader::GetTypeId ()
{
  static TypeId tid = TypeId ("ns3::DataHeader")
    .SetParent<Header> ()
    .AddConstructor<DataHeader> ()
  ;
  return tid;
}

TypeId
DataHeader::GetInstanceTypeId () const
{
  return GetTypeId ();
}

uint32_t
DataHeader::GetSerializedSize () const
{
  return 10;
}

void
DataHeader::Serialize (Buffer::Iterator i ) const
{
  WriteTo (i, m_src);
  i.WriteU8 (m_packet_id);
  WriteTo (i, m_dst);
  i.WriteU8 (m_hopcount);
}

uint32_t
DataHeader::Deserialize (Buffer::Iterator start )
{
  Buffer::Iterator i = start;
```

```
  ReadFrom (i, m_src);
  m_packet_id = i.ReadU8 ();
  ReadFrom (i, m_dst);
  m_hopcount = i.ReadU8 ();
  uint32_t dist = i.GetDistanceFrom (start);
  NS_ASSERT (dist == GetSerializedSize ());
  return dist;
}

void
DataHeader::Print (std::ostream &os ) const
{
os << " source: ipv4 "   << m_src
   << " destination: ipv4 " << m_dst
   << " packet ID number: " << m_packet_id
   << " hop count: "     << m_hopcount;
}

std::ostream &
operator<< (std::ostream & os, DataHeader const & h )
{
  h.Print (os);
  return os;
}

bool
DataHeader::operator== (DataHeader const & o ) const
{
  return (m_src == o.m_src
  && m_packet_id == o.m_packet_id
  && m_dst == o.m_dst
  && m_hopcount == o.m_hopcount);
}

}
```

I.2 Application Helper

I.2.1 offloading-helper.h

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2008 INRIA
 *
 * 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: Mathieu Lacage <mathieu.lacage@sophia.inria.fr>
 */
#ifndef OFFLOADING_HELPER_H
#define OFFLOADING_HELPER_H

#include <stdint.h>
#include <map>
#include "ns3/application-container.h"
#include "ns3/node-container.h"
#include "ns3/object-factory.h"
#include "ns3/ipv4-address.h"
#include "ns3/ipv6-address.h"

namespace ns3 {

/**
 * \brief create an application which sends a udp packet and waits for an echo of this packet
 */
class OffloadingHelper
{
```

```
public:
  /**
   * Create OffloadingHelper which will make life easier for people trying
   * to set up simulations with biddings.
   *
   * \param ip The IP address of the remote udp echo server
   * \param port The port number of the remote udp echo server
   */
  OffloadingHelper (uint16_t port);
  //OffloadingHelper (Ipv4Address ip, uint16_t port);

  /**
   * Record an attribute to be set in each Application after it is is created.
   *
   * \param name the name of the attribute to set
   * \param value the value of the attribute to set
   */
  void SetAttribute (std::string name, const AttributeValue &value);

  /**
   * Create a udp echo client application on the specified node.  The Node
   * is provided as a Ptr<Node>.
   *
   * \param node The Ptr<Node> on which to create the UdpEchoClientApplication.
   *
   * \returns An ApplicationContainer that holds a Ptr<Application> to the
   *          application created
   */
  ApplicationContainer Install (Ptr<Node> node) const;

  /**
   * Create a udp echo client application on the specified node.  The Node
   * is provided as a string name of a Node that has been previously
   * associated using the Object Name Service.
   *
   * \param nodeName The name of the node on which to create the UdpEchoClientApplication
   *
   * \returns An ApplicationContainer that holds a Ptr<Application> to the
   *          application created
   */
  ApplicationContainer Install (std::string nodeName) const;

  /**
   * \param c the nodes
   *
   * Create one udp echo client application on each of the input nodes
   *
   * \returns the applications created, one application per input node.
   */
  ApplicationContainer Install (NodeContainer c) const;

  /**
   * Pass the "Tightness" strategy parameters to 'node'. The Install() method should have previously been
   * called by the user.
   *
   * \param node the node to change the parameters
   * \param k1 constant that multiply the Budget at the preference function
   * \param k2 constant that multiply the Relative Tightness at the preference function
   * \param budget_percentage percentage that multiply the offered Bid to send at a new RFB (next auction)
   * \param fine_percentage percentage that multiply the new Budget(Bn = budget_percentage*offerBid) to send at a new RFB (next auction)
   */
  void SetTightnessParameters (Ptr<Node> node, double k1, double k2, double budget_percentage, double fine_percentage);

  /**
   * Pass the Backbone addresses to 'node'. The Install() method should have previously been
   * called by the user.
   *
   * \param node the node to receive Backbone addresses
   * \param backboneAddr AP address of the backbone
   */
  void SetBackbone (Ptr<Node> node, std::vector<Ipv4Address > backbone);

  /**
   * Pass the nodes IDs and its respective IPv4 addresses to the node container 'c'. The Install() method should have previously been
   * called by the user.
   *
   * \param c node container
   * \param nodes_id map nodes (node address --> node id)
   */
  void SetMapNodes (NodeContainer c, std::map<Ipv4Address, int> nodes_id);
```

```
  /**
   * Pass the topology name to the node container 'c'. The Install() method should have previously been
   * called by the user.
   *
   * \param c node container
   * \param topologyName topology name
   */
  void SetTopologyName (NodeContainer c, std::string topologyName);

  /**
   * Pass the seed index to the node container 'c'. The Install() method should have previously been
   * called by the user.
   *
   * \param c node container
   * \param seedIndex seed index
   */
  void SetSeedIndex (NodeContainer c, int seedIndex);

  /**
   * Pass the parameters folder name to the node container 'c'. The Install() method should have previously been
   * called by the user.
   *
   * \param c node container
   * \param paramName parameters folder name
   */
  void SetParamName (NodeContainer c, std::string paramName);

  /**
   * Pass the experiment index to the node container 'c'. The Install() method should have previously been
   * called by the user.
   *
   * \param c node container
   * \param expIndex experiment index
   */
  void SetExpIndex (NodeContainer c, std::string expIndex);

private:
  Ptr<Application> InstallPriv (Ptr<Node> node) const;
  ObjectFactory m_factory;
};

} // namespace ns3

#endif /* OFFLOADING_HELPER_H */
```

I.2.2 offloading-helper.cc

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2008 INRIA
 *
 * 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: Mathieu Lacage <mathieu.lacage@sophia.inria.fr>
 */
#include "offloading-helper.h"
#include "ns3/offloading.h"
#include "ns3/uinteger.h"
#include "ns3/names.h"

namespace ns3 {

OffloadingHelper::OffloadingHelper (uint16_t port)
{
  m_factory.SetTypeId (Offloading::GetTypeId ());
  //SetAttribute ("RemoteAddress", AddressValue (address));
  SetAttribute ("RemotePort", UintegerValue (port));
}
```

```
/*
OffloadingHelper::OffloadingHelper (Ipv4Address address, uint16_t port)
{
  m_factory.SetTypeId (Offloading::GetTypeId ());
  //SetAttribute ("RemoteAddress", AddressValue (Address(address)));
  SetAttribute ("RemotePort", UintegerValue (port));
}
*/

void
OffloadingHelper::SetAttribute (
  std::string name,
  const AttributeValue &value)
{
  m_factory.Set (name, value);
}

ApplicationContainer
OffloadingHelper::Install (Ptr<Node> node) const
{
  return ApplicationContainer (InstallPriv (node));
}

ApplicationContainer
OffloadingHelper::Install (std::string nodeName) const
{
  Ptr<Node> node = Names::Find<Node> (nodeName);
  return ApplicationContainer (InstallPriv (node));
}

ApplicationContainer
OffloadingHelper::Install (NodeContainer c) const
{
  ApplicationContainer apps;
  for (NodeContainer::Iterator i = c.Begin (); i != c.End (); ++i)
    {
      apps.Add (InstallPriv (*i));
    }

  return apps;
}

void
OffloadingHelper::SetBackbone (Ptr<Node> node, std::vector<Ipv4Address > backbone){
Ptr<Offloading> offloading = DynamicCast<Offloading> (node->GetApplication (0));
offloading->SetBackbone(backbone);
}

void
OffloadingHelper::SetTightnessParameters (Ptr<Node> node, double k1, double k2, double budget_percentage, double fine_percentage){
Ptr<Offloading> offloading = DynamicCast<Offloading> (node->GetApplication (0));
offloading->SetTightnessParameters (k1, k2, budget_percentage, fine_percentage);
}

void
OffloadingHelper::SetMapNodes (NodeContainer c, std::map<Ipv4Address, int> nodes_id)
{
  for (NodeContainer::Iterator i = c.Begin (); i != c.End (); ++i)
    {
  Ptr<Offloading> offloading = DynamicCast<Offloading> ((*i)->GetApplication (0));
  offloading->SetMapNodes(nodes_id);
    }
}

void
OffloadingHelper::SetTopologyName (NodeContainer c, std::string topologyName)
{
  for (NodeContainer::Iterator i = c.Begin (); i != c.End (); ++i)
    {
  Ptr<Offloading> offloading = DynamicCast<Offloading> ((*i)->GetApplication (0));
  offloading->SetTopologyName(topologyName);
    }
}

void
OffloadingHelper::SetSeedIndex (NodeContainer c, int seedIndex)
{
  for (NodeContainer::Iterator i = c.Begin (); i != c.End (); ++i)
    {
  Ptr<Offloading> offloading = DynamicCast<Offloading> ((*i)->GetApplication (0));
```

```
  offloading->SetSeedIndex(seedIndex);
    }
}

void
OffloadingHelper::SetParamName (NodeContainer c, std::string paramName)
{
  for (NodeContainer::Iterator i = c.Begin (); i != c.End (); ++i)
    {
  Ptr<Offloading> offloading = DynamicCast<Offloading> ((*i)->GetApplication (0));
  offloading->SetParamName(paramName);
    }
}

void
OffloadingHelper::SetExpIndex (NodeContainer c, std::string expIndex)
{
  for (NodeContainer::Iterator i = c.Begin (); i != c.End (); ++i)
    {
  Ptr<Offloading> offloading = DynamicCast<Offloading> ((*i)->GetApplication (0));
  offloading->SetExpIndex(expIndex);
    }
}

Ptr<Application>
OffloadingHelper::InstallPriv (Ptr<Node> node) const
{
  Ptr<Application> app = m_factory.Create<Offloading> ();
  node->AddApplication (app);

  return app;
}

} // namespace ns3
```

I.3 Main Script

I.3.1 offloadingScript.cc

```
#include "ns3/wifi-module.h"
#include "ns3/mobility-module.h"
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/applications-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/olsr-helper.h"
#include "ns3/flow-monitor-module.h"
#include "ns3/random-variable.h"
#include <cmath>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <string>
#include <ctime>
#include <cstdio>
#include <cstdlib>
#include <stdexcept>
#include <boost/lexical_cast.hpp>
//#include <math.h>
#include <sys/stat.h>
#include <sys/types.h>

using namespace ns3;
using namespace std;

NS_LOG_COMPONENT_DEFINE ("offloadingScript");

// MAC and PHY layers data:
bool useTwoRay = false;
double txPowerDbm = 16.0206;
double distancia = 150;//max distance
double altura = 1.0;//height
const double lambda = (299792458 / 2.407e9); // v = lambda(meters) * freq --> v = speed of light(ns3 default value), freq = 2.4 GHz(WiFi)

//Nodes coordinates and number of nodes:
```

```
float *x_readTopo, *y_readTopo;
int total_nodes, backbone_size, offloading_size;

//Variables used to configure the AP nodes:
double *initialBudget, *initialFine, *startGen;
int *deadline, *Npackets, *destBackbone;

//Tightness parameters:
double *k1, *k2, *budget_percentage, *fine_percentage;

//Backbone vector (used to "inform" to all the nodes which are the AP nodes):
std::vector<Ipv4Address > backbone;

//Node map (used to map all nodes and its respective IPv4 addresses):
std::map<Ipv4Address, int> nodes_id;

void readConfig (char *topofile_backbone, char *topofile_offloading, char *configfile_backbone, char *configfile_tightness);
double rxPowerDbm (double distance, double height, double txPowerDbm, bool useTwoRay);

int
main (int argc, char *argv[])
{
GlobalValue::Bind ("ChecksumEnabled", BooleanValue (true));
//LogComponentEnable ("OlsrRoutingProtocol", LOG_LEVEL_INFO);
LogComponentEnable ("OffloadingApplication", LOG_LEVEL_DEBUG);
LogComponentEnable ("OffloadingPacket", LOG_LEVEL_DEBUG);

std::string home_dir(getenv("HOME"));
std::string topologyName("");
std::string paramName("");
std::string expIndex = "";
int seedIndex = 0;
int seedValue = 0;
double simStop = 5200.0;
double appStart = 30.0;
double appStop = simStop - 1.0;
int strategy = STRATEGYTYPE_TIGHTNESS;
int preference_function = PREFERENCEFUNCTION_PLANE;
double mobility_speed = 0.0;

CommandLine cmd;
cmd.AddValue("topologyName","The topology selected", topologyName);
cmd.AddValue("seedIndex","The seed selected", seedIndex);
cmd.AddValue("expIndex","The experiment index", expIndex);
cmd.AddValue("seedValue","The seed value", seedValue);
cmd.AddValue("paramName","The parameters folder name selected", paramName);
cmd.AddValue("preferenceFunctionType","The type of preference function used", preference_function);
cmd.AddValue("strategyType","The type of strategy used", strategy);
cmd.AddValue("mobilitySpeed","The speed of mobility used. Static if zero.", mobility_speed);
cmd.Parse (argc, argv);

char topofile_backbone[200], topofile_offloading[200], configfile_backbone[200], configfile_tightness[200]; //Files path name
if((strategy == STRATEGYTYPE_TIGHTNESS) && (preference_function == PREFERENCEFUNCTION_PLANE)){
sprintf (topofile_backbone,"%s%s%s%s%s%d%s%s%s", getenv("HOME"), "/Dropbox/unb/mestrado/tese/simulations/topology_config/exp", expIndex.c_str(), "/",
paramName.c_str(), "/seed", seedIndex, "/tp",topologyName.c_str(),"/backbone/topo.dat");
sprintf (topofile_offloading,"%s%s%s%s%s%d%s%s%s", getenv("HOME"), "/Dropbox/unb/mestrado/tese/simulations/topology_config/exp", expIndex.c_str(), "/",
paramName.c_str(), "/seed", seedIndex, "/tp",topologyName.c_str(),"/network/topo.dat");
sprintf (configfile_backbone,"%s%s%s%s%s%d%s%s%s", getenv("HOME"), "/Dropbox/unb/mestrado/tese/simulations/topology_config/exp", expIndex.c_str(), "/",
paramName.c_str(), "/seed", seedIndex, "/tp",topologyName.c_str(),"/backbone/config.dat");
sprintf (configfile_tightness,"%s%s%s%s%s%d%s%s%s", getenv("HOME"), "/Dropbox/unb/mestrado/tese/simulations/topology_config/exp", expIndex.c_str(), "/",
paramName.c_str(), "/seed", seedIndex, "/tp",topologyName.c_str(),"/network/config.dat");
}
else{
sprintf (topofile_backbone,"%s%s%s%d%s%s", getenv("HOME"), "/Dropbox/unb/mestrado/tese/simulations/topology_config/exp", expIndex.c_str(), "/seed",
seedIndex, "/tp",topologyName.c_str(),"/backbone/topo.dat");
sprintf (topofile_offloading,"%s%s%s%d%s%s", getenv("HOME"), "/Dropbox/unb/mestrado/tese/simulations/topology_config/exp", expIndex.c_str(), "/seed",
seedIndex, "/tp",topologyName.c_str(),"/network/topo.dat");
sprintf (configfile_backbone,"%s%s%s%d%s%s", getenv("HOME"), "/Dropbox/unb/mestrado/tese/simulations/topology_config/exp", expIndex.c_str(), "/seed",
seedIndex, "/tp",topologyName.c_str(),"/backbone/config.dat");
sprintf (configfile_tightness,"%s%s%s%d%s%s", getenv("HOME"), "/Dropbox/unb/mestrado/tese/simulations/topology_config/exp", expIndex.c_str(), "/seed",
seedIndex, "/tp",topologyName.c_str(),"/network/config.dat");
}

readConfig (topofile_backbone, topofile_offloading, configfile_backbone, configfile_tightness);
std::cout << "Function readConfig terminated." << endl;
std::cout << "backbone_size = " << backbone_size << "\n";
std::cout << "offloading_size = " << offloading_size << "\n";
std::cout << "total_nodes = " << total_nodes << "\n";

//Creating all nodes:
NodeContainer nodes;
```

```
nodes.Create(total_nodes);

// Topology:
MobilityHelper mobility_backbone;
Ptr<ListPositionAllocator> positionAlloc_backbone = CreateObject<ListPositionAllocator> ();
for(int j=0;j<backbone_size;j++) positionAlloc_backbone->Add (Vector (x_readTopo[j],y_readTopo[j],altura)); //backbone node position allocation
mobility_backbone.SetPositionAllocator (positionAlloc_backbone);
mobility_backbone.SetMobilityModel ("ns3::ConstantPositionMobilityModel");

MobilityHelper mobility_network;
Ptr<ListPositionAllocator> positionAlloc_network = CreateObject<ListPositionAllocator> ();
for(int j=backbone_size;j<total_nodes;j++) positionAlloc_network->Add (Vector (x_readTopo[j],y_readTopo[j],altura)); //network node position allocation
mobility_network.SetPositionAllocator (positionAlloc_network);

std::string speed_attribute;
// Mobility Model
if(mobility_speed != 0.0){
speed_attribute = "ns3::ConstantRandomVariable[Constant=" + boost::lexical_cast<std::string>(mobility_speed) + "]";
mobility_network.SetMobilityModel ("ns3::RandomWalk2dMobilityModel","Bounds", RectangleValue (Rectangle (0, 800, 0, 800)),
"Distance", DoubleValue(10.0),
"Speed", StringValue(speed_attribute));
}
else{
mobility_network.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
}

int node_count = 0;
for(NodeContainer::Iterator it = nodes.Begin();it!=nodes.End();it++, node_count++){
if(node_count<backbone_size)
mobility_backbone.Install(*it);
else
mobility_network.Install(*it);
}
std::cout << "Mobility installed (mobility speed = " << mobility_speed << ")" << endl;

// MAC and PHY: WiFi Ad Hoc:

//Change seed:
uint32_t oldSeed = RngSeedManager::GetSeed();
srand(seedValue);
RngSeedManager::SetSeed(rand());

// MAC:
NqosWifiMacHelper wifiMac = NqosWifiMacHelper::Default ();
wifiMac.SetType ("ns3::AdhocWifiMac");

// PHY:
YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ();
wifiPhy.Set ("TxPowerStart", DoubleValue(txPowerDbm));
wifiPhy.Set ("TxPowerEnd", DoubleValue(txPowerDbm));
wifiPhy.Set ("TxGain", DoubleValue(0));
wifiPhy.Set ("RxGain", DoubleValue(0));
wifiPhy.Set ("EnergyDetectionThreshold", DoubleValue(rxPowerDbm (distancia, altura, txPowerDbm, useTwoRay)));
wifiPhy.Set ("CcaMode1Threshold", DoubleValue(rxPowerDbm (distancia*1.5, altura, txPowerDbm, useTwoRay)));

// Channel:
YansWifiChannelHelper wifiChannel;
wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
if(!useTwoRay)
wifiChannel.AddPropagationLoss ("ns3::FriisPropagationLossModel",
"Frequency", DoubleValue(2.407e9));
else
wifiChannel.AddPropagationLoss ("ns3::TwoRayGroundPropagationLossModel",
"Frequency", DoubleValue(2.407e9));
wifiPhy.SetChannel (wifiChannel.Create ());

// WiFi Helper
WifiHelper wifi = WifiHelper::Default ();
wifi.SetStandard (WIFI_PHY_STANDARD_80211g);
wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
"DataMode", StringValue ("DsssRate1Mbps"),
"RtsCtsThreshold", UintegerValue (1000));

NetDeviceContainer devices = wifi.Install (wifiPhy, wifiMac, nodes);
std::cout << "MAC and PHY installed" << endl;

RngSeedManager::SetSeed(oldSeed);

// OLSR
OlsrHelper olsr;
```

```
// Internet stack, with OLSR as routing protocol
InternetStackHelper stack;
stack.SetRoutingHelper (olsr);
//stack.Install (nodesource);
stack.Install (nodes);
Ptr<OutputStreamWrapper> routingStream = Create<OutputStreamWrapper> ("routes.log", std::ios::out);
olsr.PrintRoutingTableAllEvery (Seconds (15), routingStream); //Useful to estimate stabilization time.

// IP addressing
Ipv4AddressHelper address;
address.SetBase ("10.0.0.0", "255.0.0.0");
Ipv4InterfaceContainer interfaces = address.Assign (devices);
std::cout << "IPv4Adress installed" << endl;

//Aplication: offloading

//Write backbone and respective initial budgets (B0) on file (used to make account):
FILE *backbone_file;
std::string dir_mybackbone , filename_report, content;
dir_mybackbone = home_dir + "/Dropbox/unb/mestrado/tese/simulations/";
filename_report = "backbone";
dir_mybackbone = dir_mybackbone + filename_report + ".txt";

std::stringstream str;
std::string s;
for(int i = 0; i<backbone_size;i++){
str << std::fixed << std::setprecision(2) << initialBudget[i];
s = str.str();
//Fill the backbone node id with zeros:
if(i < 10)
content = content + "00" + boost::lexical_cast<std::string>(i) + "\t" + s;
else if(i < 100)
content = content + "0" + boost::lexical_cast<std::string>(i) + "\t" + s;
else
content = content + boost::lexical_cast<std::string>(i) + "\t" + s;

if(i < (backbone_size - 1))
content = content + "\n";

str.str(std::string());
}

backbone_file = fopen(dir_mybackbone.c_str(), "w+");
fputs(content.c_str(), backbone_file);
fclose(backbone_file);

//Initialization of backbone vector:
for(int i = 0;i<backbone_size;i++)
backbone.push_back(((nodes.Get(i)->GetObject<Ipv4>())->GetAddress(1,0)).GetLocal());

std::cout << "Backbone stored" << endl;

//'Helper' creation:
OffloadingHelper offloadinghelper (9);
offloadinghelper.SetAttribute ("StrategyType" , UintegerValue (strategy)); //Choice strategy.
offloadinghelper.SetAttribute ("PreferenceFunctionType" , UintegerValue (preference_function)); //Choice preference function type.
offloadinghelper.SetAttribute ("NumberNodes"    , IntegerValue(total_nodes));

//Declare 'Application':
ApplicationContainer offloadingApp;
std::cout << "Application container created" << endl;
std::cout << endl;

//Install 'applications' at nodes:
int i = 0;
for(NodeContainer::Iterator it = nodes.Begin() ; it != nodes.End() ; it++, i++){
//Mapping node id and address:
Ptr<Ipv4> ipv4 = (*it)->GetObject<Ipv4>();
Ipv4InterfaceAddress iaddr = ipv4->GetAddress(1,0);
nodes_id[iaddr.GetLocal()] = (*it)->GetId();

if(i<backbone_size){
//Backbone nodes configuration:
offloadinghelper.SetAttribute ("NPackets"  , IntegerValue(Npackets[i]));
if(Npackets[i] > 0){
offloadinghelper.SetAttribute ("Budget"              , DoubleValue(initialBudget[i]));
offloadinghelper.SetAttribute ("Fine"        , DoubleValue(initialFine[i]));
offloadinghelper.SetAttribute ("Deadline"            , UintegerValue(deadline[i]));
offloadinghelper.SetAttribute ("DestinationAddress"  , Ipv4AddressValue(Ipv4Address::ConvertFrom(((nodes.Get(destBackbone[i])->
GetObject<Ipv4>())->GetAddress(1,0)).GetLocal())));
offloadinghelper.SetAttribute ("StartOffloading"     , DoubleValue(startGen[i]));
```

```
}
}

//Install nodes:
offloadingApp.Add(offloadinghelper.Install (*it));
offloadinghelper.SetBackbone(*it, backbone);
if(i >= backbone_size)
offloadinghelper.SetTightnessParameters(*it, k1[i - backbone_size], k2[i - backbone_size], budget_percentage[i - backbone_size], fine_percentage[i - backbone_size]);
}
std::cout << "Application installed." << endl;
offloadinghelper.SetMapNodes(nodes, nodes_id);
offloadinghelper.SetTopologyName(nodes, topologyName);
offloadinghelper.SetSeedIndex(nodes, seedIndex);
offloadinghelper.SetParamName(nodes, paramName);
offloadinghelper.SetExpIndex(nodes, expIndex);

offloadingApp.Start (Seconds (appStart));
offloadingApp.Stop (Seconds (appStop));

// Tracing:
std::string trace_dir;
if((strategy == STRATEGYTYPE_TIGHTNESS) && (preference_function == PREFERENCEFUNCTION_PLANE))
trace_dir = home_dir + "/Dropbox/unb/mestrado/tese/simulations/topology_config/exp" + expIndex + "/" + paramName + "/seed" +
boost::lexical_cast<std::string>(seedIndex) + "/tp" + topologyName + "/traces/";
else
trace_dir = home_dir + "/Dropbox/unb/mestrado/tese/simulations/topology_config/exp" + expIndex + "/seed" + boost::lexical_cast<std::string>(seedIndex) +
"/tp" + topologyName + "/traces/";

std::string pcap_dir = trace_dir + "pcap/node";
//wifiPhy.EnablePcapAll (pcap_dir, false);

/*
AsciiTraceHelper ascii;
std::string ascii_dir = trace_dir + "ascii/";
//Generate a log from IP layer
std::string ascii_ipv4 = ascii_dir + "ipv4.tr";
Ptr<OutputStreamWrapper> stream = ascii.CreateFileStream(ascii_ipv4);
stack.EnableAsciiIpv4All(stream);

//Generate a mobility log
AsciiTraceHelper ascii2;
std::string ascii_mob = ascii_dir + "mob.tr";
MobilityHelper::EnableAsciiAll (ascii2.CreateFileStream (ascii_mob));

//Generate a log from PHY layer
AsciiTraceHelper ascii3;
std::string ascii_phy = ascii_dir + "phy.tr";
wifiPhy.EnableAsciiAll (ascii3.CreateFileStream (ascii_phy));

//Generate a log from MAC layer:
std::string ascii_mac = ascii_dir + "mac.tr";
std::ofstream ascii4(ascii_mac);
 */
//std::cout << "Traces generated." << endl;

Simulator::Stop (Seconds(simStop));
Simulator::Run ();

Simulator::Destroy ();
return 0;
}

void
readConfig (char *topofile_backbone, char *topofile_offloading, char *configfile_backbone, char *configfile_tightness)
{
ifstream in_topo_backbone, in_topo_offloading, in_config_backbone, in_config_tightness;
char ch;
backbone_size = 0;
offloading_size = 0;
total_nodes = 0;

//Counting number of lines. Number of lines is the number of AP nodes:
in_topo_backbone.open(topofile_backbone);
if (!in_topo_backbone){
cerr << "Backbone topology file not found" << endl;
return;
}else{
while (in_topo_backbone.get(ch)){
if (ch=='\n'){ backbone_size++;}
}
}
```

```
in_topo_backbone.close ();

//Counting number of lines. Number of lines is the number of offloading nodes:
in_topo_offloading.open(topofile_offloading);
if (!in_topo_offloading){
cerr << "Offloading topology file not found" << endl;
return;
}else{
while (in_topo_offloading.get(ch)){
if (ch=='\n'){ offloading_size++;}
}
}
in_topo_offloading.close ();

//Allocate coordinates vectors (x,y):
total_nodes = backbone_size + offloading_size;
x_readTopo = (float *)malloc(total_nodes * sizeof(float));
y_readTopo = (float *)malloc(total_nodes * sizeof(float));

//Backbone parameters:
initialBudget = (double *)malloc(backbone_size * sizeof(double));
initialFine = (double *)malloc(backbone_size * sizeof(double));
deadline = (int *)malloc(backbone_size * sizeof(int));
Npackets = (int *)malloc(backbone_size * sizeof(int));
destBackbone = (int *)malloc(backbone_size * sizeof(int));
startGen = (double *)malloc(backbone_size * sizeof(double));

//Tightness parameters:
k1 = (double *)malloc(offloading_size * sizeof(double));
k2 = (double *)malloc(offloading_size * sizeof(double));
budget_percentage = (double *)malloc(offloading_size * sizeof(double));
fine_percentage = (double *)malloc(offloading_size * sizeof(double));

//Read AP nodes coordinates and store at (x,y):
in_topo_backbone.open(topofile_backbone);
if (!in_topo_backbone){ cerr << "Backbone topology file not found!" << endl; }
else{
while (in_topo_backbone){
for(int i=0; i<backbone_size; i++){in_topo_backbone >> x_readTopo[i] >> y_readTopo[i];}
in_topo_backbone.close ();
}
}
in_topo_backbone.close ();

//Read offloading nodes coordinates and store at (x,y), after the last AP coordinate:
in_topo_offloading.open(topofile_offloading);
if (!in_topo_offloading){ cerr << "Offloading topology file not found!" << endl; }
else{
while (in_topo_offloading){
for(int i=backbone_size; i<total_nodes; i++){in_topo_offloading >> x_readTopo[i] >> y_readTopo[i];}
in_topo_offloading.close ();
}
}
in_topo_offloading.close ();

//Read AP configurations:
in_config_backbone.open(configfile_backbone);
if (!in_config_backbone){ cerr << "Backbone Configuration file not found!" << endl; }
else{
while (in_config_backbone){
for(int i=0; i<backbone_size; i++){in_config_backbone >> initialBudget[i] >> initialFine[i] >> deadline[i] >> Npackets[i] >> destBackbone[i] >> startGen[i];}
in_config_backbone.close ();
}
}
in_config_backbone.close ();

//Read Tightness parameters:
in_config_tightness.open(configfile_tightness);
if (!in_config_tightness){ cerr << "Tightness Parameters file not found!" << endl; }
else{
while (in_config_tightness){
for(int i=0; i<offloading_size; i++){in_config_tightness >> k1[i] >> k2[i] >> budget_percentage[i] >> fine_percentage[i];}
in_config_tightness.close ();
}
}
in_config_tightness.close ();

}

double
rxPowerDbm (double distance, double height, double txPowerDbm, bool useTwoRay)
```

```
{
double lossPowerDbm;

if (useTwoRay){
double dCross = (4 * 3.141592 * height * height) / lambda;
if (distance <= dCross){
lossPowerDbm = 10 * log10( lambda*lambda / (16.0 * 3.141592 * 3.141592 * distance*distance));
} else {
lossPowerDbm = 10 * log10( (height*height*height*height) / (distance*distance*distance*distance) );
}
}
else {
lossPowerDbm = 10 * log10( lambda*lambda / (16.0 * 3.141592 * 3.141592 * distance*distance));
}

return txPowerDbm + lossPowerDbm;
}
```

II. NS-3 CHANGELOG

Below we provide the changelog of the NS-3 files using the Unix *diff* command.

II.1 Internet Module

II.1.1 arp-cache.h

```
59a60
>   void Print (void);
207a209,215
>
> public:
>     /**
>      * \brief Get the entry state
>      */
>     ArpCacheEntryState_e GetEntryState (void);
>
```

II.1.2 arp-cache.cc

```
182a183,197
> ArpCache::Print (void)
> {
>   ArpCache::Entry* entry;
>   for (CacheI i = m_arpCache.begin (); i != m_arpCache.end (); i++)
>     {
>       entry = (*i).second;
>       if (entry)
>         {
>           NS_LOG_UNCOND ("mac: " << entry->GetMacAddress() << " ipv4:" << entry->GetIpv4Address() << " State:" << entry->GetEntryState());
>         }
>
>     }
> }
>
> void
432a448,453
> }
>
> ArpCache::Entry::ArpCacheEntryState_e
> ArpCache::Entry::GetEntryState (void)
> {
>  return m_state;
```

II.1.3 arp-l3-protocol.h

```
90d89
<   Ptr<ArpCache> FindCache (Ptr<NetDevice> device);
95a95,97
>
> public:
>   Ptr<ArpCache> FindCache (Ptr<NetDevice> device);
```

II.2 OLSR Model

II.2.1 olsr-routing-protocol.h

```
105a106,113
>   /**
>    * Return the state of OLSR
>    **/
>   OlsrState GetOlsrState ()
>   {
```

```
>    return m_state;
>   }
>
```

II.3 Core Module

II.3.1 make-event.h

```
31a32,36
> template <typename MEM, typename OBJ,
>           typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
> EventImpl * MakeEvent (MEM mem_ptr, OBJ obj,
>                        T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6);
>
52a58,61
> template <typename U1, typename U2, typename U3, typename U4, typename U5, typename U6,
>           typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
> EventImpl * MakeEvent (void (*f)(U1,U2,U3,U4,U5,U6), T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6);
>
273a283,323
> template <typename MEM, typename OBJ,
>           typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
> EventImpl * MakeEvent (MEM mem_ptr, OBJ obj,
>                        T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6)
> {
>   // six argument version
>   class EventMemberImpl6 : public EventImpl
>   {
> public:
>     EventMemberImpl6 (OBJ obj, MEM function, T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6)
>       : m_obj (obj),
>         m_function (function),
>         m_a1 (a1),
>         m_a2 (a2),
>         m_a3 (a3),
>         m_a4 (a4),
>         m_a5 (a5),
>         m_a6 (a6)
>     {
>     }
> protected:
>     virtual ~EventMemberImpl6 ()
>     {
>     }
> private:
>     virtual void Notify (void)
>     {
>       (EventMemberImplObjTraits<OBJ>::GetReference (m_obj).*m_function)(m_a1, m_a2, m_a3, m_a4, m_a5, m_a6);
>     }
>     OBJ m_obj;
>     MEM m_function;
>     typename TypeTraits<T1>::ReferencedType m_a1;
>     typename TypeTraits<T2>::ReferencedType m_a2;
>     typename TypeTraits<T3>::ReferencedType m_a3;
>     typename TypeTraits<T4>::ReferencedType m_a4;
>     typename TypeTraits<T5>::ReferencedType m_a5;
>     typename TypeTraits<T6>::ReferencedType m_a6;
>   } *ev = new EventMemberImpl6 (obj, mem_ptr, a1, a2, a3, a4, a5, a6);
>   return ev;
> }
>
438a489,528
>   return ev;
> }
>
> template <typename U1, typename U2, typename U3, typename U4, typename U5, typename U6,
>           typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
> EventImpl * MakeEvent (void (*f)(U1,U2,U3,U4,U5,U6), T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6)
> {
>   // six arg version
>   class EventFunctionImpl6 : public EventImpl
>   {
> public:
>     typedef void (*F)(U1,U2,U3,U4,U5,U6);
>
>     EventFunctionImpl6 (F function, T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6)
>       : m_function (function),
```

```
>         m_a1 (a1),
>         m_a2 (a2),
>         m_a3 (a3),
>         m_a4 (a4),
>         m_a5 (a5),
>         m_a6 (a6)
>     {
>     }
> protected:
>     virtual ~EventFunctionImpl6 ()
>     {
>     }
> private:
>     virtual void Notify (void)
>     {
>       (*m_function)(m_a1, m_a2, m_a3, m_a4, m_a5, m_a6);
>     }
>     F m_function;
>     typename TypeTraits<T1>::ReferencedType m_a1;
>     typename TypeTraits<T2>::ReferencedType m_a2;
>     typename TypeTraits<T3>::ReferencedType m_a3;
>     typename TypeTraits<T4>::ReferencedType m_a4;
>     typename TypeTraits<T5>::ReferencedType m_a5;
>     typename TypeTraits<T6>::ReferencedType m_a6;
>   } *ev = new EventFunctionImpl6 (f, a1, a2, a3, a4, a5, a6);
```

II.3.2 simulator.h

```
210a211,229
>
>   /**
>    * @param time the relative expiration time of the event.
>    * @param mem_ptr member method pointer to invoke
>    * @param obj the object on which to invoke the member method
>    * @param a1 the first argument to pass to the invoked method
>    * @param a2 the second argument to pass to the invoked method
>    * @param a3 the third argument to pass to the invoked method
>    * @param a4 the fourth argument to pass to the invoked method
>    * @param a5 the fifth argument to pass to the invoked method
>    * @param a6 the sixth argument to pass to the invoked method
>    * @returns an id for the scheduled event.
>    */
>   template <typename MEM, typename OBJ,
>             typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
>   static EventId Schedule (Time const &time, MEM mem_ptr, OBJ obj,
>                            T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6);
>
>
275a295,309
>    * @param time the relative expiration time of the event.
>    * @param f the function to invoke
>    * @param a1 the first argument to pass to the function to invoke
>    * @param a2 the second argument to pass to the function to invoke
>    * @param a3 the third argument to pass to the function to invoke
>    * @param a4 the fourth argument to pass to the function to invoke
>    * @param a5 the fifth argument to pass to the function to invoke
>    * @param a6 the sixth argument to pass to the function to invoke
>    * @returns an id for the scheduled event.
>    */
>   template <typename U1, typename U2, typename U3, typename U4, typename U5, typename U6,
>             typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
>   static EventId Schedule (Time const &time, void (*f)(U1,U2,U3,U4,U5,U6), T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6);
>
>   /**
360a395,414
>
>   /**
>    * This method is thread-safe: it can be called from any thread.
>    *
>    * @param time the relative expiration time of the event.
>    * @param context user-specified context parameter
>    * @param mem_ptr member method pointer to invoke
>    * @param obj the object on which to invoke the member method
>    * @param a1 the first argument to pass to the invoked method
>    * @param a2 the second argument to pass to the invoked method
>    * @param a3 the third argument to pass to the invoked method
>    * @param a4 the fourth argument to pass to the invoked method
>    * @param a5 the fifth argument to pass to the invoked method
>    * @param a6 the sixth argument to pass to the invoked method
```

```
>    */
>   template <typename MEM, typename OBJ,
>             typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
>   static void ScheduleWithContext (uint32_t context, Time const &time, MEM mem_ptr, OBJ obj,
>                                    T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6);
>
437a492,508
>    * This method is thread-safe: it can be called from any thread.
>    *
>    * @param time the relative expiration time of the event.
>    * @param context user-specified context parameter
>    * @param f the function to invoke
>    * @param a1 the first argument to pass to the function to invoke
>    * @param a2 the second argument to pass to the function to invoke
>    * @param a3 the third argument to pass to the function to invoke
>    * @param a4 the fourth argument to pass to the function to invoke
>    * @param a5 the fifth argument to pass to the function to invoke
>    * @param a6 the sixth argument to pass to the function to invoke
>    */
>   template <typename U1, typename U2, typename U3, typename U4, typename U5, typename U6,
>             typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
>   static void ScheduleWithContext (uint32_t context, Time const &time, void (*f)(U1,U2,U3,U4,U5,U6), T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6);
>
>   /**
502a574,589
>
>   /**
>    * @param mem_ptr member method pointer to invoke
>    * @param obj the object on which to invoke the member method
>    * @param a1 the first argument to pass to the invoked method
>    * @param a2 the second argument to pass to the invoked method
>    * @param a3 the third argument to pass to the invoked method
>    * @param a4 the fourth argument to pass to the invoked method
>    * @param a5 the fifth argument to pass to the invoked method
>    * @param a6 the sixth argument to pass to the invoked method
>    */
>   template <typename MEM, typename OBJ,
>             typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
>   static EventId ScheduleNow (MEM mem_ptr, OBJ obj,
>                               T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6);
>
558a646,658
>    * @param f the function to invoke
>    * @param a1 the first argument to pass to the function to invoke
>    * @param a2 the second argument to pass to the function to invoke
>    * @param a3 the third argument to pass to the function to invoke
>    * @param a4 the fourth argument to pass to the function to invoke
>    * @param a5 the fifth argument to pass to the function to invoke
>    * @param a6 the sixth argument to pass to the function to invoke
>    */
>   template <typename U1, typename U2, typename U3, typename U4, typename U5, typename U6,
>             typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
>   static EventId ScheduleNow (void (*f)(U1,U2,U3,U4,U5,U6), T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6);
>
>   /**
624a725,740
>
>   /**
>    * @param mem_ptr member method pointer to invoke
>    * @param obj the object on which to invoke the member method
>    * @param a1 the first argument to pass to the invoked method
>    * @param a2 the second argument to pass to the invoked method
>    * @param a3 the third argument to pass to the invoked method
>    * @param a4 the fourth argument to pass to the invoked method
>    * @param a5 the fifth argument to pass to the invoked method
>    * @param a6 the sixth argument to pass to the invoked method
>    */
>   template <typename MEM, typename OBJ,
>             typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
>   static EventId ScheduleDestroy (MEM mem_ptr, OBJ obj,
>                                   T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6);
>
680a797,809
>    * @param f the function to invoke
>    * @param a1 the first argument to pass to the function to invoke
>    * @param a2 the second argument to pass to the function to invoke
>    * @param a3 the third argument to pass to the function to invoke
>    * @param a4 the fourth argument to pass to the function to invoke
>    * @param a5 the fifth argument to pass to the function to invoke
>    * @param a6 the fifth argument to pass to the function to invoke
>    */
```

```
>   template <typename U1, typename U2, typename U3, typename U4, typename U5, typename U6,
>             typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
>   static EventId ScheduleDestroy (void (*f)(U1,U2,U3,U4,U5,U6), T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6);
>
>   /**
861a991,998
> template <typename MEM, typename OBJ,
>           typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
> EventId Simulator::Schedule (Time const &time, MEM mem_ptr, OBJ obj,
>                              T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6)
> {
>   return DoSchedule (time, MakeEvent (mem_ptr, obj, a1, a2, a3, a4, a5, a6));
> }
>
895a1033,1038
> template <typename U1, typename U2, typename U3, typename U4, typename U5, typename U6,
>           typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
> EventId Simulator::Schedule (Time const &time, void (*f)(U1,U2,U3,U4,U5,U6), T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6)
> {
>   return DoSchedule (time, MakeEvent (f, a1, a2, a3, a4, a5, a6));
> }
941a1085,1092
> template <typename MEM, typename OBJ,
>           typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
> void Simulator::ScheduleWithContext (uint32_t context, Time const &time, MEM mem_ptr, OBJ obj,
>                                      T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6)
> {
>   return ScheduleWithContext (context, time, MakeEvent (mem_ptr, obj, a1, a2, a3, a4, a5, a6));
> }
>
976c1127,1132
<
---
> template <typename U1, typename U2, typename U3, typename U4, typename U5, typename U6,
>           typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
> void Simulator::ScheduleWithContext (uint32_t context, Time const &time, void (*f)(U1,U2,U3,U4,U5,U6), T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6)
> {
>   return ScheduleWithContext (context, time, MakeEvent (f, a1, a2, a3, a4, a5, a6));
> }
1027a1184,1192
> template <typename MEM, typename OBJ,
>           typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
> EventId
> Simulator::ScheduleNow (MEM mem_ptr, OBJ obj,
>                         T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6)
> {
>   return DoScheduleNow (MakeEvent (mem_ptr, obj, a1, a2, a3, a4, a5, a6));
> }
>
1067a1233,1239
> template <typename U1, typename U2, typename U3, typename U4, typename U5, typename U6,
>           typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
> EventId
> Simulator::ScheduleNow (void (*f)(U1,U2,U3,U4,U5,U6), T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6)
> {
>   return DoScheduleNow (MakeEvent (f, a1, a2, a3, a4, a5, a6));
> }
1118a1291,1299
> template <typename MEM, typename OBJ,
>           typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
> EventId
> Simulator::ScheduleDestroy (MEM mem_ptr, OBJ obj,
>                             T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6)
> {
>   return DoScheduleDestroy (MakeEvent (mem_ptr, obj, a1, a2, a3, a4, a5, a6));
> }
>
1156a1338,1345
> }
>
> template <typename U1, typename U2, typename U3, typename U4, typename U5, typename U6,
>           typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
> EventId
> Simulator::ScheduleDestroy (void (*f)(U1,U2,U3,U4,U5,U6), T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6)
> {
>   return DoScheduleDestroy (MakeEvent (f, a1, a2, a3, a4, a5, a6));
```