DISSERTAÇÃO DE MESTRADO

Acesso Inicial Omnidirecional de Usuário em Redes de Ondas Milimétricas via Esquema de Alamouti

Thayane Rodrigues Viana

Brasília, agosto de 2017

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

DISSERTAÇÃO DE MESTRADO

Acesso Inicial Omnidirecional de Usuário em Redes de Ondas Milimétricas via Esquema de Alamouti

Thayane Rodrigues Viana

Dissertação de Mestrado 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, Ph.D, ENE/UnB _____
Orientador

Prof. Renato Mariz de Moraes, Ph.D, ENE/UnB _____
Examinador interno

Porf. Jacir Luiz Bordim, Ph.D, CIC/UnB _____
Examinador externo

FICHA CATALOGRÁFICA

VIANA, THAYANE RODRIGUES
Acesso Inicial Omnidirecional de Usuário em Redes de Ondas Milimétricas via Esquema de Alamouti [Distrito Federal] 2017.
xvi, 110 p., 210 x 297 mm (ENE/FT/UnB, Mestre, Engenharia Elétrica, 2017).
Dissertação de Mestrado - Universidade de Brasília, Faculdade de Tecnologia.
Departamento de Engenharia Elétrica

1. Redes mmWave	2. Acesso Inicial
3. Acesso Aleatório	4. Alamouti
I. ENE/FT/UnB	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

VIANA, T.R. (2017). *Acesso Inicial Omnidirecional de Usuário em Redes de Ondas Milimétricas via Esquema de Alamouti*. Dissertação de Mestrado, Publicação PGEA.DM-674/2017, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 110 p.

Thayane Rodrigues Viana
Depto. de Engenharia Elétrica (ENE) - FT
Universidade de Brasília (UnB)
Campus Darcy Ribeiro
CEP 70919-970 - Brasília - DF - Brasil

Dedicatória

Dedico esta dissertação de mestrado a todos os estudantes e pesquisadores que assim como eu, são fascinados por comunicações móveis e desejam aprender ainda mais.

Thayane Rodrigues Viana

Agradecimentos

Agradeço a Deus pela oportunidade de concluir o meu mestrado na Universidade de Brasília. Como disse Samuel em 1 Samuel 7:12, "até aqui nos ajudou o Senhor". Sem Deus, eu jamais teria conseguido terminar, Ele é a minha força e o meu refúgio, o Senhor foi muito gracioso comigo ao me capacitar para fazer esse mestrado. Reconheço que toda boa dádiva vem de Deus, e que Ele me ajudou e me fortaleceu para que eu não desistisse nos momentos mais difíceis, me dando ânimo e me sustentando até o final. A Deus seja toda glória! "Porque dele e por ele, e para ele, são todas as coisas; glória, pois, a ele eternamente. Amém." (Romanos 11:36). Agradeço muito também ao meu orientador, o professor Marcelo Carvalho, por toda paciência e dedicação ao me orientar no mestrado. Sou muito grata por tudo que tenho aprendido com o professor Marcelo, e, sem dúvida, ele sempre foi uma grande inspiração para mim. Agradeço também à minha família por todo o apoio, palavras de incentivo e orações. Agradeço especialmente à Nayara Viana por ter me ajudado com as figuras da dissertação. Agradeço também aos meus amigos e irmãos em Cristo do PG4U e do Ministério Resgate, que sempre se preocuparam comigo, me dando apoio e orando por mim e pelo meu mestrado. Finalmente, agradeço à Universidade de Brasília por ter me proporcionado um tempo tão produtivo de estudos. Foi um grande privilégio concluir a graduação e a pós-graduação em uma universidade pública de qualidade. Guardarei sempre uma grata memória do meu período de mestrado, foi um tempo muito bom de aprendizado.

Thayane Rodrigues Viana

RESUMO

O crescente volume de tráfego de dados e a demanda por altas taxas nas atuais redes sem fio afirmam a necessidade de novas tecnologias para melhorar a eficiência espectral na próxima geração de redes celulares. Entre essas tecnologias, as comunicações por ondas milimétricas (mmWave, do inglês *Millimeter Wave*) emergem como uma candidata chave aproveitando bandas de frequência abundantes e inexploradas, juntamente com a densificação de células menores e o grande número de antenas nas duas extremidades do enlace. Entretanto, a alta perda de propagação e o bloqueio do sinal são desafios importantes a serem superados nas bandas mmWave. Para compensar isso, as técnicas de conformação de feixe foram propostas para alcançar altos ganhos de diretividade. Infelizmente, o uso de conformação de feixe no acesso inicial (quando o equipamento do usuário (UE, do inglês *User Equipment*) precisa se conectar à estação base (BS, do inglês *Base Station*)) deve introduzir uma sobrecarga de atraso significativa, uma vez que exigirá algoritmos de busca de feixes ou estimativa de canal fora da banda para proporcionar a configuração e o alinhamento dos feixes. Para evitar esse problema, este trabalho investiga o uso do Esquema de Alamouti como meio de realizar a conexão do dispositivo de usuário dentro da banda mmWave de forma omnidirecional aproveitando seus ganhos de diversidade. Os resultados das simulações para uma célula *outdoor* em 28 GHz são apresentados para estimação da probabilidade de conexão do UE e cálculo da razão sinal-ruído (SNR, do inglês *signal-to-noise ratio*) média do canal *uplink* como função da distância entre BS e UE. Os resultados das simulações indicam que, com apropriados parâmetros selecionados, o Esquema de Alamouti pode prover probabilidades de conexão do UE similares às probabilidades proporcionadas pelo uso de conformação de feixe, sem provocar a sobrecarga de atraso causada pelos algoritmos de busca de feixe ou soluções fora da banda.

ABSTRACT

The ever-increasing volume of data traffic and demand for higher data rates in current wireless networks have posed the need for new technologies to improve spectral efficiency in next-generation cellular networks. Among such technologies, millimeter wave (mmWave) communications has surfaced as a key candidate by taking advantage of abundant and unexploited frequency bands, coupled with dense small cell deployments and a large number of antenna elements at both ends of the link. However, severe path loss and signal blockage are important challenges to be overcome at mmWave bands. To compensate for that, beamforming techniques have been proposed to achieve high directional gains. Unfortunately, the use of beamforming in the initial access (when the user equipment (UE) needs to connect to the base station (BS)) may incur sig-

nificant delay overhead, since it will require beam searching algorithms or out-of-band channel estimation for proper beam set up and alignment. To avoid this problem, this work investigates the use of the Alamouti scheme as a means to achieve in-band omnidirectional UE connection by leveraging its transmission diversity gains. Simulation results for an outdoor cell at 28 GHz are presented for estimation of the UE connection probability and uplink average SNR as a function of the UE-BS distance. The simulation results indicate that, with appropriate chosen parameters, the Alamouti scheme can deliver UE connection probabilities close to beamforming, and without incurring the delay overhead caused by beam searching algorithms or out-of-band solutions.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

Siglas

3GPP	*3rd Generation Partnership Project*
4G	*4rd Generation*
5G	*5rd Generation*
ADC	*Analog Digital Converter*
AoA	*Angle of Arrrival*
AoD	*Angle of Departure*
AWGN	*Additive White Gaussian Noise*
BLER	*Block Error Rate*
BS	*Base Station*
CB	*Code Block*
CFR	*Code of Federal Regulations*
CoMP	*Cooperative Multipoint*
CSMA/CA	*Carrier Sense Multiple Access with Collision Avoidance*
D2D	*Device-to-device*
EHF	*Extremely High Frequence*
FCC	*Federal Comunications Commission*
GPS	*Global Positioning System*
HARQ	*Hybrid Automatic Retransmission Request*
IEEE	*Institute of Electrical and Electronics Engineers*
ITU	*International Telecommunication Union*
LMDS	*Local Multipoint Distribution Systems*
LOS	*Line-of-sight*
LSM	*Link-to-System Mapping*
LTE	*Long Term Evolution*
MAC	*Medium Access Control*
MCS	*Modulation and Coding Scheme*
MIESM	*Mutual Information Based Effective SINR*
MIMO	*Multiple Input Multiple Output*
MISO	*Multiple Input Single Output*
MMIB	*Mean Mutual Information per Coded Bit*
mmWave	*Millimeter Wave*
NLOS	*Non-light-of-sight*
PHY-CC	*Physical Control Channel*
PLE	*Path Loss Exponent*
QoS	*Quality of Service*

RA	*Random Access*
RACH	*Random Access Channel*
RAR	*Random Access Response*
RF	*Radio Frequence*
RLF	*Radio Link Failure*
RNTI	*Radio Network Temporary Identifier*
SHF	*Super High Frequence*
SINR	*Signal Interference Noise Ratio*
SISO	*Single Input Single Output*
SNR	*Signal Noise Ratio*
STBC	*Space Time Code Block*
TB	*Transport Block*
TDD	*Time Division Duplex*
TDMA	*Time Division Multiple Access*
TTI	*Transmission Time Interval*
UE	*User Equipment*
UHF	*Ultra High Frequency*
WN	*Wireless Node*
WPAN	*Wireless Personal Area Network*

Símbolos

h_i	ganho complexo do canal entre a i-ésima antena transmissora e a antena receptora
n_i	amostra do ruído AWGN no receptor, associada com a transmissão do i-ésimo símbolo
s_i	símbolo transmitido a partir da antena i
E_s	energia total transmitida
E_n	energia do ruído AWGN
y_i	símbolo recebido no período de símbolo i
$\|\boldsymbol{H}\|_F^2$	norma de Frobenius
$\tilde{\mathbf{n}}$	vetor de ruído complexo Gaussiano
M	quantidade de antenas transmissoras
N	quantidade de antenas receptoras
K	quantidade de grupos de caminhos de propagação do sinal
L	quantidade de sub-caminhos de propagação do sinal dentro de cada grupo
$\boldsymbol{H}(t, f)$	matriz de canal entre o transmissor e o receptor
$g_{kl}(t, f)$	ganho complexo de desvanecimento em pequena escala
$\boldsymbol{u}_{rx}(.) \in \mathbb{C}^N$	vetor de função de resposta para as antenas RX
$\boldsymbol{u}_{tx}(.) \in \mathbb{C}^M$	vetor de função de resposta para as antenas TX
θ_{kl}^{rx}	ângulo de chegada (AoAs) horizontal
ϕ_{kl}^{rx}	ângulo de chegada (AoAs) vertical
θ_{kl}^{tx}	ângulo de partida (AoDs) horizontal
ϕ_{kl}^{tx}	ângulo de partida (AoDs) vertical
P_{kl}	espalhamento de potência do sub-caminho l do grupo k
f_{dmax}	deslocamento Doppler máximo
w_{kl}	ângulo de chegada do sub-caminho l do grupo k relativo à direção do movimento
τ_{kl}	espalhamento do atraso do sub-caminho l do grupo k
f	frequência da onda portadora do sinal
$P_{out}(d)$	probabilidade de falha no enlace
$P_{LOS}(d)$	probabilidade do enlace ser LOS
$P_{NLOS}(d)$	probabilidade do enlace ser NLOS
d	distância entre o transmissor e o receptor
P_{REF}	variável aleatória de referência
$PL(d)$	perda de propagação (expressa em dB)

α e β	parâmetros estimados obtidos a partir do ajuste linear das perdas de propagação obtidas como função da distância BS-UE em medições realizadas
$\xi \sim N(0, \sigma^2)$	variável aleatória que considera os efeitos do sombreamento
σ^2	variância log-normal do sombreamento
$C_{BLER,i}(\gamma_i)$	probabilidade de erro de bloco (BLER) de cada CB
$b_{C_{SIZE},MCS}$	média da Função de Distribuição Acumulativa Gaussiana
$c_{C_{SIZE},MCS}$	desvio padrão da Função de Distribuição Acumulativa Gaussiana
γ_i	informação mútua média por bit codificado do bloco de código i
R	quantidade de blocos de recurso (RB) alocados para o usuário
r	índice do RB alocado
m	esquema de modulação adotado
SNR_r	valor da SNR instantânea associada ao RB de índice r
$I_m(.)$	função de informação mútua associada ao esquema de modulação m
T_{BLER}	taxa de erro de bloco de transporte
$G(t,f)_{BFi,j}$	ganho de conformação de feixe do transmissor i para o receptor j
$\boldsymbol{H}(t,f)_{i,j}$	matriz de canal $N \times M$ do ij-ésimo enlace
$\boldsymbol{w}_{tx_{i,j}}$	vetor de conformação de feixe $M \times 1$ do transmissor i, quando este está transmitindo para o receptor j
$\boldsymbol{w}_{rx_{i,j}}$	vetor de conformação de feixe $N \times 1$ do receptor j, quando este está recebendo do transmissor i.
$SNR_{i,j}$	relação sinal-ruído
$P_{tx_{i,j}}$	potência transmitida
$PL_{i,j}$	perda de propagação entre a BS_i e o UE_j
BW	largura de banda
N_0	densidade espectral de potência do ruído

1 INTRODUÇÃO

O crescimento do número de dispositivos móveis e das aplicações multimídia de alta velocidade tem aumentado grandemente o volume de tráfego de dados em redes celulares nos últimos anos. A demanda por altas taxas de dados em sistemas de comunicação sem fio motiva os pesquisadores a investigarem tecnologias avançadas para melhorar a eficiência espectral na futura quinta geração (5G) de redes celulares. As futuras redes celulares 5G vêm com mudanças fundamentais para prover altas taxas de dados para cada dispositivo móvel suportar aplicações multimídia de alta velocidade com rigorosos requisitos de Qualidade de Serviço (QoS). É esperado que as redes celulares 5G apresentem melhorias significativas em relação às redes celulares 4G referente à taxa de dados, latência, eficiência energética e custo [2]. As taxas de pico devem chegar até dezenas de Gb/s, a latência deve ser de aproximadamente 1 ms e o consumo de energia e custo por enlace devem ter uma redução significativa.

Dentre as tecnologias consideradas para as redes celulares 5G, a rede de ondas milimétricas (mmWave, do inglês *millimeter wave*) é considerada como uma solução bastante promissora [3]. De fato, é discutido que as redes mmWave possam alcançar taxas de dados da ordem de múltiplos Gb/s [3]. As redes mmWave aproveitam a vantagem de grandes faixas de frequência inexploradas (de 6 GHz até 300 GHz [4]), caracterizadas pelo pequeno comprimento de onda e alta frequência. Os pequenos comprimentos de onda permitem a utilização de um grande número de pequenas antenas na estação base e no equipamento do usuário. Com isso, é possível utilizar a conformação de feixe para realizar transmissões altamente direcionais, compensando a forte atenuação e a perda de propagação da banda mmWave com os altos ganhos de diretividade proporcionados pela conformação de feixe (5).

Muitos trabalhos sobre redes mmWave têm o foco na camada física e na operação de conformação de feixe [6, 7, 8, 1, 9]. Mas, poucos trabalhos apresentam soluções para os desafios de camada MAC (do inglês *Medium Access Control*) em redes mmWave, principalmente no que se refere ao acesso inicial do dispositivo do usuário na rede e ao impacto das transmissões direcionais no canal de controle. O acesso inicial dos usuários é uma etapa fundamental para que seja estabelecido um enlace físico de comunicação entre a estação base e o equipamento do usuário. Só após o estabelecimento dessa conexão é que o usuário e a estação base podem trocar mensagens no canal de dados. Portanto, é importante discutir e apresentar soluções para o problema do acesso inicial em redes mmWave.

Quando consideramos uma transmissão direcional no canal de controle de redes celulares mmWave, é preciso levar em conta que os procedimentos de busca de feixe podem incluir um atraso adicional na etapa de acesso aleatório. Os trabalhos apresentados em [10, 11, 8] abordam a sobrecarga de tempo requerida pelo procedimento de alinhamento de feixes. Esse atraso adicional para realizar o alinhamento de feixes é prejudicial principalmente para a fase de acesso aleatório do acesso inicial de um dispositivo móvel na rede, quando o dispositivo móvel tenta se conectar

na rede reservando seus recursos de transmissão para conseguir transmitir os pacotes de dados. A longa duração do acesso aleatório pode impactar atributos cruciais das redes mmWave, como nos aspectos de acesso inicial, falha de enlace de rádio, *handover*, configuração dos canais *uplink* e *downlink* utilizando o TDD (do inglês *time-division duplex*) e agendamento de feixes [12].

O atraso adicional requerido pelo alinhamento dos feixes no acesso aleatório faz com que o dispositivo móvel permaneça mais tempo na transição do estado ocioso para o estado conectado, consumindo mais energia do dispositivo móvel. Assim, essa duração prolongada da fase de acesso aleatório devido ao alinhamento de feixes impacta na eficiência energética dos dispositivos mmWave [12]. A longa duração do acesso aleatório também impacta no reestabelecimento da conexão do usuário com a estação base em caso de RLF (do inglês *Radio Link Failure*), afetando a qualidade da experiência do usuário. Outro aspecto importante a ser considerado é o procedimento de *handover* do dispositivo móvel de uma célula para outra. Devido à configuração de um grande número de pequenas células em redes mmWave, um dispositivo móvel pode mudar de célula frequentemente. Se esses procedimentos de *handover* requererem uma longa duração, a qualidade de serviço do usuário será prejudicada, especialmente em serviços de tempo real.

Alguns trabalhos abordam o atraso no acesso inicial requerido por transmissões direcionais em redes mmWave [13, 14, 15], demandado pelo uso de técnicas de alinhamento de feixes. Em [13] são analisadas duas técnicas de busca de feixes para o acesso inicial dos dispositivos à rede: a busca exaustiva e a busca iterativa. O atraso total obtido nas simulações para a busca exaustiva chega a 360 ms considerando uma probabilidade de não-detecção menor do que 0,01 para usuários de borda. No estudo realizado em [15], o desempenho dos procedimentos de acesso inicial foram avaliados em termos do atraso de descoberta, que é o tempo requerido pela BS e UE para determinar as melhores direções e alinhar os seus feixes. Já Barati *et al.* [14] também considera o envio do preâmbulo de acesso aleatório em modo omnidirecional, além do modo direcional. De acordo com a avaliação de desempenho realizada, no geral, o menor atraso na etapa de sincronização foi obtido com uma transmissão omnidirecional do sinal de sincronização.

Considerando o problema da longa duração do acesso inicial em transmissões direcionais, nosso trabalho faz uma avaliação do desempenho do canal de controle físico em modo omnidirecional com o esquema de diversidade de transmissão de Alamouti [16]. Um canal de controle físico omnidirecional não requer o atraso adicional de alinhamento de feixes demandado pelo canal de controle físico direcional, mas tem a limitação do alcance em distância, devido à forte atenuação e alta perda de propagação da banda mmWave. Por isso, nós propomos utilizar o Esquema de Alamouti para prover ganhos de diversidade de transmissão e aumentar o alcance em distância da transmissão omnidirecional no acesso aleatório. Com o Esquema de Alamouti é possível aproveitar o grande número de antenas no transmissor e receptor para prover altos ganhos de diversidade, requerendo um processamento linear simples e sem necessidade do conhecimento do canal pelo transmissor [16].

Neste trabalho avaliamos o desempenho do canal de controle físico omnidirecional com o Esquema de Alamouti na banda mmWave em termos da probabilidade de conexão do dispositivo

móvel na rede durante o acesso inicial em função da distância entre a estação base e o usuário. Também avaliamos a SNR recebida no canal de controle *uplink* em função da distância entre a estação base e o dispositivo do usuário. Assim, analisamos a viabilidade de utilizar um canal de controle físico omnidirecional com o Esquema de Alamouti na fase de acesso inicial do usuário a fim de realizar a conexão do usuário na rede sem a sobrecarga de tempo demandada pelo procedimento de alinhamento de feixes em um canal de controle físico direcional.

1.1 OBJETIVO

Este trabalho tem o objetivo de apresentar uma avaliação de desempenho de um canal de controle omnidirecional com o uso do Esquema de Alamouti para realizar o acesso inicial de um dispositivo móvel em uma rede celular mmWave. Apresentamos a proposta de utilizar o Esquema de Alamouti para aumentar o alcance em distância da transmissão omnidirecional durante o acesso inicial, tomando vantagem do ganho de diversidade de transmissão proporcionada pelo Esquema de Alamouti através das múltiplas antenas no transmissor e no receptor. Apresentamos a opção de realizar o canal de controle físico em modo omnidirecional para evitar a sobrecarga de tempo demandada por uma transmissão direcional, devido ao atraso adicional para fazer o alinhamento dos feixes.

Avaliamos o desempenho do canal de controle omnidirecional com o Esquema de Alamouti em relação a um canal de controle omnidirecional SISO (do inglês *Single Input Single Output*),sem ganho de diversidade, e em relação a um canal de controle físico direcional com conformação de feixe. Com essa comparação, temos o objetivo de analisar a viabilidade de utilizar um canal de controle omnidirecional com o Esquema de Alamouti para realizar o acesso aleatório em redes mmWave.

1.2 CONTRIBUIÇÕES

Nosso trabalho contribui para a pesquisa em redes celulares mmWave, principalmente no que se refere ao acesso inicial do dispositivo do usuário na rede durante a fase de acesso aleatório. Propomos uma alternativa para realizar o acesso inicial em redes mmWave sem o atraso adicional exigido pela busca de feixes, utilizando o esquema de diversidade de transmissão de Alamouti para realizar uma transmissão omnidirecional no acesso aleatório a fim de conectar o usuário na rede. Entre as principais contribuições deste trabalho estão:

- Implementação do esquema Esquema de Alamouti no módulo desenvolvido para redes mmWave no simulador de redes ns-3, através do ganho de diversidade no cálculo da SNR.
- Inclusão da opção de realizar o canal de controle do acesso inicial de forma omnidirecional com o Esquema de Alamouti no módulo mmWave do ns-3.

- Implementação do modelo de erro para o canal de controle no módulo mmWave do ns-3, permitindo que haja erros na transmissão de quadros de controle (já que o modelo de erro só estava implementado para canal de dados, tratando o canal de controle como ideal, assumindo uma recepção perfeita de todos os quadros de controle).
- Avaliação de desempenho do canal de controle físico omnidirecional com o Esquema de Alamouti através de gráficos que reportam a probabilidade de conexão do dispositivo do usuário na rede e a SNR recebida no canal de controle *uplink* para o caso omnidirecional SISO, para diferentes configurações do Esquema de Alamouti e para o canal de controle direcional com conformação de feixe.

1.3 ESTRUTURA DA DISSERTAÇÃO

Esta dissertação está dividida em alguns capítulos para facilitar a leitura e compreensão do trabalho desenvolvido. O Capítulo 2 apresenta os principais fundamentos teóricos em que este trabalho foi baseado, apresentando importantes conceitos relacionados ao acesso inicial em redes mmWave. O Capítulo 3 traz uma revisão bibliográfica dos principais trabalhos relacionados com o foco dessa dissertação. O Capítulo 4 apresenta a nossa proposta de realizar a conexão do usuário no acesso inicial de redes mmWave utilizando o Esquema de Alamouti. O Capítulo 5 descreve o sistema e modelos utilizados para implementação da nossa proposta no simulador de redes ns-3. O Capítulo 6 apresenta os cenários de simulação utilizados e os resultados obtidos com as simulações, incluindo a avaliação do desempenho da nossa proposta. Finalmente, o Capítulo 7 traz nossas principais conclusões sobre o nosso trabalho e sugestões de trabalhos futuros a serem desenvolvidos a partir das nossas conclusões.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os principais fundamentos teóricos nos quais esse trabalho foi embasado. Assim, os conceitos mais estudados e necessários para a compreensão da proposta são expostos de forma a trazer um conhecimento básico teórico para o entendimento da solução.

2.1 QUINTA GERAÇÃO DE REDES CELULARES - 5G

Esperam-se que algumas mudanças fundamentais sejam implementadas nas futuras redes celulares 5G. Essas mudanças devem ocorrer devido aos requerimentos de desempenho em termos de taxa de transferência de dados, latência e consumo de energia [2]. A taxa de transferência de dados desempenha o papel mais importante na arquitetura de redes celulares 5G, já que a explosão do tráfego móvel é um dos principais motivadores para o desenvolvimento da tecnologia 5G. A taxa de transferência de dados agregada precisa ser 1000 vezes maior que a taxa permitida na tecnologia 4G [2]. A taxa de transferência de dados na borda da célula, que costuma ser a pior taxa de dados que um usuário experimenta na rede, deve estar na faixa de 100 Mbps a 1 Gbps nas redes celulares 5G, diferente da taxa na borda da célula na tecnologia LTE, que é de aproximadamente 1 Mbps. Já a taxa de pico, que é a melhor taxa que o usuário experimenta na rede, deve ser na ordem de dezenas de Gbps.

Na tecnologia 4G a latência de ida e de volta é da ordem de 15 ms. Para a tecnologia 5G, a latência deve ser da ordem de 1 ms, a fim de suportar as aplicações emergentes. Também é esperado que o consumo de energia diminua nas redes celulares 5G. Devido ao aumento significativo da taxa de transferência de dados, é necessário que o consumo de energia por bit diminua em cerca de 99%. Algumas tecnologias promissoras como as redes de ondas milimétricas e as pequenas células proporcionam um custo razoável e redução do consumo de energia. Além dessas, outras tecnologias têm sido promissoras para implementar as redes celulares 5G. As principais tecnologias consideradas para as redes celulares 5G são [17]:

- Arquitetura centrada nos dispositivos: O aumento das redes heterogêneas, a coexistência de bandas de frequência com características de propagação diferentes, a utilização de comunicações com CoMP (do inglês *Cooperative Multipoint*) e a comunicação D2D (do inglês *device-to-device*) têm mostrado a necessidade de mudança de uma arquitetura centrada em célula para uma arquitetura centrada nos dispositivos, alterando os conceitos de *downlink* e *uplink* e também de plano de controle e plano de dados.
- Sistemas celulares *Millimeter Wave (mmWave)*: A banda de frequência de ondas milimétricas (mmWave) tem sido promissora para as redes celulares 5G por tomar vantagem da larga e inexplorada faixa de frequência de 30 até 300 GHz. A utilização de um grande número de

pequenas antenas e os ganhos de transmissão direcional proporcionam altas taxas de dados para redes mmWave.

- *Massive* MIMO: O *Massive* MIMO (do inglês *Massive Multiple-Input Multiple-Output*) utiliza uma grande número de antenas no transmissor e no receptor, sendo que o número de antenas na estação base deve ser bem maior que o número de dispositivos atendidos. As mensagens são multiplexadas em recursos de tempo-frequência e é possível alcançar altos ganhos na eficiência espectral.

- Dispositivos Inteligentes: Algumas tecnologias como D2D (do inglês *Device-to-device*), cache local e rejeição de interferência devem ser incorporadas nos dispositivos móveis, permitindo que eles tenham um papel mais ativo nas redes celulares 5G aumentando a inteligência dos dispositivos.

- Suporte à comunicação máquina-a-máquina: Alguns serviços emergentes requerem um grande número de dispositivos conectados, tal como sensores, *smart grids* e comunicação veicular. Esses serviços também requerem enlaces de alta confiabilidade, baixa latência e operação em tempo real.

As redes 5G devem contar com mais estações base, combinando macro células com células menores. Várias tecnologias de rádio devem operar em conjunto formando uma rede heterogênea, integrando tecnologias como o LTE, o Wi-Fi e comunicações mmWave ou outras tecnologias do 5G [15]. A técnica do *Massive* MIMO e as pequenas células são duas abordagens promissoras para as redes celulares 5G [18]. O pequeno comprimento de onda da banda mmWave permite que centenas de elementos de antena sejam colocados em uma matriz em uma pequena plataforma na estação base. O *Massive* MIMO aproveita os benefício do MIMO em larga escala, aumentando a eficiência espectral e a capacidade do sistema, podendo ser implantado com componentes de baixo custo e baixo consumo de energia [19]. As pequenas células permitem que a distância média entre o transmissor e o receptor diminua, resultando em uma perda de propagação menor, altas taxas de dados e eficiência energética [18].

2.2 REDES CELULARES MMWAVE

Devido à sobrecarga do espectro do LTE (do inglês *Long Term Evolution*) abaixo de 6 GHz, tem surgido um grande interesse em bandas mmWave, onde há uma grande quantidade de espectro inutilizado disponível. As redes mmWave são caracterizadas por seus pequenos comprimentos de onda, sua alta frequência e sua grande largura de banda. Sistemas mmWave aproveitam a vantagem de grande bandas de frequências inexploradas (de 6 GHz até 300 GHz [4]), e seus pequenos comprimentos de onda (de 1 a 100mm) permitem o desenvolvimento de um grande número de antenas do tamanho de chips de rádio. O espectro de 3 a 30 GHz é chamado de banda SHF (do inglês *Super High Frequency*) e o espectro de 30 GHz a 300 GHz é chamado

de banda EHF (do inglês, *Extremely High Frequency*). Essas bandas possuem características de propagação semelhantes e por isso são incluídas na banda de ondas milimétricas [3]. Essa faixa do espectro não tem sido muito explorada para aplicações comerciais, mas recentemente essa porção do espectro tem sido estudada para permitir comunicações de curto alcance com altas taxas de dados.

O espectro não licenciado mmWave proporciona as seguintes vantagens [2]:

- Grande alocação de frequência: O espectro mmWave está disponível na maioria das regiões do mundo.
- Espectro limpo: Como é uma faixa de frequência não utilizada pelas operadoras tanto no ambiente *indoor* quanto no *outdoor*, há menos chance de interferência.
- A alta frequência permite a utilização de pequenas antenas com alto ganho e amplificadores RF de baixa potência.
- A alta perda de propagação permite a sobreposição de redes que não interferem muito umas nas outras. As antenas altamente direcionais nas frequências mmWave facilitam a reutilização espacial.

Apesar dessas vantagens, a alta perda de percurso e o bloqueio de sinal são importantes desafios a serem vencidos em bandas mmWave. Uma das características das ondas milimétricas é a alta absorção por oxigênio e vapor de água [3]. Outra característica é a alta perda de penetração em materiais sólidos como objetos, paredes, vidros e corpos humanos [4]. Devido à essa alta vulnerabilidade a obstáculos, as redes de ondas milimétricas estão sujeitas ao bloqueio, sofrendo com a grande atenuação através dos obstáculos. A Figura 2.1 retrata uma rede mmWave sujeita ao bloqueio e reflexão do sinal, com um obstáculo entre a estação base BS1 e o dispositivo móvel UE1. Essas perdas não são compensadas com o aumento da potência do transmissor, elas precisam ser compensadas com transmissões direcionais sobre canais sem bloqueio. Para vencer o bloqueio, é necessário um busca por um canal espacial que não está bloqueado, exigindo a sobrecarga de um novo procedimento de conformação de feixe. Isso torna necessário um novo conceito de definição da célula, pois requer células menores e dinâmicas, centralizadas no dispositivo do usuário ao invés da estação base.

As transmissões direcionais podem provocar o fenômeno da "surdez", que se refere à situação em que os feixes de transmissão e recepção não se encontram alinhados [4], dificultando o estabelecimento de um enlace de comunicação entre o transmissor e o receptor. A surdez impacta principalmente no acesso inicial do usuário na rede, quando o usuário e a estação base precisam estar com os feixes de transmissão e recepção alinhados para começar a trocar mensagens de controle que permitem que o usuário seja conectado na estação base. Por outro lado, a surdez reduz a interferência significativamente, já que o receptor só escuta em uma direção específica do canal espacial. A redução da interferência permite o desenvolvimento de células menores e um maior fator de reuso de frequência.

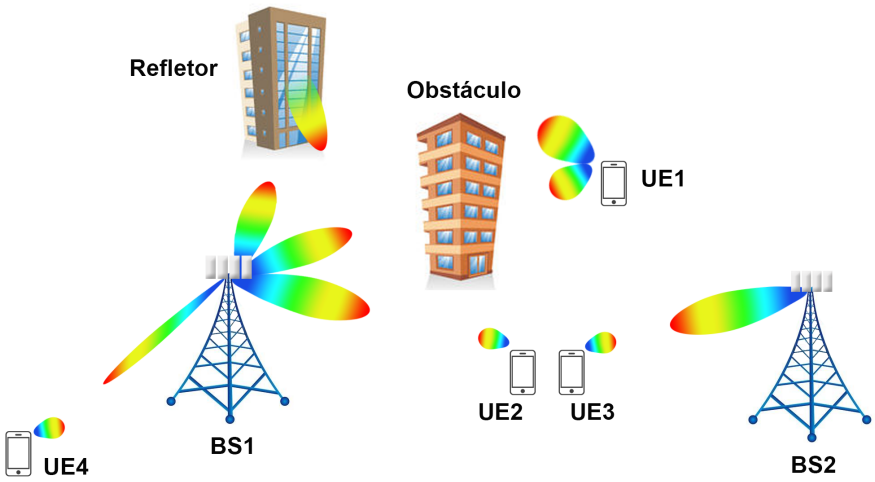

Figura 2.1: Rede mmWave com transmissão direcional a partir da BS e do UE com bloqueio do sinal devido aos obstáculos presentes entre o transmissor e receptor.

Espera-se que os dispositivos mmWave operem tanto em redes mmWave como em redes de microondas, na banda do LTE por exemplo. A banda de microondas pode ser explorada para a troca de mensagens de controle, que exige taxas de dados menores mas com maior confiabilidade e estabilidade do enlace. Dessa forma a troca de mensagens de controle pode ser feita de forma omnidirecional na faixa de frequência do LTE. É esperado também que uma variedade de tamanhos diferentes de célula trabalhem juntas nas redes celulares 5G [4], como macro-células, pico-células e femto-células. Isso facilita a transição das redes de microondas para as redes mmWave, pois permite a coexistência de duas faixas de frequência em operação. É possível perceber então que as redes celulares mmWave podem possuir heterogeneidade tanto de espectro quanto de cenário de desenvolvimento. Mas, apesar da possibilidade de utilizar a banda de microondas no canal de controle, essa abordagem traz limitações como a incompatibilidade dos parâmetros de canal devido às diferenças nas características de propagação das duas bandas [20] e os esforços exaustivos em medições multi-bandas [21].

Em [22] algumas características de propagação foram resumidas em termos do expoente de perda de propagação (PLE, do inglês *path loss exponent*) sobre canais com linha de visada (LOS, do inglês *line-of-sight*) e sem linha de visada (NLOS, do inglês *non-line-of-sight*), com raio de atenuação em 200 m e absorção de oxigênio em 200 m. É mostrado que para distância de 200 m entre o transmissor e o receptor, as bandas de 28 GHz e 38 GHz sofrem baixa atenuação causada pela chuva e baixa absorção de oxigênio, enquanto que as bandas de 60 GHz e 73 GHz já sofrem uma alta atenuação causada pela chuva e alta absorção de oxigênio também. É possível observar também que as transmissões NLOS têm uma perda de propagação maior do que as transmissões LOS, para as quatro bandas.

As bandas de 28 e 38 GHz estão disponíveis com alocação de espectro de 1 GHz de largura de banda [18]. Rappaport *et al.* [18] realizaram uma série de medições de propagação nas bandas de 28 GHz e 38 GHz. Os estudos realizados nessas bandas concluíram que os sinais nessas frequências podem ser detectados para distâncias de até 200 m a partir da estação base, mesmo

em uma conexão NLOS. Considerando esse tamanho de célula em ambientes urbanos, a absorção atmosférica não traz uma perda de propagação adicional significante nas bandas de 28 e 38 GHz, cerca de somente 1,4 dB de atenuação na distância de 200 m. Para pequenas distâncias (menos que 1 km), a atenuação da chuva apresenta um efeito mínimo na propagação de ondas milimétricas em 28 e 38 GHz para pequenas células [18].

A faixa de frequência de ondas milimétricas tem sido considerada para as redes celulares 5G, mas até o momento de elaboração deste trabalho, ainda não existe um padrão definido com a especificação do uso de redes mmWave para redes celulares 5G. Mas além da quinta geração de redes celulares, existem outras iniciativas que também utilizam a banda mmWave. Essa banda já tem sido adotada por exemplo no padrão IEEE 802.11ad e no padrão IEEE 802.15.3c. A alteração IEEE 802.11ad do padrão IEEE 801.11 define um esquema de comunicação direcional que aproveita a técnica de conformação de feixe para compensar a forte atenuação na banda de 60 GHz. Esse padrão utiliza setores "virtuais" das antenas, que são implementados usando vetores de peso pré-calculados para uma matriz de antenas ou utilizando um sistema com múltiplas antenas direcionais [23]. O padrão IEEE 802.15.3c foi desenvolvido para tratar de redes WPAN (do inglês *Wireless Personal Area Network*) baseadas em ondas milimétricas. Essa rede WPAN mmWave opera na banda não licenciada de 57-66 GHz, definida pela FCC 47 CFR 15.255. Esse padrão especifica as camadas física e MAC para redes WPAN *indoor*, que são compostas de vários nós sem fio (WNs) e um único controlador de pico-redes (PNC) que provê a sincronização da rede e coordena a transmissão na pico-rede [22].

2.3 CONFORMAÇÃO DE FEIXE

A conformação de feixe é uma técnica de processamento de sinais em que as antenas são adaptadas para formarem um padrão de feixe centralizado e direcional. A conformação de feixe pode ser utilizada no transmissor e no receptor para prover ganhos significativos na SNR (do inglês *Signal-Noise-Ratio*), mitigando a perda de propagação [1]. A conformação de feixe tem sido uma técnica chave para compensar a alta atenuação do canal e interferência através dos altos ganhos de diretividade. Devido à seleção espacial das antenas direcionais, a interferência co-canal é reduzida com a conformação de feixe [1]. Em sistemas mmWave é possível utilizar um grande número de antenas adaptativas direcionais com pequenos tamanhos e orientadas em várias direções, explorando as reflexões e sombreamentos causadas por objetos para maximizar a intensidade do sinal. A Figura 2.2 representa a transmissão direcional a partir da estação base para os dispositivos móveis, utilizando feixes estreitos de transmissão dentro de cada setor do arranjo de antenas.

Em redes celulares mmWave a conformação de feixe é utilizada tanto na estação base quanto no dispositivo do usuário. O transmissor seleciona um padrão de feixe de transmissão, que determina os pesos de deslocamento de fase para conduzir o feixe em uma determinada direção. Da mesma forma, o receptor seleciona um padrão de feixe para receber os sinais em uma certa dire-

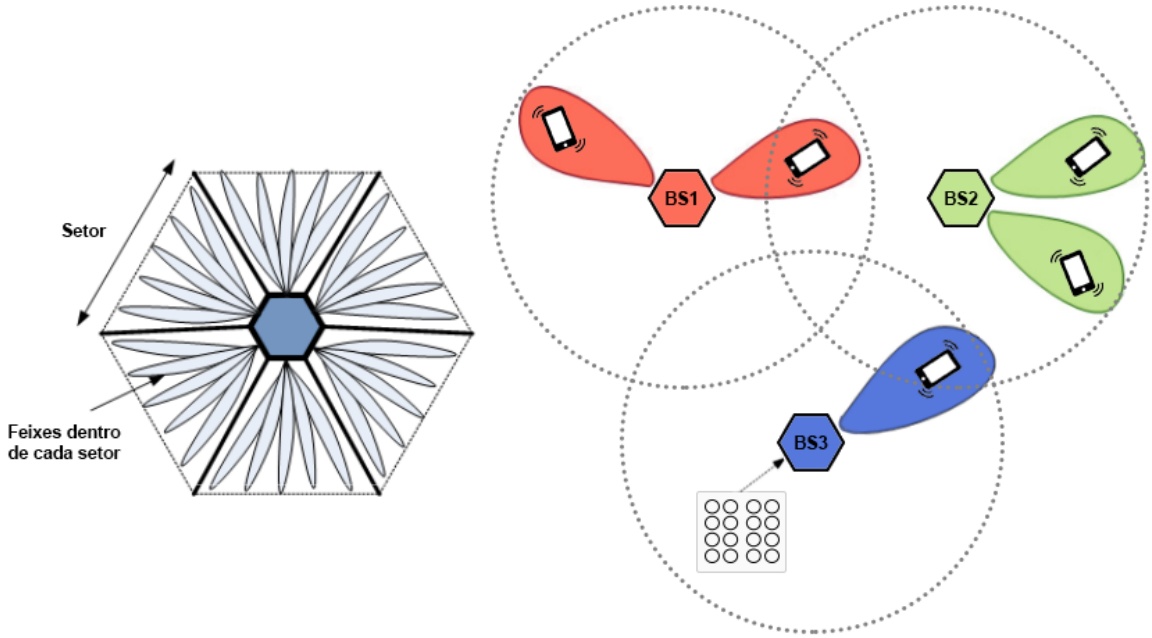

Figura 2.2: Transmissão direcional com conformação de feixe.

ção [12]. Assim, os feixes de transmissão e recepção precisam estar alinhados um com o outro para obter altos ganhos de conformação de feixe. Então, depois que o equipamento do usuário já estiver conectado em uma determinada célula, o equipamento do usuário envia periodicamente para a estação base o índice do melhor feixe de transmissão dessa estação base utilizando o canal de controle do *uplink*. O melhor feixe de transmissão do equipamento do usuário também é enviado periodicamente no canal de controle do *downlink* a partir da estação base para o equipamento do usuário. Após esse conhecimento dos melhores feixes de transmissão das duas extremidades do enlace, os dados podem ser transmitidos no melhor par de feixes tanto no *downlink* como no *uplink*.

Na conformação de feixe MIMO, o mesmo símbolo, ponderado por um fator de escala complexo, é enviado sobre cada antena. Considerando os vetores coluna de conformação de feixe $\mathbf{v}$ e $\mathbf{u}$ para a transmissão e recepção, respectivamente, o símbolo transmitido x é enviado sobre a i-ésima antena com peso v_i. Do outro lado, o sinal recebido na i-ésima antena é ponderado por u_i. Os vetores de peso de transmissão e recepção são normalizados, logo, $||\mathbf{v}|| = ||\mathbf{u}|| = 1$. Quando a matriz de canal $\mathbf{H}$ é conhecida pelo transmissor, a SNR recebida é otimizada escolhendo $\mathbf{v}$ e $\mathbf{u}$ como principais vetores singulares da matriz de canal $\mathbf{H}$. Quando a matriz de canal não é conhecida pelo receptor, os pesos das antenas transmissoras são todos iguais, resultando em uma menor SNR e capacidade do que a transmissão com a ponderação de transmissão ideal [24].

Existem diferentes arquiteturas de conformação de feixe. A arquitetura de conformação de feixe analógica utiliza apenas um canal RF (do inglês *Radio Frequency*) para focar o ganho da conformação de feixe na direção do percurso dominante [1]. A conformação de feixe digital exige um canal RF por antena, requerendo um ADC (do inglês *Analog to Digital Converter*) por cada

canal RF. A arquitetura híbrida implementa a multiplexação espacial e a conformação de feixe, reduzindo o número de canais RF e permitindo que múltiplos fluxos de dados sejam enviados em direções espaciais diferentes. A Figura 2.3 apresenta algumas arquiteturas de conformação de feixe presentes na literatura. A arquitetura de conformação de feixe digital é mais flexível do que a analógica e provê alto grau de liberdade, oferecendo um melhor desempenho apesar de sua maior complexidade [6]. Como na conformação de feixe totalmente digital há um canal RF separado e um ADC para cada antena, há um consumo de energia maior e também um maior custo. Como redes mmWave funcionam com um grande número de antenas e uma ampla largura de banda, essa técnica pode ser inviável para redes mmWave devido ao alto consumo de energia requerido por cada ADC em cada antena. Já a conformação de feixe analógica é mais simples e também provê altos ganhos de diretividade a partir do grande número de antenas, economizando energia usando um único ADC. A partir dessa perspectiva, a conformação de feixe analógica é mais apropriada para ser usada em redes mmWave.

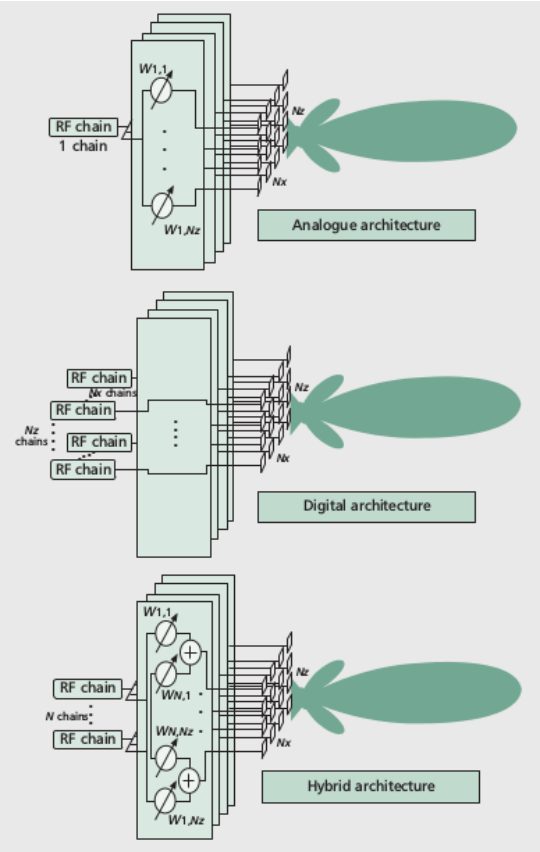

Figura 2.3: Arquiteturas de conformação de feixe. Fonte: [1].

2.4 DESAFIOS DA CAMADA MAC

As redes celulares mmWave precisam prover taxas de dados de múltiplos Gbps para os usuários a fim de suportar aplicações multimídia com requerimentos de QoS. Com isso, surgem alguns

desafios a serem explorados na camada MAC (do inglês *Medium Access Control*) para atender o aumento da demanda de tráfego móvel. Mudanças fundamentais são necessárias no design da camada MAC para redes celulares mmWave, principalmente em funcionalidades como sincronização, acesso aleatório, *handover*, gerenciamento de interferência, agendamento e associação [4]. Jian Qiao apresenta uma série de desafios de camada MAC para comunicações mmWave em seu trabalho descrito em [2], apresentados a seguir.

Devido à enorme largura de banda disponível, a comunicação mmWave pode prover uma grande capacidade. É possível explorar o reuso espacial por causa da alta perda de propagação e da transmissão direcional em redes mmWave, permitindo transmissões concorrentes para melhorar a capacidade agregada da rede. No entanto, múltiplos enlaces de comunicação simultâneos resultam em uma maior interferência multi-usuário, afetando a vazão do sistema [2]. Dessa forma, é necessário que mudanças sejam feitas na camada MAC para permitir as transmissões concorrentes de forma adequada.

Outro desafio importante é o alinhamento dos feixes entre o transmissor e o receptor. Como as antenas irradiam a maior parte da potência em certas direções, o transmissor e o receptor precisam do procedimento de conformação de feixe para alcançar uma maior capacidade na rede, apesar de não conhecerem a localização um do outro. Esse procedimento de conformação de feixe é mais complexo para enlaces de comunicação concorrentes, considerando a interferência mútua [2].

A camada MAC precisa ser revisada para lidar com a severa atenuação do canal, a direcionalidade e o bloqueio. A vulnerabilidade do canal físico mmWave é uma questão fundamental que deve influenciar o design da camada MAC. Devido à alta perda de propagação e bloqueio na banda mmWave, o enlace de comunicação fica sujeito à interrupções frequentes, principalmente no caso de usuários com alta mobilidade em áreas metropolitanas. Essas interrupções são preocupantes para a entrega do QoS necessário pelas aplicações.

Também é necessário discutir sobre o acesso inicial e o gerenciamento de mobilidade em redes mmWave. Essas funções da camada MAC especificam como o dispositivo do usuário deve se conectar à rede e preservar sua conectividade. Em um MAC baseado em contenção, os pacotes que chegam na estação base com maior potência são provenientes de usuários que estão mais próximos da estação base na célula. Geralmente, esses são os usuários que conseguem reservar os recursos de canal na presença de transmissões conflitantes, uma vez que a comunicação mmWave sofre uma perda de propagação severa com o aumento da distância. Por isso, é necessário desenvolver um protocolo CSMA/CA que proporcione equidade entre os usuários [2]. Apesar do nosso trabalho não ser focado na camada MAC, tratamos de aspectos da camada física que influenciam diretamente no funcionamento da camada MAC, principalmente sobre os aspecto do acesso inicial, que é abordado na próxima seção.

2.5 ACESSO INICIAL EM REDES MMWAVE

O acesso inicial permite que um equipamento de usuário móvel (UE, do inglês *user equipment*) estabeleça uma conexão com a estação base (BS, do inglês *base station*). Esse procedimento é necessário para que o UE tenha acesso à rede e é um procedimento crítico para redes mmWave, já que em transmissões direcionais esse passo só pode ser realizado após o estabelecimento do enlace físico. Nos sistemas LTE atuais o acesso inicial é realizado em canais omnidirecionais. Já em redes mmWave, se o acesso inicial for realizado em canais direcionais, deve ser incluída a fase de busca celular (*cell search*) para que a BS e o UE determinem primeiro as direções de transmissão adequadas para permitir a troca de mensagens de controle, já que o melhor par de feixes de transmissão não é previamente conhecido. Dessa forma, o acesso inicial em redes mmWave seria composto de duas etapas: 1) busca celular para determinar as direções de conformação de feixe iniciais da BS e do UE e 2) acesso aleatório para detectar a requisição de acesso do UE [15].

2.5.1 Busca de Feixes

O desalinhamento é particularmente crítico durante o acesso inicial do dispositivo móvel na célula, quando o UE precisa se conectar na estação base através de um esquema de acesso aleatório. Então, se um canal de controle físico direcional é desejado nesse estágio, a informação do estado do canal será necessária em ambas as extremidades do enlace para proporcionar a configuração do alinhamento dos feixes. Para conseguir isso, um número de trabalhos tem proposto algoritmos de pesquisa de feixes para encontrar os melhores pares de feixe que maximizam o *link budget* entre o transmissor e o receptor.

Wang [10] propôs um protocolo de conformação de feixe baseado em um *codebook* projetado para encontrar o melhor par de feixes para transmissão de dados. Seu protocolo mostrou reduzir o tempo de configuração do feixe em comparação com esquemas de busca exaustivos. Os resultados numéricos mostraram que o protocolo de conformação de feixe proposto reduziu o tempo de *set-up* de 31,57 ms do esquema de busca exaustiva para 619,782 μs. Li *et al.* [11] propuseram uma nova técnica que reduz a sobrecarga de protocolo e o consumo de energia usando um algoritmo de conformação de feixe que é baseado no algoritmo de Rosenbrock [25]. Também, Sung *et al.* [8] propuseram um mecanismo de escolha dos feixes controlado pelo UE para canais *uplink* baseados em contenção. Apesar desses avanços, essas abordagens ainda demandam uma sobrecarga de tempo extra para a configuração dos feixes e alinhamento, que pode ser crítico e prejudicial, especialmente durante o acesso inicial, uma vez que isso afeta a conexão do UE na célula, principalmente durante o *handover* entre diferentes estações base.

2.5.2 Acesso Aleatório

Em uma rede celular, um dispositivo móvel precisa estabelecer um enlace de rádio com a estação base para realizar a transmissão e recepção de dados. É provável que alguns dos procedimentos básicos e técnicas das atuais redes LTE sejam usadas nas futuras redes 5G, que ainda não possuem um padrão estabelecido. Consequentemente, de forma similar ao LTE, o esquema de acesso aleatório será usado no acesso inicial para permitir que cada UE execute um mecanismo de *handshake* básico com a BS. No LTE, o procedimento de acesso aleatório funciona com reserva de acesso, para que o UE reserve os recursos necessários para suas transmissões de dados no *uplink* usando um mecanismo baseado no *slotted ALOHA*.

O dispositivo móvel acessa a rede utilizando o RACH (do inglês *Random Access Channel*), para estabelecer uma enlace de rádio com a estação base. O RACH é formado por uma sequência de recursos alocados no tempo e frequência, denominados slots RA (do inglês *Random Access*). O RACH é utilizado para várias funções, como o acesso inicial, o *handover*, a sincronização de manutenção do *uplink* e a solicitação de agendamento [12].

O procedimento de acesso aleatório baseado em contenção no LTE é composto de 4 mensagens de controle diferentes entre o UE e a BS [26], como mostrado na Figura 2.4. No método baseado em contenção, a comunicação é iniciada pelo UE. Uma requisição de acesso só é concluída se as quatro mensagens forem trocadas com sucesso nos quatro passos de acesso aleatório a seguir.

- Passo 1 - Preâmbulo de acesso aleatório: Quando um UE ocioso tenta se conectar à rede LTE, ele transmite um preâmbulo de acesso aleatório, utilizando o RACH. Esse preâmbulo RACH é uma assinatura digital que o dispositivo transmite em um slot RA. Existem 64 preâmbulos pseudo-aleatórios ortogonais disponíveis para o RA. A estação base reserva alguns desses preâmbulos para o acesso livre de contenção. Cada dispositivo escolhe aleatoriamente um preâmbulo (dentro do conjunto de preâmbulos do acesso baseado em contenção) e transmite esse preâmbulo nos recursos de tempo e frequência indicados pela informação do sistema divulgada pela BS. Quando dois ou mais dispositivos utilizam o mesmo preâmbulo no mesmo slot RA, uma colisão acontece. Quando não há colisão, a estação base detecta os diferentes preâmbulos enviados devido à ortogonalidade entre eles [27].

- Passo 2 - Resposta de acesso aleatório (RAR): Quando a BS recebe o preâmbulo de acesso aleatório, a mensagem RAR (do inglês *Random Access Response*) é enviada para o UE a fim de identificar o preâmbulo recebido e atribuir um identificador temporário (RNTI, do inglês *Radio Network Temporary Identifier*) ao dispositivo, assim como atribuir recurso de tempo-frequência no canal *uplink* para o próximo passo [26]. Também são enviadas instruções para sincronizar as transmissões no *uplink*. Quando um dispositivo recebe a mensagem RAR associada ao slot RA em que o preâmbulo foi enviado, mas ela não contém o identificador do seu preâmbulo transmitido, ele aguarda um tempo de *backoff* e realiza o passo 1 novamente.

- Passo 3 - Requisição de conexão: Após receber a mensagem RAR, o UE envia uma mensagem de requisição de conexão para a BS utilizando o recurso de tempo e frequência designado no passo 2 solicitando o acesso inicial, incluindo informações sobre autenticação e identificação. Quando a colisão do preâmbulo RACH não é identificada pela BS no passo 1, a mesma mensagem RAR é enviada para mais de um dispositivo no passo 2, ocasionando colisão também no passo 3, já que os mesmos recursos de uplink são utilizados por mais de um usuário [27].
- Passo 4 - Resolução de contenção: Quando a requisição de acesso do UE não sofre colisão no passo 3 e é recebida pela BS, a BS responde à requisição de conexão com uma mensagem de resolução de contenção, alocando os blocos de recurso solicitados ou negando a requisição se não existirem recursos disponíveis. Quando uma colisão acontece no passo 3, nenhuma confirmação é transmitida pela BS. Assim, cada dispositivo retransmite a requisição de conexão através do mecanismo HARQ (do inglês *Hybrid Automatic Retransmission Request*), de acordo com o número máximo de retransmissões permitidas até a declaração de falha de acesso, sendo necessário agendar uma nova tentativa de conexão. A cada nova tentativa de conexão, um contador de transmissão é incrementado e se esse contador chega no seu valor máximo, a rede é declarada não disponível pelo dispositivo [27].

Figura 2.4: Procedimento de acesso aleatório baseado em contenção de acesso.

Assumindo que a BS e o UE utilizem conformação de feixe, quando um UE faz um acesso inicial à rede usando um RACH, os melhores pares de feixe não são conhecidos até que o procedimento de busca celular seja executado, o que dificulta a escolha dos feixes de transmissão e recepção do preâmbulo RACH. Para obter um alto ganho de conformação de feixe, os feixes de transmissão e recepção do preâmbulo precisam estar alinhados. Se o UE transmitir o preâmbulo RACH em múltiplas direções, apenas algumas dessas transmissões conseguirão obter o ganho máximo de conformação de feixe quando os feixes de transmissão e recepção estiverem alinhados. Por isso é necessário utilizar algum mecanismo de busca de feixes para realizar o acesso aleatório, a fim de selecionar os melhores feixes de transmissão e recepção.

Como mencionado anteriormente, os procedimentos de busca de feixe podem incluir um atraso adicional na etapa de acesso aleatório. A longa duração do acesso aleatório pode impactar em aspectos cruciais das redes mmWave, como abordado por Jeong *et al.* em [12], onde é apresentado o impacto de um acesso aleatório de longa duração nos aspectos de acesso inicial, falha de enlace de rádio, *handover*, configuração do *uplink* e *downlink* utilizando o TDD (do inglês *time-division duplex*) e agendamento de feixes.

A fim de reduzir o consumo de energia do dispositivo móvel, o dispositivo deve permanecer o menor tempo possível no estado de transição do estado ocioso para o estado conectado, mas se o acesso aleatório precisar de uma duração prolongada devido ao alinhamento de feixes, essa fase de transição terá uma duração maior, consumindo mais energia do dispositivo móvel. Outro impacto da longa duração do acesso aleatório é em caso de RLF (do inglês *Radio Link Failure*), em que o dispositivo precisa reestabelecer sua conexão com a BS após uma falha no enlace de rádio. O tempo de interrupção de serviço não deve ser longo para não afetar a qualidade da experiência do usuário. Portanto, é importante que o dispositivo consiga se conectar na célula rapidamente, exigindo um procedimento rápido de acesso aleatório.

O procedimento de *handover* também sofre o impacto de uma longa duração do procedimento de acesso aleatório. Devido à configuração de um grande número de pequenas células em redes mmWave, um dispositivo móvel pode mudar de célula frequentemente, conforme mostrado na Figura 2.5, em que o dispositivo móvel UE3 se move de uma célula para a outra e precisa se conectar à nova estação base BS2. O tempo de *handover* deve ser curto para garantir a qualidade de serviço do usuário, especialmente em serviços de tempo real, como voz sobre IP (VoIP). É importante que a movimentação do dispositivo de uma célula para outra seja praticamente imperceptível, sem exigir um tempo longo de conexão do usuário na BS a cada vez que o dispositivo mudar de célula.

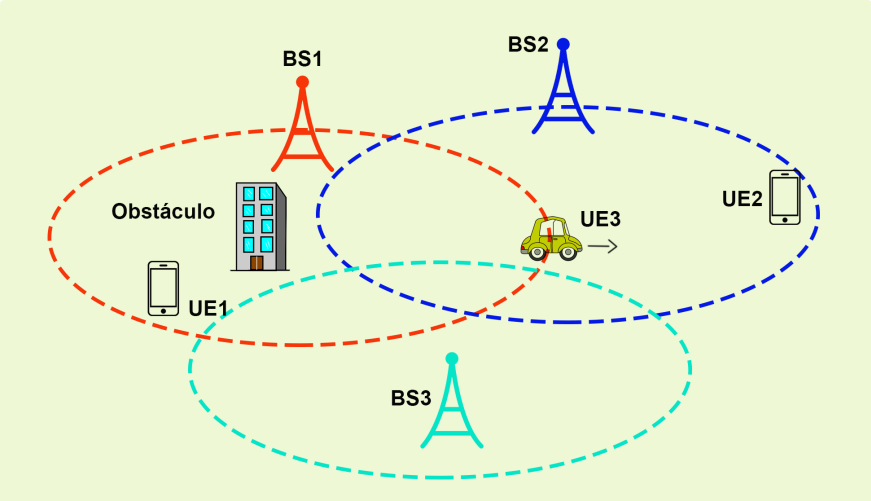

Figura 2.5: Representação do *handover* de um dispositivo móvel de uma célula para outra.

Ao considerarmos uma transmissão direcional do preâmbulo RACH, é preciso levar em conta que o procedimento de alinhamento de feixes precisa ser executado para que a transmissão seja

feita na melhor direção, caso contrário, a transmissão será feita em múltiplas direções, demandando uma duração maior do preâmbulo RACH [12]. Frente ao impacto que a longa duração do acesso aleatório provoca nas redes mmWave, é importante repensar no modo de realizar o canal de controle físico durante o acesso inicial, permitindo que o procedimento de acesso aleatório tenha uma curta duração para manter a qualidade do serviço percebida pelo usuário. Dessa forma, o procedimento de acesso aleatório representa um aspecto crucial para o planejamento de redes mmWave.

3 REVISÃO BIBLIOGRÁFICA

Neste capítulo é apresentada uma revisão bibliográfica dos principais trabalhos relacionados ao acesso inicial em redes mmWave que utilizamos ao longo do desenvolvimento desse trabalho. Apresentamos os principais pontos de alguns trabalhos relacionados com o acesso aleatório em redes mmWave, incluindo abordagens sobre um canal de controle direcional, omnidirecional e fora da banda mmWave. Apesar de poucos trabalhos abordarem a questão do acesso inicial em redes mmWave, reportamos alguns resultados relevantes encontrados na literatura.

Jeong *et al.* [12] aborda questões fundamentais do RACH, canal de acesso aleatório, em comunicações celulares mmWave e apresenta possíveis abordagens para resolver essas questões. O principal problema abordado nesse trabalho é que o acesso aleatório não é totalmente beneficiado pela técnica de conformação de feixe devido à falta de informação do melhor feixe de transmissão e recepção, principalmente em canais NLOS (do inglês *Non-line-of-sight*). Quando um dispositivo acessa a rede inicialmente, utilizando o RACH, ele não tem a informação do melhor par de feixes de transmissão. Então, existe uma dificuldade para o dispositivo do usuário escolher a melhor direção de transmissão para o envio do preâmbulo RACH, assim como para a estação base escolher a melhor direção de recepção do preâmbulo. Esse artigo analisa o desempenho do RACH com o envio do preâmbulo em múltiplas direções com antena direcional e antena omnidirecional. De acordo com os resultados obtidos, os ganhos de desempenho com antena direcional sobre o uso de antena omnidirecional ainda são mantidos mesmo que o preâmbulo não seja transmitido sempre com os melhores pares de feixe. Mas a duração do preâmbulo RACH pode ser muito maior quando o preâmbulo é enviado em múltiplas direções do que quando ele é enviado na melhor direção, já que a maioria das transmissões do preâmbulo não alcançam altos ganhos de conformação de feixe devido à falta de alinhamento da direção dos feixes de transmissão e recepção. Assim, a transmissão do preâmbulo RACH em várias direções demanda um atraso maior, pois apenas as transmissões nas melhores direções são recebidas com sucesso. A longa duração do preâmbulo RACH pode impactar em alguns procedimentos importantes, como o acesso inicial, a recuperação de falha de enlace de rádio e o *handover* entre pequenas células. Algumas possíveis abordagens são apresentadas para resolver o problema da longa duração da transmissão do preâmbulo RACH. Uma das propostas apresentadas pelos autores é melhorar a detecção do preâmbulo pela estação base desenvolvendo um novo algoritmo de detecção levando em conta a seletividade de frequência do canal. Outras possíveis soluções seriam utilizar múltiplos canais digitais na estação base, explorar a reciprocidade do canal no modo TDD e planejar as células de forma que as estações base sejam alocadas em lugares com linha de visada para os dispositivos móveis.

Shokri-Ghadikolaei *et al.* [4] discutem as implicações que uma comunicação altamente direcional traz para um modelo eficiente de camada MAC. Esse artigo discute questões importantes relacionadas à camada MAC, como a sincronização, o acesso inicial, o *handover*, entre outras.

São apresentados dois tipos de compromissos essenciais em canais de controle físicos para redes mmWave: o compromisso entre enviar as mensagens de controle sobre frequências de micro-ondas ou em mmWave e o compromisso de direcionalidade, que se refere à opção de enviar as mensagens de controle sobre um canal de controle físico (PHY-CC, do inglês *Physical Control Channel*) omnidirecional (tanto a BS quanto o UE são omnidirecionais), semi-direcional (a BS ou o UE é omnidirecional enquanto o outro é direcional) ou totalmente direcional (tanto a BS quanto o UE são direcionais).

Um canal de controle físico em banda mmWave está sujeito a uma forte atenuação e bloqueio, já a banda de microondas facilita o envio de mensagens de difusão e a sincronização da rede, proporcionando uma cobertura maior e um enlace mais estável. Por outro lado, um canal de controle físico sobre microondas exige uma maior complexidade de hardware e maior consumo de energia [4]. Em relação à direcionalidade, um PHY-CC em modo omnidirecional possui um curto alcance em distância, mas pode receber as mensagens de controle sem o problema da surdez. Já o PHY-CC semi-direcional aumenta o alcance em distância e introduz menos interferência na rede, mas exige uma busca espacial para mitigar o problema da surdez, introduzindo um atraso extra da busca espacial para realizar o alinhamento dos feixes de transmissão. Por último, um PHY-CC no modo totalmente direcional aumenta ainda mais a cobertura com menos interferência, mas exige um atraso ainda maior de pesquisa espacial. Os resultados das simulações em [4] mostraram que a cobertura de um PHY-CC omnidirecional foi a menor se comparado com as outras opções de PHY-CC, devido à falta do ganho de diretividade e à forte atenuação do canal mmWave. Já o PHY-CC totalmente direcional obteve a maior cobertura das três opções, exigindo menos estações base com linha de visada para obter uma cobertura mínima de 97%.

As transmissões direcionais em redes mmWave provocam um grande impacto nos procedimentos da camada de controle. Como já mencionado, o acesso inicial pode ser prolongado significativamente devido à necessidade da estação base e do usuário de encontrarem os melhores feixes para a transmissão e recepção direcional. Giordani *et al.* [15, 13] apresentam uma análise de técnicas propostas recentemente para realizar esse alinhamento dos feixes. Foram avaliados três procedimentos de busca na célula para o acesso inicial, focando nas técnicas de conformação de feixe analógica: busca exaustiva, busca iterativa e busca baseada em informação do contexto.

A busca exaustiva é feita aplicando a técnica de busca de feixe sequencialmente por força bruta. Os usuários e as estações base possuem um *codebook* pré-definido de N direções que cobrem todo o espaço angular e são usadas sequencialmente para transmitir e receber, sendo que cada direção é identificada com um vetor de conformação de feixe. Já a busca iterativa é composta de dois estágios de exploração do espaço angular. Na primeira fase, a estação base transmite sobre todos os setores mais largos, e na segunda fase essa pesquisa é refinada dentro do melhor desses setores, estreitando os feixes de transmissão. A busca baseada em contexto é baseada em um algoritmo que possui três estágios. No primeiro estágio, a macro estação base que opera em frequências do LTE espalha as coordenadas de GPS de todas as estações mmWave dentro de um alcance omnidirecional. No segundo estágio cada equipamento de usuário recebe suas coordenadas GPS, o que requer um custo de energia. No último estágio cada UE seleciona

uma estação base mais perto geometricamente a partir das informações obtidas nos estágios 1 e 2, e então direciona seus feixes de transmissão para essa estação base. Enquanto isso, a estação base mmWave faz uma busca exaustiva para detectar a melhor direção de transmissão e recepção.

No estudo realizado em [15] o desempenho dos procedimentos de acesso inicial foram avaliados em termos do atraso de descoberta, que é o tempo requerido pela BS e UE para determinar as melhores direções e alinhar os seus feixes, e a probabilidade de não-detecção, que é a probabilidade da UE dentro da célula não ser detectada pela BS na fase de busca celular. A principal conclusão dos autores foi que a busca exaustiva provavelmente será a melhor configuração de acesso inicial para a busca celular, principalmente se for desejada uma boa cobertura em distâncias relativamente grandes, por exemplo em caso de usuários de borda em grandes células. A busca iterativa deve ser preferida em caso de pequenas células em que as distâncias entre a estação base e o usuário forem menores. Entretanto, a melhor técnica geralmente vai depender da SNR alvo e do cenário considerado. A técnica de algoritmo puramente baseado em informação de contexto sem um refinamento não é adequada para cenários urbanos sem linha de visada, apesar de ter o potencial de reduzir o atraso de descoberta e garantir uma boa cobertura.

Em [13] é mostrado que existe um compromisso entre o atraso do acesso inicial e a probabilidade de não-detecção do UE na célula. Nesse trabalho, o desempenho do acesso inicial em uma célula em 28 GHz é avaliado através do atraso de descoberta e da probabilidade de não-detecção utilizando duas diferentes técnicas de acesso inicial, a técnica exaustiva e a técnica iterativa. O atraso de descoberta é o tempo que a estação base necessita para identificar todos os usuários no seu raio de cobertura. A probabilidade de não detecção é a probabilidade de um usuário dentro da célula não ser identificado pela BS, obtendo uma SNR abaixo do limiar de -5 db. A probabilidade de não-detecção obtida com a técnica iterativa é de 0,4 para a distância UE-BS de 100 m e 0,15 com a técnica exaustiva, considerando a configuração de antenas 64×16. O atraso de descoberta obtido com as simulações, considerando 144 slots enviados no acesso inicial e com a configuração de antenas 64×16, é 28,8 ms utilizando a técnica exaustiva, e 8,8 ms utilizando a técnica iterativa para o acesso inicial. As técnicas iterativas requerem menos slots de tempo para realizar a busca angular em comparação à busca exaustiva, mas apresentam maiores probabilidades de não-detecção. Considerando uma cobertura mínima de cerca de 100 metros para um canal mmWave urbano e com múltiplos percursos, os procedimentos de busca exaustivas são preferíveis por apresentarem um atraso total menor quando comparado à outras técnicas de acesso inicial, considerando o pior cenário com uma probabilidade de não detecção menor do que 0,01 para os usuários de borda. Ainda assim, o atraso total obtido para a busca exaustiva chega a 360 ms em uma célula com alta densidade de usuários.

Barati *et al.* [14] propõem a inclusão de uma nova etapa no procedimento de acesso aleatório do 3GPP LTE. É proposta a inclusão de um passo de detecção de sinal de sincronização antes do passo de envio do preâmbulo de acesso aleatório para que o UE e a BS determinem as direções da conformação de feixe iniciais, além de detectar a presença da BS e a solicitação de acesso do UE. Nessa etapa, cada BS transmite periodicamente um sinal de sincronização que o UE utiliza para detectar a presença da estação base e obter a duração do quadro de *downlink*. Esse

sinal de sincronização também é utilizado para determinar a direção de conformação de feixe do UE, que está relacionada aos ângulos de chegada dos percursos do sinal emitido pela BS. Dessa forma, no passo de sincronização do sinal, o UE determina sua direção de conformação de feixe e no passo em que o UE envia o preâmbulo de acesso aleatório, a BS determina sua direção de conformação de feixe. Nesse trabalho também é considerado o envio do sinal de sincronização de forma omnidirecional, assim como o envio do preâmbulo de acesso aleatório. Os autores propõem uma combinação de transmissão omnidirecional, direcional e direcional com conformação de feixe digital para realizar a sincronização e o acesso aleatório. São apresentadas diferentes opções como o envio do sinal de sincronização pela BS de forma omnidirecional, a recepção desse sinal pelo UE de forma direcional e o envio do preâmbulo de acesso aleatório pelo UE de forma direcional. De acordo com a avaliação de desempenho realizada, no geral, o menor atraso na etapa de sincronização foi obtida com uma transmissão omnidirecional do sinal de sincronização. Na fase de acesso aleatório após a sincronização, o menor atraso obtido foi com a opção de enviar o preâmbulo direcionalmente com a conformação de feixe digital. Apesar das contribuições mencionadas desse trabalho, não foram feitas análises considerando todos os passos do procedimento de acesso aleatório sobre uma canal omnidirecional.

Recentemente, alguns trabalhos [21, 9] propuseram uma solução fora da banda mmWave para adquirir informação em bandas de microondas para estimar o estado do canal em bandas mmWave, uma vez que a sincronização e a transmissão por difusão são facilitadas em frequências de microondas. Precilc *et al.* [21] propuseram reduzir a sobrecarga usando informações provenientes de redes de microondas coletadas a partir de sensores ou outros sistemas de comunicação operando em frequências abaixo de 6 GHz. Então, a informação é extraída a partir de bandas de microondas a fim de adquirir informação sobre o canal para a banda mmWave.

Apesar de algumas vantagens, como a redução da sobrecarga da busca de feixes, a exploração da informação fora da banda ainda apresenta desafios de pesquisa que permanecem sem solução, que requerem algoritmos de processamento de sinais apropriados e esforços exaustivos em medições multi-bandas [21]. A incompatibilidade dos parâmetros de canal é o maior desafio, desde que podem haver incompatibilidades entre os ângulos de chegada e o espalhamento angular usando informações de outra banda na banda mmWave. Shokri-Ghadikolaei *et al.* [20] discutem que não é apropriado usar um canal de controle em redes de microondas para estimar o canal mmWave a fim de alcançar uma apropriada conformação de feixe. A sincronização dos sinais sobre uma banda de microondas não pode prover informação suficiente para realizar a sincronização espacial na banda mmWave devido às diferenças nas características de propagação.

4 ACESSO INICIAL OMNIDIRECIONAL COM ESQUEMA DE ALAMOUTI

Assumindo uma comunicação direcional para realizar o acesso inicial, os feixes de transmissão tanto do UE quanto da BS precisarão ser corretamente alinhados antes do UE tentar se conectar via procedimento de acesso aleatório. A vantagem de ter um PHY-CC com conformação de feixe é a alta cobertura proporcionada pelo ganho de diretividade [1], desde que somente os UEs que forem conectados com sucesso no acesso inicial conseguirão transmitir sobre os canais de dados direcionais. Em outras palavras, o potencial total e a cobertura direcional através da conformação de feixe só farão efeito no canal de dados se primeiro os UEs conseguirem se conectar à BS com sucesso no canal de controle.

Como explicado no Capítulo 2, o alinhamento dos feixes para realizar a comunicação direcional incorre em um atraso extra, que pode vir a ser crítico especialmente no acesso inicial, durante o *handover* entre as células e em caso de falha do enlace de rádio. As técnicas de busca de feixe como a busca exaustiva ou a busca iterativa requerem um atraso para determinar as direções iniciais de transmissão mais adequadas, retardando o procedimento de acesso inicial em que o UE e a BS estabelecem um enlace físico de rádio. Normalmente, são adotadas estratégias simples e rápidas para o acesso inicial, pois a longa duração do processo de acesso inicial pode impactar a qualidade de serviço do usuário, principalmente em serviços de tempo real. Além disso, devido ao atraso demandado para realizar o procedimento de alinhamento de feixes, o dispositivo permanece mais tempo no estado de transição do estado ocioso para o estado conectado, consumindo mais energia do dispositivo móvel [12].

Para evitar uma longa duração do acesso inicial devido ao alinhamento de feixes, o acesso inicial pode ser realizado em modo omnidirecional. Um canal de controle omnidirecional alivia o problema da surdez, já que não demanda um alinhamento entre os feixes de transmissão e recepção. Entretanto, enquanto evita o problema da surdez, um PHY-CC omnidirecional pode introduzir um problema de incompatibilidade devido às possíveis assimetrias entre a cobertura alcançada na transmissão sobre canais de controle e a cobertura alcançada na transmissão sobre canais de dados, desde que é esperado que os canais de dados operem com conformação de feixe (que normalmente pode alcançar maiores distâncias). Essa incompatibilidade pode levar a tamanhos de células menores, uma vez que apenas os UEs que estiverem próximos seriam capazes de se conectar à rede e, então, transmitir sobre o canal de dados. Portanto se medidas não forem tomadas, ou se não forem investigadas outras abordagens, o uso de um canal de controle omnidirecional pode fazer com que a cobertura na célula seja muito pequena. Alguns trabalhos na literatura apresentam a opção de realizar o canal físico de controle em modo omnidirecional em banda mmWave [14, 4, 20], mas mencionam a distância de cobertura como um fator limitante.

Nós propomos o uso do Esquema de Alamouti [16] com o propósito de implementar um PHY-

CC omnidirecional dentro da banda para o acesso inicial em redes mmWave. Nossa proposta é que apenas durante a fase de acesso aleatório as mensagens do canal de controle sejam transmitidas em modo omnidirecional. Assim, apenas os passos de acesso aleatório seriam realizados em um PHY-CC omnidirecional, incluindo o envio do preâmbulo RACH, a mensagem RAR, a requisição de conexão e a mensagem de resolução de contenção, completando a conexão do usuário na rede. Dessa forma, a fase de acesso aleatório não exigiria o atraso demandado pelo procedimento de alinhamento de feixes, permitindo que o usuário se conecte rapidamente à rede. Somente após a conexão do usuário na rede, o alinhamento dos feixes seria executado a fim de iniciar a transmissão dos pacotes de dados de forma direcional, com as direções de conformação de feixe já conhecidas pela BS e pelo UE. A transmissão omnidirecional das mensagens de controle do acesso aleatório evita o problema da surdez e não introduz uma interferência significativa, por se tratar de uma troca rápida de pacotes tipicamente pequenos.

Dada a abundância de antenas esperada tanto no UE quanto na BS em redes mmWave, o uso da técnica de diversidade com o propósito de realizar a conexão do UE pode evitar o atraso extra introduzido pelos algoritmos de busca de feixe e fornecer cobertura similar ao PHY-CC direcional se os parâmetros forem ajustados apropriadamente. Sendo uma técnica de diversidade de transmissão, o Esquema de Alamouti entrega ganhos de diversidade significativos, especialmente em cenários com baixa SNR, permitindo uma alta cobertura de célula às custas do baixo processamento. Isto porque o Esquema de Alamouti requer estimação do canal somente no receptor e tem processamento linear e decodificação simples. O Esquema de Alamouti já é adotado em diversos sistemas que utilizam MIMO, por exemplo, em sistemas IEEE 802.11n [28]. Neste trabalho, nós aproveitamos as múltiplas antenas disponíveis tanto no UE quanto na BS para implementar esquemas $2 \times N$ em cada direção do enlace. Na próxima seção apresentamos uma revisão dos principais conceitos do Esquema de Alamouti.

4.1 DIVERSIDADE DE TRANSMISSÃO - ESQUEMA DE ALAMOUTI

A diversidade de transmissão é uma técnica efetiva para combater o desvanecimento em comunicações móveis sem fio, sendo amplamente utilizada para reduzir o efeito do desvanecimento multi-percurso. A diversidade de transmissão é desejável em sistemas celulares, em que há mais disponibilidade de espaço, energia e capacidade de processamento no lado do transmissor. Na diversidade de transmissão existem múltiplas antenas transmissoras com a potência de transmissão dividida entre essas antenas. O modelo de diversidade de transmissão a ser usado depende se os ganhos complexos do canal são conhecidos pelo transmissor ou não. Sem o conhecimento do canal no transmissor, o ganho de diversidade de transmissão requer a combinação da diversidade no espaço e tempo [24] através do Esquema de Alamouti.

O Esquema de Alamouti [16] é um esquema de Codificação de Bloco Espaço-Temporal (STBC) que combina diversidade espacial e diversidade temporal. Este esquema suporta a detecção de máxima verossimilhança baseada somente no processamento linear do receptor. Além

disso, o esquema Alamouti não requer nenhuma retroalimentação do receptor para o transmissor, ou seja, não é necessário que o transmissor tenha conhecimento sobre o canal, sem incorrer em qualquer expansão de largura de banda [29]. Neste trabalho, assumimos que o equipamento do usuário que deseja se conectar à rede não tem conhecimento do canal na fase de acesso inicial, sendo apropriado utilizar o Esquema de Alamouti para realizar a transmissão durante o acesso inicial.

No Esquema de Alamouti são utilizados dois períodos de símbolo para a transmissão, assumindo que o ganho do canal permanece constante sobre esses dois períodos de símbolo consecutivos. Considerando a transmissão em banda base com duas antenas transmissoras, no primeiro período de símbolo são transmitidos dois símbolos diferentes s_1 e s_2 simultaneamente. O símbolo s_1 é transmitido a partir da antena 1, e o símbolo s_2 é transmitido a partir da antena 2. No próximo período de símbolo, o símbolo $-s_2^*$ é transmitido a partir da antena 1 e o símbolo s_1^* é transmitido a partir da antena 2, sendo * a operação de complexo conjugado. A energia total transmitida E_s é dividida igualmente entre os dois símbolos a cada período, sendo que cada símbolo é transmitido com uma energia $E_s/2$.

Para o caso de duas antenas transmissoras e uma antena receptora, assumimos que o ganho complexo do canal entre a i-ésima antena transmissora e a antena receptora seja h_i, com i = 1,2. O símbolo recebido no primeiro período de símbolo é $y_1 = h_1 s_1 + h_2 s_2 + n_1$ e o símbolo recebido no segundo período de símbolo é $y_2 = -h_1 s_2^* + h_2 s_1^* + n_2$ onde $n_i,\ i = 1, 2$ é a amostra do ruído AWGN (do inglês *Additive White Gaussian Noise*) no receptor, associada com a transmissão do i-ésimo símbolo. É assumido que a amostra de ruído tem média zero e energia E_n.

O receptor usa os símbolos recebidos para formar o vetor $\mathbf{y} = [y_1 y_2^*]^T$ dado por

$$\mathbf{y} = \begin{bmatrix} h_1 & h_2 \\ h_2^* & -h_1^* \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} + \begin{bmatrix} n_1 \\ n_2^* \end{bmatrix} = \mathbf{H}_A \mathbf{s} + \mathbf{n}, \tag{4.1}$$

onde $\mathbf{s} = [s_1\ s_2]^T$, $\mathbf{n} = [n_1\ n_2]^T$ e

$$\mathbf{H}_A = \begin{bmatrix} h_1 & h_2 \\ h_2^* & -h_1^* \end{bmatrix}. \tag{4.2}$$

Presumindo que os ganhos complexos de canal da matriz $\mathbf{H}_A$ são estimados corretamente, o vetor $\mathbf{z}$ é definido como $\mathbf{z} = \mathbf{H}_A^H \mathbf{y}$. A estrutura de $\mathbf{H}_A$ implica que

$$\mathbf{H}_A^H \mathbf{H}_A = (|h_1^2| + |h_2^2|)\mathbf{I}_2 \tag{4.3}$$

é diagonal, em que $|h_i|$ é a norma do ganho complexo de canal para a i-ésima antena transmissora e a antena receptora, e então

$$\mathbf{z} = [z_1\ z_2]^T = (|h_1^2| + |h_2^2|)\mathbf{I}_2 \mathbf{s} + \tilde{\mathbf{n}}, \tag{4.4}$$

onde $\tilde{\mathbf{n}} = \mathbf{H}_A^H \mathbf{n}$ é o vetor de ruído complexo Gaussiano com média zero e matriz de covariância $E[\tilde{\mathbf{n}}\tilde{\mathbf{n}}^*] = (|h_1^2| + |h_2^2|)E_n\mathbf{I}_2$. Cada componente de $\mathbf{z}$ corresponde a um dos símbolos transmitidos:

$$z_i = (|h_1^2| + |h_2^2|)s_i + \tilde{n}_i, i = 1, 2. \tag{4.5}$$

A SNR recebida corresponde à SNR para z_i, que será dada por [24]

$$SNR = \frac{(|h_1^2| + |h_2^2|)E_s}{2E_n}, \tag{4.6}$$

onde o fator 2 vem do fato de que s_i é transmitido usando metade da energia total de símbolo E_s. A SNR recebida é então igual à soma da SNR de cada componente. Logo, o Esquema de Alamouti alcança uma diversidade de ordem 2 para um sistema com duas antenas transmissoras.

O Esquema de Alamouti básico consiste de duas antenas transmissoras e uma antena receptora, constituindo um sistema MISO (do inglês *Multiple Input Single Output*), mas esse sistema pode ser facilmente generalizado para o caso de duas antenas transmissoras e N antenas receptoras para prover a diversidade de ordem $2N$ [29]. O Esquema de Alamouti pode ser aplicado, por exemplo, a um sistema MIMO (do inglês *Multiple Input Multiple Output*) com duas antenas receptoras. A Figura 4.1 apresenta o esquema de diversidade de transmissão com duas antenas de transmissão e duas antenas de recepção (2×2). A cada período de símbolo dois sinais são transmitidos simultaneamente a partir das duas antenas. O combinador no receptor combina os sinais recebidos e os envia para o detector de máxima verossimilhança, que então aplica o critério de decisão.

No caso de um sistema MIMO com duas antenas transmissoras e duas antenas receptoras, a matriz de canal $\mathbf{H}$ é dada por

$$\mathbf{H} = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \tag{4.7}$$

e os sinais recebidos y_1 e y_2 nas antenas receptoras sobre os períodos de símbolo consecutivos são dados por

$$\mathbf{y}_1 = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} + \begin{bmatrix} n_1 \\ n_2 \end{bmatrix}, \tag{4.8}$$

$$\mathbf{y}_2 = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} -s_2^* \\ s_1^* \end{bmatrix} + \begin{bmatrix} n_3 \\ n_4 \end{bmatrix}. \tag{4.9}$$

Assim, o vetor $\mathbf{y}$ formado com a sequência de símbolos recebidos no receptor é dado por

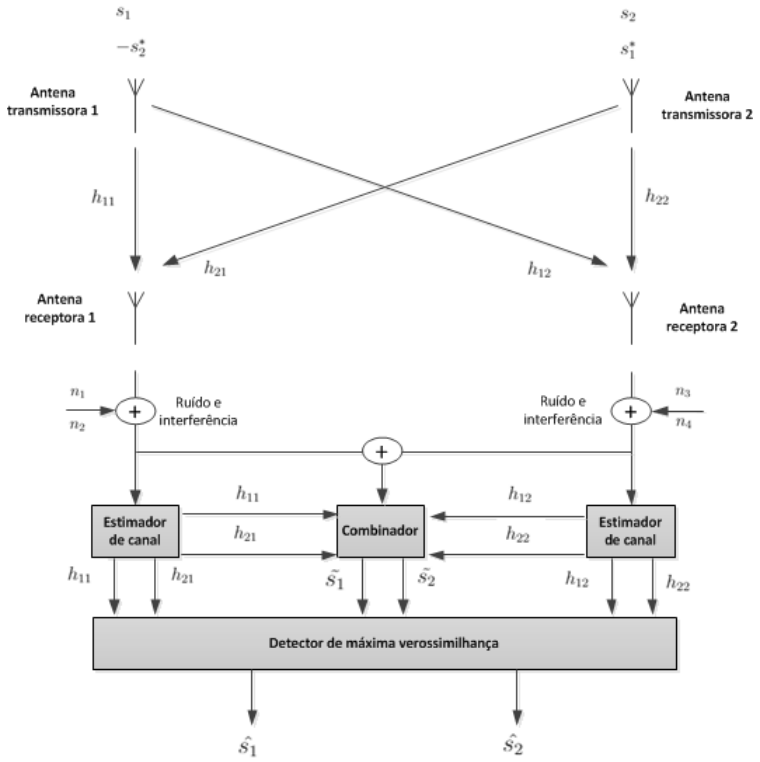

Figura 4.1: Esquema de diversidade de transmissão Alamouti em um sistema MIMO 2×2.

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2^* \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \\ h_{12}^* & -h_{11}^* \\ h_{22^*} & -h_{21}^* \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} + \begin{bmatrix} n_1 \\ n_2 \\ n_3^* \\ n_4^* \end{bmatrix} = \mathbf{H}_A \mathbf{s} + \mathbf{n}, \tag{4.10}$$

onde $\mathbf{s} = [s_1\ s_2]^T$ e $\mathbf{n} = [n_1\ n_2\ n_3^*\ n_4^*]^T$. Dessa forma, o vetor $\mathbf{z}$ pode ser escrito como

$$\mathbf{z} = \|\boldsymbol{H}\|_F^2 \mathbf{I}_2 \mathbf{s} + \tilde{\mathbf{n}}, \tag{4.11}$$

onde $\|\boldsymbol{H}\|_F^2$ é a norma de Frobenius, definida por

$$\|\boldsymbol{H}\|_F^2 = \sum_m \sum_n |h_{mn}|^2. \tag{4.12}$$

Então, a SNR recebida para um sistema MIMO aplicando-se o esquema Alamouti é dada por

$$SNR = \frac{\|\boldsymbol{H}\|_F^2 E_s}{2E_n}. \tag{4.13}$$

Em nossa proposta, aproveitamos o número máximo de N antenas do receptor para implementar um esquema Alamouti $2 \times N$ no *downlink* e *uplink*, a fim de garantir o ganho máximo de diversidade na BS e no UE. Dessa forma, é possível compensar a alta perda de propagação

da banda mmWave através do ganho de diversidade proporcionado pelo esquema Alamouti em uma comunicação omnidirecional durante o acesso inicial do UE na rede. Com uma transmissão omnidirecional, o UE não precisa executar um procedimento de busca de feixe para alinhar seu feixe de transmissão com a BS durante o acesso inicial, evitando assim, o atraso introduzido por esse procedimento. O ganho de diversidade do Esquema de Alamouti permite que a transmissão do UE alcance uma distância de cobertura maior para que a BS receba o preâmbulo RACH com sucesso e os passos do acesso aleatório sejam completados. Nos próximos capítulos apresentamos o modelo de simulação utilizado para implementar nossa proposta e os resultados obtidos com as simulações.

5 DESCRIÇÃO DO SISTEMA E MODELO

Neste capítulo fazemos uma breve revisão do modelo de canal mmWave e da operação da conformação de feixe implementada por Mezzavilla *et al.* [30] para o simulador ns-3 [31], assim como nossa abordagem para implementar o esquema Alamouti no mesmo simulador.

5.1 MODELO DE CANAL

Mezzavilla *et al.* [30] implementaram um módulo mmWave no simulador ns-3 [31] que compreende os modelos de propagação e de canal, as camadas física e MAC, e uma implementação básica de dispositivos mmWave. O modelo de canal assumido é o modelo apresentado por Akdeniz *et al.* [5]. Para caracterizar o padrão espacial da antena, foi seguido um modelo padrão da especificação 3GPP/ITU MIMO. No modelo 3GPP/ITU MIMO, o canal é assumido como composto de um número aleatório de K grupos de caminhos de propagação, em que cada grupo corresponde a um caminho de espalhamento em um nível macro, e cada caminho ou sub-grupo é composto de vários sub-caminhos. A Figura 5.1 representa o modelo de grupos e sub-caminhos de transmissão e recepção do sinal. Na Figura 5.1, o sinal parte da estação base para o dispositivo móvel. Esse sinal é transmitido através de diversos grupos de propagação devido ao desvanecimento do canal, sendo que as linhas próximas representam sub-grupos de caminhos de propagação do sinal. Cada grupo é descrito por:

- Uma fração da potência total;
- Ângulos de chegada e partida do azimute central (horizontal) e elevação (vertical);
- Espalhamento de feixes angulares ao redor dos ângulos centrais e;
- Atraso absoluto com relação ao tempo de propagação do grupo, e o perfil de atraso de potência em relação a este grupo.

Para M antenas transmissoras e N antenas receptoras, e K grupos com L sub-caminhos cada, o ganho de canal de banda estreita, variável no tempo, entre o transmissor e o receptor pode ser representado pela matriz $\boldsymbol{H}(t, f)$ de dimensões $N \times M$ a seguir [30] [5]:

$$\boldsymbol{H}(t, f) = \sum_{k=1}^{K} \sum_{l=1}^{L} g_{kl}(t, f) \boldsymbol{u}_{rx}(\theta_{kl}^{rx}, \phi_{kl}^{rx}) \boldsymbol{u}_{tx}^{*}(\theta_{kl}^{tx}, \phi_{kl}^{tx}), \tag{5.1}$$

sendo $g_{kl}(t, f)$ o ganho complexo de desvanecimento em pequena escala no l-ésimo sub-caminho do k-ésimo grupo, e $\boldsymbol{u}_{rx}(.) \in \mathbb{C}^N$ e $\boldsymbol{u}_{tx}(.) \in \mathbb{C}^M$ vetores de função de resposta para as antenas RX e TX, expressas como função dos ângulos de chegada (AoAs) horizontais e verticais, θ_{kl}^{rx}, ϕ_{kl}^{rx},

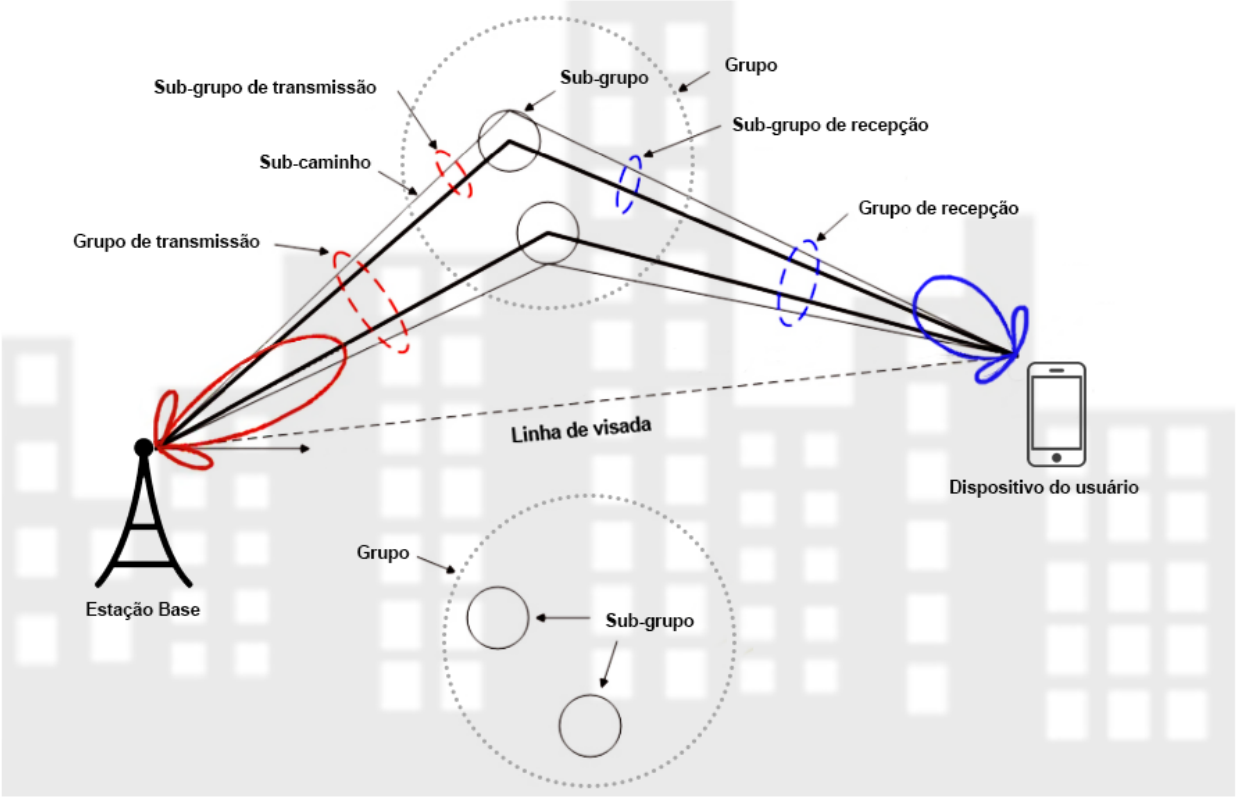

Figura 5.1: Representação do modelo de canal MIMO com grupos e sub-caminhos de propagação do sinal.

e ângulos de partida (AoDs) horizontais e verticais, θ_{kl}^{tx}, ϕ_{kl}^{tx}, ou seja, as assinaturas espaciais do receptor e do transmissor, respectivamente. O ganho complexo de desvanecimento em pequena escala é dado por [30] [5]

$$g_{kl}(t,f) = \sqrt{P_{kl}} e^{2\pi i f_{d\max} \cos(w_{kl})t - 2\pi i \tau_{kl} f}, \tag{5.2}$$

sendo P_{kl} o espalhamento de potência do sub-caminho l do grupo k, f_{dmax} é o deslocamento Doppler máximo, w_{kl} é o ângulo de chegada do sub-caminho l do grupo k relativo à direção do movimento, τ_{kl} é o espalhamento do atraso do sub-caminho l do grupo k, e f é a frequência da onda portadora do sinal.

O modelo de perda de propagação apresentado por Akdeniz *et al.* [5] considera três estados de enlace: com linha de visada (LOS, do inglês *line-of-sight*), sem linha de visada (NLOS, do inglês *non-line-of-sight*) e falha do enlace. Na condição de falha é assumido que não há enlace entre a BS e o UE, e, portanto, a perda de propagação é assumida infinita. O estado de falha acontece quando, por exemplo, a distância entre o transmissor e o receptor é maior do que 200 m e fica impossível detectar o sinal com potência de transmissão entre 20 e 30 dBm. A falha da transmissão acontece, provavelmente, devido a obstáculos ambientais que obstruem todos os caminhos até o receptor, seja por reflexão ou por espalhamento. Essa falha do enlace é uma das maiores diferenças entre as frequências de microondas/UHF tradicionais e as frequências mmWave.

A perda de propagação é função do estado do enlace, e uma função de distribuição de proba-

bilidade é assumida para cada um dos três estados. As funções de distribuição de probabilidade para os três estados são:

$$P_{out}(d) = \max(0, 1 - e^{-a_{out}d+b_{out}}) \tag{5.3}$$

$$P_{LOS}(d) = (1 - P_{out}(d))e^{-a_{los}d} \tag{5.4}$$

$$P_{NLOS}(d) = 1 - P_{out}(d) - P_{LOS}(d) \tag{5.5}$$

onde d é a distância entre o transmissor e o receptor, e os parâmetros a_{los}, a_{out} e b_{out} são parâmetros ajustados com base na estimativa de máxima verossimilhança nas medições em [32, 33], em que $1/a_{los} = 67,1$ m, $1/a_{out} = 30,0$ m e $b_{out} = 5,2$.

Dessa forma, para cada enlace, o estado é determinado da seguinte forma:

- Baseado na distância entre o UE e a BS, a probabilidade do enlace estar em cada um dos três estados (P_{LOS}, P_{NLOS} e P_{out}) é determinada;
- Uma variável aleatória uniforme (P_{REF}) entre 0 e 1 é utilizada como um valor referência para ser comparada com a probabilidade associada com cada estado do enlace;
- Se $P_{REF} \leq P_{LOS}$, o estado LOS é escolhido,
  se $P_{LOS} < P_{REF} \leq P_{LOS} + P_{NLOS}$, o estado NLOS é escolhido,
  caso contrário, o estado de falha é escolhido.

A perda de propagação (expressa em dB) é dada por

$$PL(d) = \alpha + 10\beta \log_{10}(d) + \xi, \tag{5.6}$$

onde d é a distância (em metros), e α e β são parâmetros estimados obtidos a partir do ajuste linear das perdas de propagação obtidas como função da distância BS-UE nas medições realizadas em [5] para enlaces NLOS. Para enlaces LOS, os parâmetros α e β são obtidos a partir do Modelo de Friis [34]. $\xi \sim N(0, \sigma^2)$ é uma variável aleatória que considera os efeitos do sombreamento, onde σ^2 é a variância log-normal do sombreamento. Os valores desses parâmetros são descritos na Tabela 5.1, para os estados NLOS e LOS.

Tabela 5.1: Valores dos Parâmetros de Larga Escala

Estado	Valor de α	Valor de β	Valor de σ
NLOS	72,0	2,92	8,7 dB
LOS	61,4	2	5,8 dB

Na camada MAC do módulo mmWave, o TDMA (do inglês *Time Division Multiple Access*) é utilizado como esquema de acesso múltiplo, devido ao fato da conformação de feixe analógica

ser assumida nesse modelo. Um TTI (do inglês *Transmissior Time Interval*) variável foi implementado nesse módulo, permitindo que o tamanho do *slot* varie de acordo com o comprimento do pacote ou bloco de transporte (TB, do inglês *Transport Block*) a ser transmitido. Na camada física foi implementado uma estrutura de quadro e sub-quadro baseado no TDD (do inglês *Time Division Duplex*), que toma vantagem da reciprocidade do canal para a estimação do canal.

O módulo mmWave do ns-3 é baseado no módulo Lena do ns-3 para a tecnologia LTE [35]. Esse módulo inclui um modelo de erro para pacotes de dados de acordo com o padrão LSM (do inglês *Link-to-system Mapping*). Utilizando o LSM e o MIESM (do inglês *Mutual Information Based Effective SINR*), o receptor computa a probabilidade de erro para cada bloco de transporte e determina se o pacote pode ser decodificado ou não. O TB pode ser composto de vários blocos de código (CB, do inglês *Code Block*) e seu tamanho depende da capacidade do canal. A probabilidade de erro de bloco (BLER) de cada CB depende do seu tamanho e do esquema de modulação e codificação (MCS, do inglês *modulation and coding scheme*) associado [30]:

$$C_{BLER,i}(\gamma_i) = \frac{1}{2}\left[1 - erf\left(\frac{\gamma_i - b_{C_{SIZE},MCS}}{\sqrt{2}c_{C_{SIZE},MCS}}\right)\right], \tag{5.7}$$

sendo $b_{C_{SIZE},MCS}$ e $c_{C_{SIZE},MCS}$ a média e o desvio padrão da Função de Distribuição Acumulativa Gaussiana, respectivamente, e erf a função de erro de Gauss. γ_i é a informação mútua média por bit codificado (MMIB, do inglês *mean mutual information per coded bit*) do bloco de código i dada por [36]

$$\gamma_i = \frac{1}{R}\sum_{r=1}^{R} I_m(SNR_r), \tag{5.8}$$

onde R é a quantidade de blocos de recurso (RB) alocados para o usuário sendo r o índice do RB alocado, m é o esquema de modulação adotado e SNR_r é o valor da SNR instantânea associada ao RB de índice r. A função $I_m(.)$ é a função de informação mútua associada ao esquema de modulação m. O esquema de modulação adaptativa utilizado pode ser QPSK, 16-QAM ou 64-QAM. Cada um desses esquemas possui uma função de informação mútua associada, de acordo com as aproximações numéricas feitas em [36].

A partir da $C_{BLER,i}$ calculada a partir da equação (5.7), a taxa de erro de bloco de transporte é calculada como [30]:

$$T_{BLER} = 1 - \prod_{i=1}^{C}(1 - C_{BLER,i}(\gamma_i)). \tag{5.9}$$

Em caso de falha, a camada física não encaminha o pacote para as camadas superiores e, ao mesmo tempo, ativa o processo de retransmissão para os pacotes de dados. Em caso de falha dos pacotes de controle do acesso inicial, o acesso inicial não é completado, e assim o dispositivo do usuário não consegue se conectar à rede. Por exemplo, se o preâmbulo RACH não for recebido com sucesso na estação base, a estação base não conseguirá identificar a requisição de acesso

desse dispositivo.

5.2 PHY-CC DIRECIONAL COM CONFORMAÇÃO DE FEIXE

Nós consideramos um PHY-CC direcional usando a conformação de feixe analógica. É assumido que os vetores de conformação de feixe são conhecidos previamente pelo transmissor e receptor. No modelo mmWave do ns-3 [37], as matrizes de canal e os vetores ótimos de conformação de feixe são pré-gerados no MATLAB para reduzir a sobrecarga computacional do ns-3. O módulo inclui 100 instâncias de matrizes de canal tanto para a BS quanto para o UE, assim como os vetores de conformação de feixe. Assim, o canal é atualizado periodicamente selecionando aleatoriamente uma das 100 combinações de matrizes de canal e vetores de conformação de feixe após um intervalo de tempo.

O ganho de conformação de feixe do transmissor i para o receptor j é dado por

$$G(t,f)_{BFi,j} = |\boldsymbol{w}^*_{rx_{i,j}} \boldsymbol{H}(t,f)_{i,j} \boldsymbol{w}_{tx_{i,j}}|^2, \tag{5.10}$$

sendo $\boldsymbol{H}(t,f)_{i,j}$ a matriz de canal $N \times M$ do ij-ésimo enlace, $\boldsymbol{w}_{tx_{i,j}}$ o vetor de conformação de feixe $M \times 1$ do transmissor i, quando este está transmitindo para o receptor j e $\boldsymbol{w}_{rx_{i,j}}$ o vetor de conformação de feixe $N \times 1$ do receptor j, quando este está recebendo do transmissor i.

A relação sinal-ruído pode ser calculada como

$$SNR_{i,j} = \frac{\frac{P_{tx_{i,j}}}{PL_{i,j}} G_{BFi,j}}{BW \times N_0}, \tag{5.11}$$

onde $P_{tx_{i,j}}$ é a potência transmitida da BS_i, $PL_{i,j}$ é a perda de propagação entre a BS_i e o UE_j, calculada segundo a equação (5.6), BW é a largura de banda e N_0 é a densidade espectral de potência do ruído. Essa SNR é utilizada no cálculo da informação mútua necessária para calcular a BLER de cada bloco de código (CB) na equação (5.7), para que posteriormente seja calculado a BLER de cada bloco de transporte (TB) na equação (5.9), utilizada pelo modelo de erro para decidir se o pacote será descartado ou encaminhado para a camada MAC.

5.3 PHY-CC OMNIDIRECIONAL COM O ESQUEMA ALAMOUTI

Finalmente, apresentamos como o Esquema de Alamouti é implementado no simulador de redes ns-3. Dado o fato de que o ns-3 não trata simulações símbolo-a-símbolo, o Esquema de Alamouti foi implementado com foco no seu efeito geral na SNR recebida [29].

Dada a matriz de canal $\mathbf{H}(t,f)$ de dimensões $N \times M$ definida em (5.1), o ganho de diversidade de transmissão provido pelo Esquema de Alamouti é obtido através da soma dos quadrados dos

elementos h_{nm} da matriz $\mathbf{H}(t, f)$, considerando que 2 antenas são utilizadas na transmissão e N antenas são utilizadas na recepção. No *downlink* duas antenas são utilizadas na BS para realizar a transmissão e N antenas são utilizadas para a recepção no UE. Já no *uplink*, duas antenas são utilizadas no UE para realizar a transmissão e N antenas são utilizadas para a recepção na BS.

Consideramos como exemplo o caso em que a BS possui 4 antenas e o UE possui 4 antenas, em que 2 antenas são utilizadas para a transmissão e 4 antenas são utilizadas para a recepção. Nesse caso, a matriz de canal é dada por

$$\mathbf{H}(t, f) = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \\ h_{31} & h_{32} \\ h_{41} & h_{42} \end{bmatrix} \quad . \tag{5.12}$$

A soma dos quadrados de todos os elementos da matriz $\mathbf{H}(t, f)$ provê o ganho de diversidade de transmissão do Esquema de Alamouti implementado no ns-3. Essa soma pode ser representada pela norma de Frobenius, definida por

$$\|\boldsymbol{H}\|_F^2 = \sum_n \sum_m |h_{nm}|^2, \tag{5.13}$$

sendo h_{nm} o ganho complexo do canal entre a m-ésima antena transmissora e a n-ésima antena receptora.

Então, para o caso de 2 antenas transmissoras e N antenas receptoras, a SNR recebida pode ser calculada, de forma semelhante à equação (4.13), como

$$SNR_{i,j} = \frac{\frac{P_{tx_{i,j}}}{PL_{i,j}} \frac{1}{2} \|\boldsymbol{H}\|_{F_{i,j}}^2}{BW \times N_0}. \tag{5.14}$$

A BLER do bloco de código (C_{BLER}) descrita na equação (5.7) é calculada em função da informação mútua média por bit codificado γ_i, que depende da SNR recebida calculada a partir da equação (5.14). A C_{BLER} é utilizada no cálculo da BLER do bloco de transporte (T_{BLER}), descrita na equação (5.9). A T_{BLER} define se o pacote recebido será descartado ou encaminhado para a camada MAC. Dessa forma, a implementação do Esquema de Alamouti tem o objetivo de aumentar a SNR recebida e assim aumentar a probabilidade de sucesso na troca dos pacotes de controle necessários na fase de acesso aleatório, permitindo que o UE tenha uma maior probabilidade de conseguir se conectar à rede, se comparada com uma transmissão omnidirecional SISO sem o ganho de diversidade de transmissão.

5.4 ALTERAÇÕES NO CÓDIGO DO MÓDULO MMWAVE DO NS-3

É importante mencionar que o modelo do Mezzavilla [30] assume condições de canal perfeitas para o estabelecimento da conexão do UE à rede, isto é, todos os quadros de controle do acesso aleatório são recebidos com sucesso durante a conexão do UE, não importa a distância entre o transmissor e o receptor. Consequentemente, devido ao nosso trabalho ser focado na conexão do UE sobre o canal de controle, nós modificamos o código fonte do ns-3 para permitir a ocorrência de erros durante a associação do nó na célula.

As principais classes referentes à camada física e ao canal de controle do módulo mmWave no ns-3 foram alteradas para que o modelo de erro fosse aplicado também às mensagens de controle do acesso aleatório. O modelo de erro é chamado para calcular a BLER a partir da SNR recebida e decidir se o pacote deve ser descartado pelo receptor. Assim, quando o UE envia o preâmbulo RACH para a BS a partir de uma distância muito grande, o preâmbulo não é detectado pela BS, devido à baixa SNR recebida pela BS e consequente BLER obtida, resultando em blocos de transporte corrompidos, não sendo recebidos com sucesso pela BS. O mesmo acontece com as outras mensagens de controle trocadas no acesso aleatório, tanto no *downlink* quanto no *uplink*. Dessa forma, o UE só consegue estabelecer sua conexão na rede para transmitir os pacotes de dados se primeiro as quatro mensagens do acesso aleatório forem recebidas com sucesso.

Foram necessárias alterações no código também para implementar o ganho de diversidade provido pelo Esquema de Alamouti através dos ganhos complexos de canal entre as antenas, conforme descrito na Seção 5.3. Foi necessário associar o ganho de diversidade à transmissão de pacotes de controle e o ganho de conformação de feixe à transmissão dos pacotes de dados. Essa implementação foi feita a partir da opção do modo omnidirecional de transmissão das antenas, incluindo algumas customizações.

Para implementar o modelo de erro também para os pacotes de controle e implementar o Esquema de Alamouti, as seguintes classes e respectivas funções foram alteradas no módulo mmWave desenvolvido para o ns-3:

- *MmWaveSpectrumPhy*: implementação da camada física, contendo a transmissão e recepção de pacotes de controle e dados. Nessa classe o valor da SNR recebida é atualizado e o modelo de erro é chamado para determinar a TBLER e decidir se o pacote será encaminhado para a camada MAC ou não. A função *EndRxCtrl*() foi alterada para chamar a função *GetTbDecodificationStats*() da classe *MmWaveMiErrorModel*, que contém o modelo de erro, a fim de determinar se os blocos de transporte foram corrompidos. Outras funções também foram alteradas para transmissão e recepção dos pacotes de controle: *StarRx*(), *StartRxCtrl*(), *EndRxCtrl*() e *StartTxControlFrames()*.
- *MmWaveEnbPhy* e *MmWaveUePhy*: implementação da camada física para a estação base e para o dispositivo do usuário, respectivamente. Essa classe manipula a transmissão e recepção de sinais e simula o início e o fim da transmissão de quadros, sub-quadros e slots. A função *SendCtrlChannels*() foi alterada para utilizar o modo de transmissão omnidirecional

da antena para os pacotes de controle.

- *MmWaveBeamforming*: determina a direção da transmissão (direcional ou omnidirecional) para pacotes de dados e controle e calcula os ganhos de conformação de feixe. Nessa classe os vetores de conformação de feixe e assinaturas espaciais do transmissor e receptor são carregados. Foram incluídas as funções *GetChannelGainVectorOmni()* e *GetChannelGainVectorOmniDownlink()* para calcular o ganho de diversidade proporcionado pelo Esquema de Alamouti. A função *DoCalcRxPowerSpectralDensity* foi alterada para considerar o ganho do Esquema de Alamouti na transmissão omnidirecional para os pacotes de controle e o ganho de conformação de feixe na transmissão direcional para os pacotes de dados.

Todas as alterações realizadas no código fonte do módulo mmWave do ns-3 estão registradas no apêndice dessa dissertação.

6 AVALIAÇÃO DE DESEMPENHO

Neste capítulo nós apresentamos os cenários de simulação e os resultados obtidos utilizando o simulador ns-3 a partir do módulo [30] apresentando previamente, com a inclusão do Esquema de Alamouti e as modificações para permitir erros no canal de controle durante a conexão do UE.

6.1 CENÁRIOS DE SIMULAÇÃO

O objetivo do nosso estudo é investigar a eficácia de usar o Esquema de Alamouti como um meio para fornecer cobertura similar à cobertura fornecida pela comunicação direcional com ganho de conformação de feixe sobre o canal de controle mmWave, durante a fase de conexão do UE na célula. Então, se obtiver sucesso, essa técnica pode evitar a sobrecarga requerida pelas técnicas de busca feixe ou soluções fora da banda para realizar o acesso inicial, permitindo uma solução dentro da banda de forma omnidirecional e com atraso reduzido.

Nosso trabalho é focado em estimar a probabilidade empírica de sucesso da conexão do UE na célula como função da sua distância até a BS. Para calcular isso, nós analisamos o sucesso da recepção das primeiras mensagens de controle que são trocadas entre o UE e a BS no acesso inicial, durante a fase de acesso aleatório, a fim de garantir a associação do UE na célula. A conexão do UE na rede só é executada quando as quatro mensagens do acesso aleatório são recebidas com sucesso pela BS (*uplink*) e pelo UE (*downlink*). Nós consideramos uma única célula *outdoor* operando em 28 GHz, e a associação de um único UE sem interferência inter- ou intra-celular. Além disso, nós avaliamos a SNR média do *uplink* para os quadros recebidos como função da distância entre UE e BS, a fim de investigar a relação entre a SNR recebida e a probabilidade de conexão do UE em um canal de 28 GHz. A SNR é medida apenas no *uplink* devido ao fato de que se o preâmbulo RACH não for recebido com sucesso na BS (canal *uplink*), a mensagem RAR não é enviada a partir da BS para o UE (canal *downlink*), não acontecendo transmissão no sentido *downlink* nesse caso. Portanto, a SNR média medida no *uplink* abrange todos os pacotes de controle enviados do UE para a BS na fase de acesso aleatório, isto é, o preâmbulo RACH e a mensagem de requisição de conexão. Essa SNR calculada não inclui a mensagem RAR e a mensagem de resolução de contenção, que são enviadas sobre o canal *downlink*.

No que diz respeito à configuração das antenas, nós consideramos primeiro a transmissão SISO (1×1) omnidirecional sem nenhum ganho de antena a fim de entender as limitações do canal de controle mmWave e os ganhos alcançáveis usando conformação de feixe e o Esquema de Alamouti. Para o Esquema de Alamouti, nós estudamos diferentes configurações de antenas: 2×1, 2×2, 2×4, 2×8, 2×16 e 2×32. Exceto pelo caso 2×32, todas as configurações de antenas são simétricas, o que significa que a mesma configuração $2 \times N$ é usada tanto no canal *uplink* quanto no canal *downlink*. Mas, devido ao modelo de simulação adotado [30] suportar um

número máximo de 64 antenas na BS e um número máximo de 16 antenas no UE, o caso "2×32" na verdade significa que, no canal *uplink*, nós usamos a configuração 2×32, enquanto que no canal *downlink* nós usamos a configuração 2×16 (número máximo de antenas disponíveis no UE).

Para fins de comparação, nós também estimamos a probabilidade de conexão do UE e a SNR *uplink* para o caso do PHY-CC direcional com ganho de conformação de feixe. Nesse caso, é assumido que tanto o UE quanto a BS têm a estimação de canal perfeita e o conhecimento instantâneo dos vetores de conformação de feixe (isto é, a busca de feixes não é realizada), como fornecido pelo modelo [30]. Para o cálculo da probabilidade empírica de conexão, nós medimos o número de associações do UE realizadas com sucesso dentro de 100 tentativas para cada distância específica entre UE e BS. Então, nós calculamos a frequência relativa de associações realizadas com sucesso variando a distância UE-BS em passos de 10 metros, a partir de 10 até 130 metros.

A potência de transmissão da BS é de 30 dBm no *downlink* em todos os cenários e utilizamos cenários com 20 e 30 dBm para a potência de transmissão do UE no *uplink*. No cenário 1, consideramos a potência de transmissão do UE como 20 dBm, valor utilizado em trabalhos como [5, 38]. Já no cenário 2, apresentamos os resultados obtidos aumentando a potência de transmissão do UE para 30 dBm como utilizado no trabalho [13]. No cenário 3 aumentamos a potência de transmissão do UE para 30 dBm apenas nos casos de transmissão omnidirecional (com e sem o Esquema de Alamouti), mantendo a potência de transmissão do UE em 20 dBm para a transmissão direcional. Utilizamos o valor de 5 dB para o ruído, conforme utilizado nos trabalhos [5, 30], que foram utilizados como referência para as simulações. Os outros parâmetros da camada física são definidos de acordo com os valores-padrão dados em [30]. A Tabela 6.1 resume os parâmetros específicos utilizados nas simulações.

A altura das antenas transmissoras da BS foi definida de acordo com [5], onde a altura de 10 m representa um transmissor em cima do telhado de um prédio de 3 a 4 andares, por exemplo. Como a arquitetura das redes mmWave deve ser baseada em pequenas células, o número de estações base deve ser maior e elas devem estar mais próximas dos usuários, sendo alocadas com altura menor do que nas redes 4G, em que a altura da estação base costuma ser próxima a 30 m. Já para o UE, foi definida a altura de 1,5 m por representar a altura em que um dispositivo móvel ficaria sendo utilizado por uma pessoa de estatura média. A quantidade de antenas na BS (64 antenas) e no UE (16 antenas) foi definida de acordo com a capacidade do modelo mmWave do ns-3 [30] e de acordo com a configuração sugerida em trabalhos recentes sobre mmWave [5, 38]. A Figura 6.1 representa a configuração de antenas utilizada para a estação base e para o dispositivo do usuário. A frequência da portadora de 28 GHz foi adotada devido ao fato de ser uma das principais frequências candidatas para redes mmWave, por ser uma frequência relativamente baixa dentro da banda mmWave, e por já ter sido utilizada para sistemas LMDS (do inglês *Local Multipoint Distribution Systems*) [5]. Além disso, o modelo de propagação utilizado para as simulações foi baseado em medições feitas em ambientes urbanos na frequência 28 GHz, sendo implementado no módulo mmWave do simulador ns-3. O valores da Tabela 5.1 foram utilizados para os parâmetros de larga de escala para os estados de enlace LOS e NLOS.

Tabela 6.1: Parâmetros de simulação da camada física mmWave

Descrição do Parâmetro	**Valor**
Potência de transmissão da BS	30 dBm
Potência de transmissão do UE (omnidirecional)	20 (cenário 1) e 30 dBm (cenários 2 e 3)
Potência de transmissão do UE (direcional)	20 (cenários 1 e 3) e 30 dBm (cenário 2)
Altura da estação base	10 m
Altura do equipamento do usuário	1,5 m
Número máximo de antenas na estação base	64
Número máximo de antenas no dispositivo móvel	16
Ruído	5 dB
Frequência da portadora	28 GHz

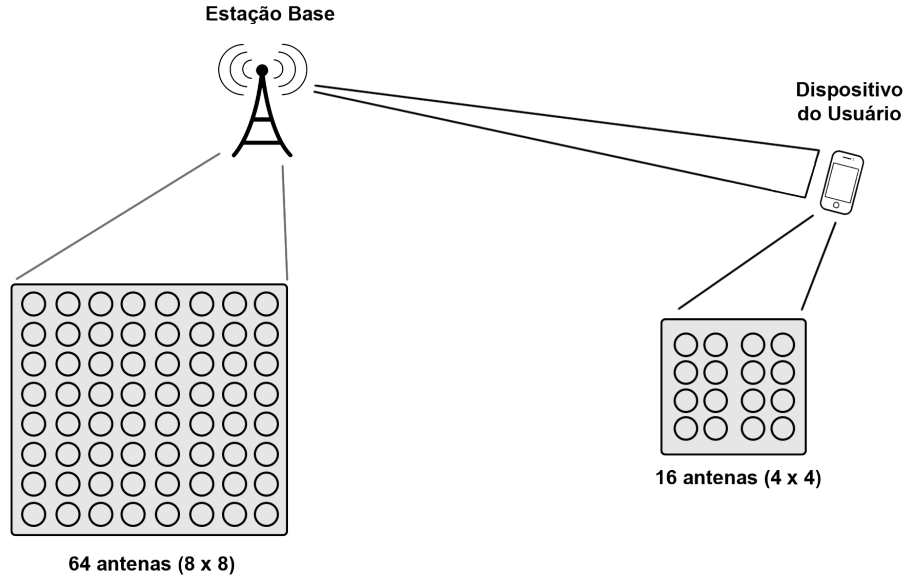

Figura 6.1: Cenário de simulação com um enlace de comunicação entre a BS e o UE e suas respectivas configurações de antenas.

6.2 RESULTADOS DAS SIMULAÇÕES

Nesta seção nós apresentamos os principais resultados obtidos neste trabalho para os diferentes cenários de simulação descritos na Seção 6.1.

6.2.1 Cenário 1

A Figura 6.2 descreve os resultados para a probabilidade empírica de conexão do UE na rede como função da distância entre o UE e a BS, considerando a potência de transmissão da BS como 30 dBm (*downlink*) e a potência de transmissão do UE como 20 dBm (*uplink*). Esse gráfico ilustra claramente a inviabilidade do uso de um PHY-CC SISO omnidirecional (sem diversidade de transmissão) para redes celulares mmWave. A probabilidade de conexão do UE nesse cenário SISO omnidirecional cai fortemente à medida em que a distância aumenta e torna-se menor do

que 0,5 para distâncias maiores do que 40 m. Assim, se um PHY-CC SISO omnidirecional fosse utilizado, somente os UEs bem próximos à BS poderiam estabelecer uma conexão bem sucedida na rede, comprometendo a fase de acesso inicial do UE. Dessa forma, o raio de cobertura da célula ficaria limitado à uma curta distância mesmo que a transmissão de dados alcançasse uma distância maior, pois a BS e o UE só poderão estabelecer uma comunicação direcional para transmissão de dados após o acesso inicial ser executado com sucesso pelo UE. Podemos perceber então que o PHY-CC omnidirecional sem diversidade de transmissão traz uma grande limitação de alcance em distância para a execução do acesso inicial do UE na rede.

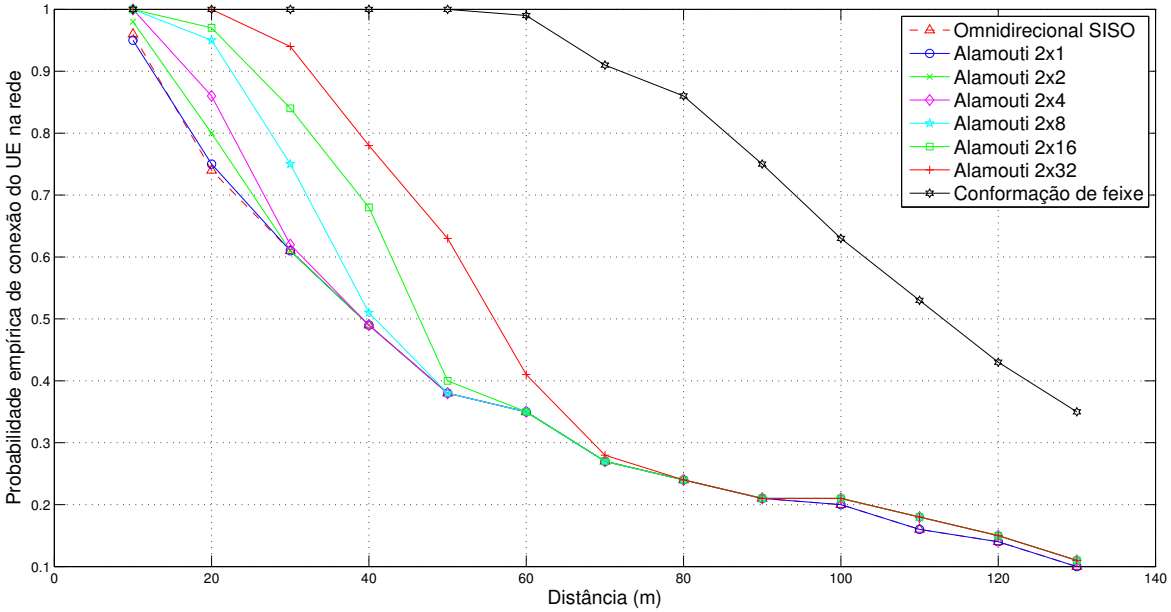

Figura 6.2: Probabilidade de conexão empírica como função da distância UE-BS para transmissão SISO omnidirecional, conformação de feixe e diferentes configurações de antena para o Esquema de Alamouti, considerando P_{tx} (UE) = 20 dBm.

Nós apresentamos os resultados de probabilidade de conexão do UE para um PHY-CC direcional com conformação de feixe a fim de ter um referencial para a análise dos resultados com um PHY-CC omnidirecional. Em contraste com o cenário PHY-CC SISO omnidirecional, o uso de um PHY-CC com conformação de feixe permite uma cobertura maior, proporcionando uma probabilidade de conexão do UE de 100% se o UE estiver em uma distância de até 50 m da BS. Essa probabilidade cai para menos do que 0,8 em distâncias acima de 90 m e fica acima de 0,5 para a distância de 110 m. Claramente, um PHY-CC SISO omnidirecional causaria uma grande incompatibilidade entre os alcances de transmissão do canal de controle e do canal de dados, já que o canal de dados deve operar de forma direcional com conformação de feixe em redes mmWave, proporcionando um maior alcance de transmissão.

Agora, considerando o Esquema de Alamouti para o PHY-CC omnidirecional, nós observamos que a probabilidade de sucesso da conexão do UE na rede em geral melhora à medida que o número de antenas de recepção aumenta, como já era esperado, pois isso é consequência dos ganhos de diversidade de transmissão do Esquema de Alamouti. Na Figura 6.2, podemos ver

que para a distância de 40 m, a probabilidade de conexão do UE é de 0,49 para o Esquema de Alamouti 2×1, 2×2 e 2×4, 0,51 para o Esquema de Alamouti 2×8, 0,68 para o Esquema de Alamouti 2×16 e finalmente 0,78 para o Esquema de Alamouti 2×32, que apresenta a melhor probabilidade devido ao maior número de antenas no receptor. Assim fica claro que a configuração de antenas 2×32 alcança o melhor desempenho, com uma probabilidade de conexão maior do que 0,5 para distâncias até 50 m. É possível perceber que o cenário com o Esquema de Alamouti 2×32 apresenta um ganho de 59% em relação ao SISO omnidirecional em uma distância de 40 m entre UE e BS.

A Figura 6.3 mostra a SNR média recebida no canal de controle *uplink* como função da distância entre o UE e a BS. Para as diferentes configurações de antena do Esquema de Alamouti, é possível perceber um ganho de aproximadamente 3 dB sempre que a quantidade de antenas de recepção é dobrada. Por exemplo, a SNR média recebida para o canal *uplink* para a distância de 100 m é de -13,76 dB para o Esquema de Alamouti 2×16 e -10,75 para o Esquema de Alamouti 2×32. Observamos também que a SNR máxima recebida para o Esquema de Alamouti 2×32 é de 26,95 dB na distância de 10 m e a SNR mínima é de -16,84 dB. Conforme mostrado na Figura 6.3, a configuração 2×32 oferece um ganho de aproximadamente 16 dB sobre a SNR recebida com uma transmissão omnidirecional SISO. É importante notar que o canal mmWave permite transmissões LOS e NLOS, dependendo das realizações específicas do canal durante as simulações. Nós podemos observar que o PHY-CC omnidirecional e o PHY-CC com o Esquema de Alamouti na configuração 2×1 apresentam praticamente a mesma SNR. Como já é conhecido, o Esquema de Alamouti não entrega os mesmos ganhos de diversidade de transmissão em canais LOS, o que explica essa diferença praticamente imperceptível entre o SISO omnidirecional e o Esquema de Alamouti 2×1.

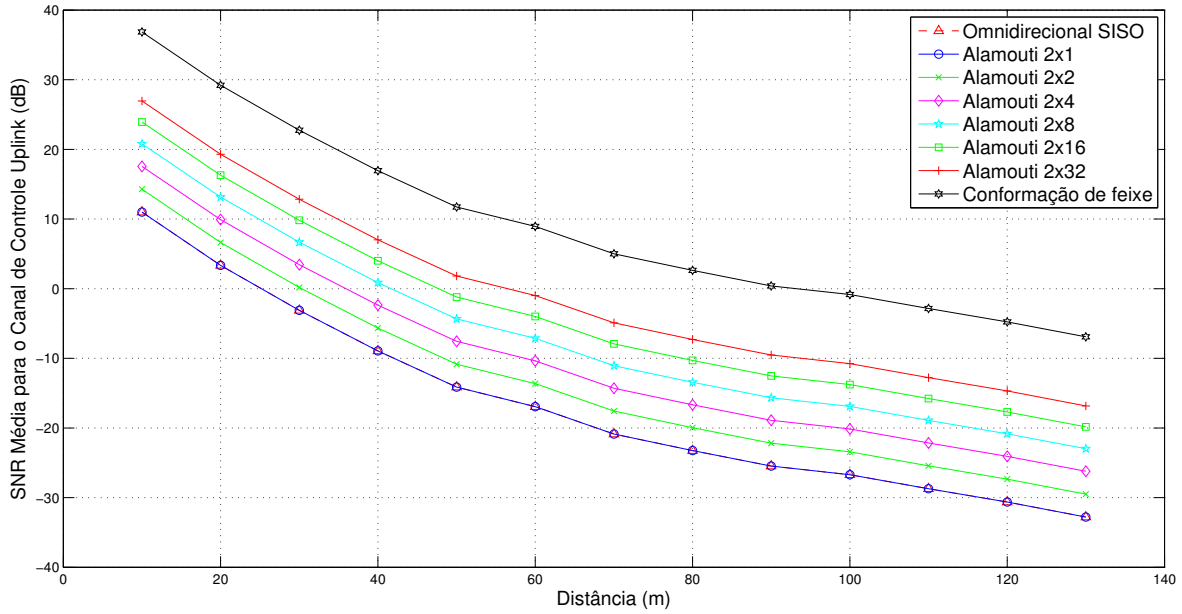

Figura 6.3: SNR média do uplink para uma transmissão SISO omnidirecional, conformação de feixe e diferentes configurações de antena para o Esquema de Alamouti, considerando P_{tx} (UE) = 20 dBm.

É possível notar através da Figura 6.3 que o PHY-CC direcional com conformação de feixe apresenta um ganho de aproximadamente 10 dB em relação ao PHY-CC com o Esquema de Alamouti na configuração 2×32. No caso direcional, o valor da SNR média máxima observada é de 36,86 dB para a distância de 10 m, enquanto que a SNR mínima observada é de -6,92 dB para a distância de 130 m. Se considerarmos os casos com 100% de probabilidade de conexão, a SNR mínima é 11,73 dB para o caso de conformação de feixe com a distância de 50 m. Considerando o PHY-CC direcional com conformação de feixe como um referencial, seria necessário um ganho de 10 dB para a configuração do Esquema de Alamouti 2×32 obter o mesmo desempenho do PHY-CC direcional. É importante lembrar que em nossas simulações consideramos que os vetores de conformação de feixe já são conhecidos pelo transmissor e receptor, pois o PHY-CC direcional só poderia ser realizado no acesso inicial após o procedimento de alinhamento dos feixes. Utilizamos o PHY-CC direcional com conformação de feixe como referência devido ao fato dele ser utilizado no canal de dados. Assim, apresentamos uma forma de realizar um PHY-CC omnidirecional que mais se aproxime do desempenho obtido com o PHY-CC direcional a fim de obter alcances de transmissão semelhantes no canal de controle e de dados.

6.2.2 Cenário 2

No cenário 2, consideramos a potência de transmissão do UE como 30 dBm (*uplink*). A Figura 6.4 descreve os resultados para a probabilidade empírica de conexão do UE na rede como função da distância entre o UE e a BS. É possível perceber que com o aumento da potência de transmissão no UE, surgiram diferenças mais significativas na probabilidade de conexão do UE para as diferentes configurações do Esquema de Alamouti. Como esperado, a configuração do Esquema de Alamouti 2×32 apresentou a maior probabilidade de conexão do UE na rede, já que possui o maior número de antenas no receptor em relação às outras configurações do Esquema de Alamouti. A probabilidade de conexão do UE para o caso 2×32 foi de 100% para as distâncias de até 30 m entre a BS e o UE. Essa probabilidade foi maior que 0,5 para até 80 m de distância. A configuração do Esquema de Alamouti 2×16 também apresentou uma probabilidade de conexão razoável, com probabilidade de 100% para distâncias de até 30 m e probabilidade de 0,64 para a distância de 70 m, enquanto que a probabilidade para a configuração 2×32 foi de 0,75.

Notamos que também nesse cenário temos uma probabilidade de conexão do UE bem maior para o PHY-CC omnidirecional com o Esquema de Alamouti do que o PHY-CC SISO omnidirecional sem ganho de diversidade de transmissão. Por exemplo, a probabilidade de conexão do UE para a distância de 80 m é de 0,24 para o PHY-CC SISO omnidirecional, enquanto que para a configuração do Esquema de Alamouti 2×32 é de 0,56, o que representa um ganho de 133% sobre o PHY-CC SISO omnidirecional. Já a transmissão direcional com conformação de feixe apresenta uma probabilidade de conexão de 100% para distâncias até 110 m, apresentando uma melhoria de desempenho significativa com o aumento da potência de transmissão no UE. Novamente a configuração do Esquema de Alamouti que mais se aproxima ao desempenho do PHY-CC direcional com conformação de feixe é a configuração 2×32, com probabilidade acima

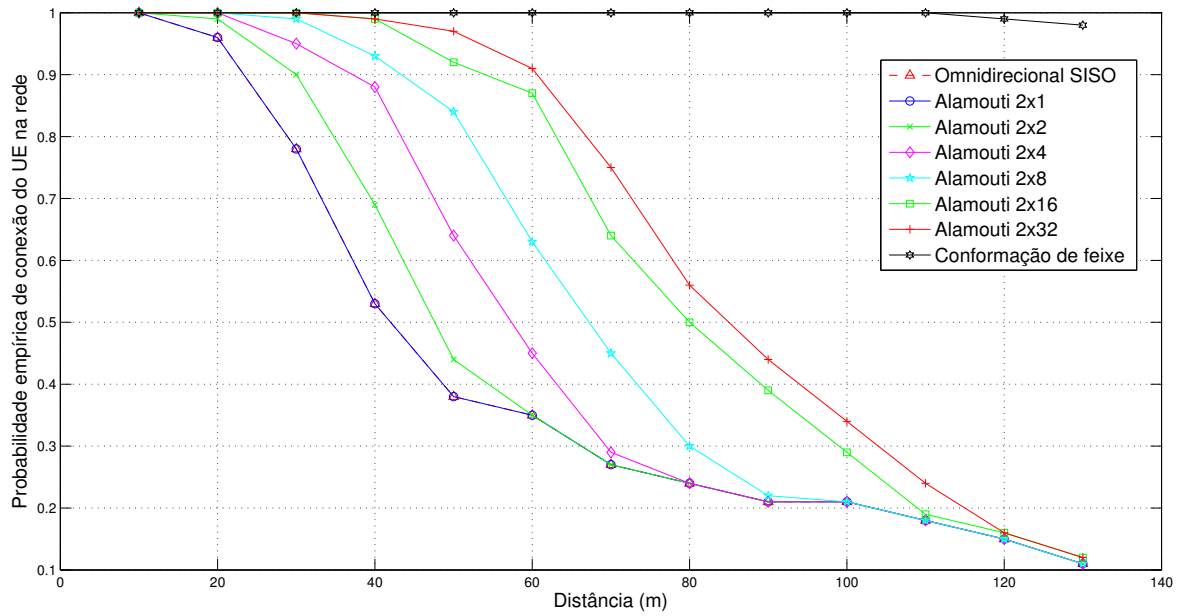

Figura 6.4: Probabilidade de conexão empírica como função da distância UE-BS para transmissão SISO omnidirecional, conformação de feixe e diferentes configurações de antena para o Esquema de Alamouti, considerando P_{tx} (UE) = 30 dBm.

de 0,9 para a distância de até 50 m.

A Figura 6.5 mostra a SNR média recebida no canal de controle *uplink* como função da distância entre o UE e a BS. É possível notar que a SNR recebida para o canal *uplink* no PHY-CC SISO omnidirecional é de -13,25 dB para a distância de 80 m, enquanto que para o PHY-CC com o Esquema de Alamouti 2×32 é de aproximadamente 2,5 dB na mesma distância, o que representa um ganho de quase 16 dB sobre o PHY-CC SISO omnidirecional. Para uma configuração do Esquema de Alamouti 2×2, que é uma configuração mais básica com menos antenas no receptor, temos um ganho de aproximadamente 3,3 dB sobre o PHY-CC SISO omnidirecional considerando a mesma distância de 80 m.

É possível perceber que, para a configuração do Esquema de Alamouti 2×32, a SNR máxima recebida no *uplink* é 36,73 dB na distância de 10 m e a SNR mínima é de -7,05 dB para a distância de 130 m. Já para o PHY-CC direcional, a SNR máxima recebida no canal *uplink* é de 46,86 dB para a distância de 10 m e de 3,08 dB para a distância de 130 m. Nessas condições, seria necessário um ganho adicional de 10 dB para o PHY-CC omnidirecional na configuração 2×32 alcançar o mesmo desempenho do PHY-CC direcional com conformação de feixe. Notamos também que a probabilidade de conexão do UE na Figura 6.4 para o caso Alamouti 2×32 não reflete totalmente o ganho na SNR apresentado na Figura 6.5 devido à medição da probabilidade de conexão levar em conta todas as fases do acesso inicial, incluindo os sinais *downlink* e *uplink*. Já a SNR medida na Figura 6.5 só considera o canal *uplink*, já que só acontece transmissão no canal *downlink* após a transmissão ser feita com sucesso no canal *uplink*, durante o envio do preâmbulo RACH do UE para a BS. Como a transmissão no canal *downlink* é feita na configuração 2×16 devido ao número máximo de antenas permitidas no UE ser 16 antenas, a diferença entre a probabilidade

de conexão da configuração Alamouti 2×32 e 2×16 não é tão significativa na Figura 6.4, se comparado às outras configurações de antenas.

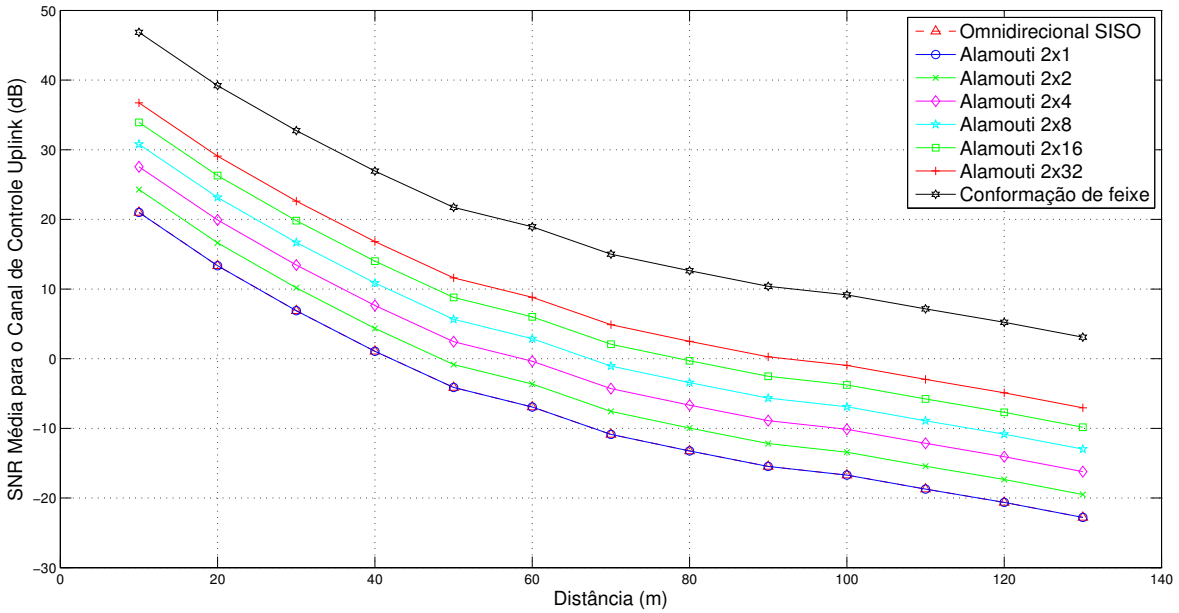

Figura 6.5: SNR média do uplink para uma transmissão SISO omnidirecional, conformação de feixe e diferentes configurações de antena para o Esquema de Alamouti, considerando P_{tx} (UE) = 30 dBm.

6.2.3 Cenário 3

Consideramos um aumento de 10 dB na potência de transmissão do dispositivo do usuário apenas durante a fase de acesso inicial, a fim de garantir a conexão do UE na rede utilizando um PHY-CC omnidirecional com o Esquema de Alamouti, considerando a potência de transmissão do UE como 30 dBm para as configurações do PHY-CC omnidirecional e 20 dBm para o PHY-CC direcional. Dessa forma o dispositivo poderia aumentar a sua potência de transmissão apenas em um curto período de tempo demandado pela fase de acesso aleatório e após a conexão do UE na rede, a transmissão de dados seria realizada de forma direcional com a conformação de feixe sem a necessidade de acréscimo na potência de transmissão. Considerando esse cenário, a Figura 6.6 apresenta a probabilidade de conexão do UE na rede. Podemos observar que a probabilidade de conexão para a configuração do Esquema de Alamouti 2×32 se aproxima mais da probabilidade obtida para o PHY-CC direcional com conformação de feixe. Até 60 m de distância, as probabilidades de conexão são bem semelhantes, acima de 0,9. A partir de 60 m de distância, a diferença entre essas probabilidades começa ser mais significativa. Para a distância de 70 m, o Esquema de Alamouti 2×32 alcança uma probabilidade de 0,75, enquanto o PHY-CC direcional alcança a probabilidade de 0,91.

Podemos notar que o Esquema de Alamouti alcança um desempenho similar ao PHY-CC com conformação de feixe, usando uma transmissão omnidirecional e sem a necessidade de algoritmos de busca de feixe durante o estabelecimento da conexão. É importante esclarecer que a configuração 2×32 na verdade significa que somente 16 antenas são usadas no canal *downlink* no UE, pois essa é a configuração máxima de antenas presente no UE no módulo mmWave utilizado nas simulações, o que significa que o desempenho poderia ser melhorado se condições simétricas fossem usadas nos dois sentidos. Isso fica mais claro quando nós analisamos a SNR média no canal *uplink* na Figura 6.7.

Os resultados mostrados na Figura 6.7 indicam que a SNR recebida no canal *uplink* para o PHY-CC com conformação de feixe e para o caso Alamouti 2×32 foram bem semelhantes. A SNR máxima observada para o PHY-CC com conformação de feixe foi de 36,86 dB para a distância de 10 m e a SNR mínima foi de -6,92 dB para a distância de 130 m, enquanto que no caso do Alamouti 2×32 a SNR máxima foi de 36,73 dB para a distância de 10 m e -7,05 dB para a distância de 130 m. Isso indica que, se a mesma configuração 2×32 fosse usada no canal *downlink*, o Esquema de Alamouti iria alcançar a mesma probabilidade de conexão que o PHY-CC com conformação de feixe na Figura 6.6, já que no canal *downlink* temos a configuração Alamouti 2×16 ao invés de 2×32 como no *uplink*.

Se nós considerarmos os casos com 100% de sucesso na conexão do UE, a SNR mínima observada para o PHY-CC com conformação de feixe é 11,73 dB para uma distância de 50 m entre a BS e o UE. Para o Esquema de Alamouti, a SNR mínima para os casos de 100% de sucesso na conexão do UE é de 22,62 dB na distância de 30 m entre a BS e o UE. Podemos observar também que tanto o PHY-CC com conformação de feixe quanto o PHY-CC com o Esquema de Alamouti 2×32 apresentam um ganho de aproximadamente 16 dB sobre o PHY-CC SISO omnidirecional.

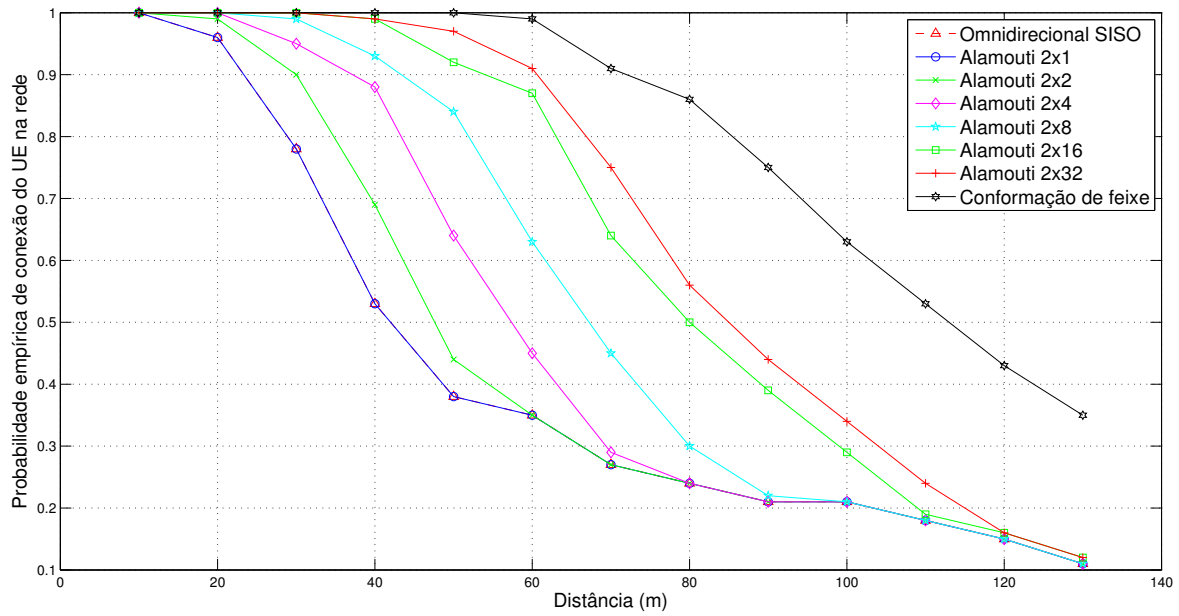

Figura 6.6: Probabilidade de conexão empírica como função da distância UE-BS para transmissão SISO omnidirecional, conformação de feixe e diferentes configurações de antena para o Esquema de Alamouti, considerando P_{tx} (UE) = 30 dBm para o PHY-CC omnidirecional e P_{tx} (UE) = 20 dBm para o PHY-CC direcional com conformação de feixe.

Os resultados obtidos mostram que um PHY-CC com diversidade de transmissão pode prover um maior alcance do que o PHY-CC SISO omnidirecional, requerendo um processamento simples e sem informação do estado do canal pelo transmissor. Observamos também que com um ganho 10 dB, o PHY-CC omnidirecional com o Esquema de Alamouti na configuração 2×32 alcança uma SNR média do canal *uplink* praticamente igual ao PHY-CC direcional com conformação de feixe e alcançaria uma probabilidade de conexão do UE na rede também semelhante se a mesma configuração de antenas fosse utilizada na BS e no UE para prover a mesma diversidade de transmissão nos canais *downlink* e *uplink*. Esses resultados provam que um PHY-CC omnidirecional com o Esquema de Alamouti pode ser uma possível opção para o realizar o acesso inicial em redes celulares mmWave nas condições discutidas anteriormente. Dessa forma, é possível realizar o PHY-CC mmWave dentro da banda com diversidade de transmissão usando o Esquema de Alamouti, provendo cobertura similar ao PHY-CC direcional e sem a sobrecarga de busca de feixe que o PHY-CC direcional traz para o acesso inicial.

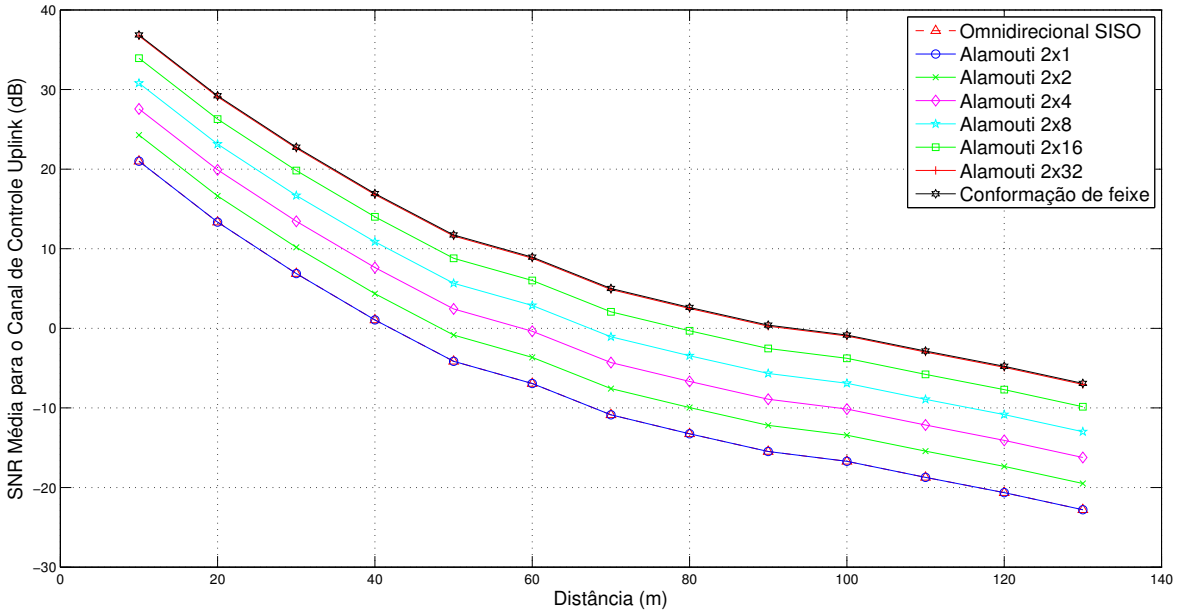

Figura 6.7: SNR média do uplink para uma transmissão SISO omnidirecional, conformação de feixe e diferentes configurações de antena para o Esquema de Alamouti, considerando P_{tx} (UE) = 30 dBm para o PHY-CC omnidirecional e P_{tx} (UE) = 20 dBm para o PHY-CC direcional com conformação de feixe.

7 CONCLUSÃO

Neste trabalho apresentamos as principais características da redes celulares mmWave e os seus desafios em relação ao acesso inicial do dispositivo do usuário na rede. Foi feita uma revisão sobre os principais fundamentos teóricos relacionados com as redes celulares mmWave e foram apresentados os principais trabalhos que contribuíram para a nossa proposta, mostrando os avanços e questões discutidas sobre o acesso inicial em redes mmWave. Mostramos então que ainda existem desafios e questões em aberto para a realização do canal de controle físico para o acesso inicial.

Entre os desafios apresentados para o acesso inicial em redes mmWave está a realização do canal de controle físico em modo direcional. Apresentamos pesquisas recentes nesse tema que abordam a questão da sobrecarga de tempo para fazer a busca de feixes em transmissões direcionais no acesso inicial, incluindo um atraso adicional para realizar o alinhamento de feixes antes de começar a fase de acesso aleatório. Discutimos que esse atraso traz prejuízo ao acesso inicial, dificultando a conexão do dispositivo móvel na rede, especialmente em situações de falha do enlace e *handover*. Foram discutidas também as vantagens e as desvantagens de utilizar um canal de controle omnidirecional sobre um canal de controle direcional para realizar o acesso inicial em redes mmWave. Apresentamos a vantagem de evitar o atraso adicional incluído pelo procedimento de alinhamento de feixes que a transmissão direcional requer. Apresentamos também a desvantagem do curto alcance da transmissão omnidirecional em redes mmWave devido à grande perda de propagação nessa banda.

Apresentamos nossa proposta de realizar o canal de controle físico para o acesso inicial em modo omnidirecional dentro da banda mmWave utilizando o esquema de diversidade de transmissão de Alamouti para aumentar a cobertura em distância da transmissão omnidirecional. Foi feita uma revisão sobre o ganho de diversidade proporcionado pelo Esquema de Alamouti, assim como seus principais benefícios para uma transmissão omnidirecional com uma grande quantidade de antenas no transmissor e no receptor, compensando a alta perda de propagação da banda mmWave através do seu ganho de diversidade de transmissão. Ressaltamos que o Esquema de Alamouti possui um processamento linear simples e não requer conhecimento do canal pelo transmissor, o que facilita sua implementação para a fase de acesso inicial em uma rede mmWave.

Descrevemos o modelo de canal mmWave utilizado, as probabilidades de estado de canal, o modelo de erro de pacotes, os ganhos de conformação de feixe e os ganhos de diversidade de transmissão via Esquema de Alamouti. Apresentamos a forma de implementação do modelo de canal e da nossa proposta no simulador de redes ns-3. Também descrevemos os nossos cenários de simulação, informando os principais parâmetros adotados para a camada física e as diferentes configurações de antenas no transmissor e no receptor.

Implementamos nossa proposta no ns-3, simulando uma rede mmWave em 28 GHz com um

PHY-CC omnidirecional com o Esquema de Alamouti em diversas configurações de antenas (2×1, 2×2, 2×4, 2×8, 2×16 e 2×32). Implementamos também o PHY-CC direcional com conformação de feixe e o PHY-CC SISO omnidirecional (sem diversidade de transmissão) a fim de comparar os desempenhos entre os diferentes modos de PHY-CC. Analisamos a probabilidade empírica de conexão do UE na rede durante o acesso inicial em função da distância entre a BS e o UE, para cada um dos modos do PHY-CC mencionados. Também analisamos a SNR média recebida no *uplink* em função da distância BS-UE para canal de controle.

No primeiro cenário, investigamos os resultados obtidos considerando a potência de transmissão de 20 dBm no UE. Foi possível perceber a inviabilidade de utilizar um PHY-CC SISO omnidirecional sem diversidade de transmissão para o acesso inicial em redes mmWave, já que proporciona pequenas probabilidades de conexão do UE até mesmo em distâncias curtas. Já o PHY-CC omnidirecional com o Esquema de Alamouti alcançou um melhor desempenho especialmente para a configuração de antenas 2×32, com uma probabilidade de conexão maior do que 0,5 para distâncias até 50 m, um ganho de de 59% em relação ao SISO omnidirecional em uma distância de 40 m entre UE e BS e um ganho de aproximadamente 16 dB sobre a SNR recebida com uma transmissão omnidirecional SISO.

Já no segundo cenário, aumentamos a potência de transmissão do UE para 30 dBm. Com isso, surgiram diferenças mais significativas na probabilidade de conexão do UE para as diferentes configurações do Esquema de Alamouti, sendo que a configuração 2×32 apresentou a maior probabilidade de conexão para o Esquema de Alamouti. Essa probabilidade foi maior do que 0,9 para até 50 m de distância entre a BS e o UE e maior que 0,5 para até 80 m de distância. A configuração do Esquema de Alamouti 2×32 apresentou a probabilidade de conexão mais próxima da probabilidade de conexão alcançada pelo modo direcional com conformação de feixe. Observando os resultados da SNR média recebida no canal *uplink*, verificamos que seria necessário um ganho adicional de 10 dB para o PHY-CC omnidirecional na configuração 2×32 alcançar o mesmo desempenho do PHY-CC direcional com conformação de feixe.

Finalmente, no Cenário 3, consideramos um aumento de 10 dB na potência de transmissão do dispositivo do usuário apenas durante a fase de acesso inicial, a fim de garantir a conexão do UE na rede utilizando um PHY-CC omnidirecional com o Esquema de Alamouti. Assim, a potência de transmissão de 30 dBm no UE seria utilizada apenas no acesso inicial durante a transmissão omnidirecional, pois a transmissão de dados poderia ser feita de forma direcional com a potência de transmissão de 20 dBm no UE. Portanto, comparamos o desempenho do PHY-CC omnidirecional com potência de transmissão do UE em 30 dBm com o desempenho do PHY-CC direcional com a potência de transmissão do UE em 20 dBm. Assumindo esse cenário, foi possível perceber que o Esquema de Alamouti na configuração 2×32 alcançou um desempenho similar ao PHY-CC com conformação de feixe, pois até 60 m de distância, as probabilidades de conexão foram bem semelhantes, acima de 0,9. O Esquema de Alamouti na configuração 2×32 alcançaria uma probabilidade de conexão do UE na rede melhor se a mesma configuração de antenas fosse utilizada na BS e no UE para prover a mesma diversidade de transmissão nos canais *downlink* e *uplink*. A SNR recebida no canal *uplink* para o PHY-CC com conformação de feixe

e para o caso Alamouti 2×32 foram bem próximas. A SNR máxima observada para o PHY-CC com conformação de feixe foi de 36,86 dB para a distância de 10 m e a SNR mínima foi de -6,92 dB para a distância de 130 m, enquanto que no caso do Alamouti 2×32 a SNR máxima foi de 36,73 dB para a distância de 10 m e -7,05 dB para a distância de 130 m.

Os resultados obtidos mostraram que o Esquema de Alamouti pode ser utilizado no acesso inicial para aumentar a cobertura em distância de uma transmissão omnidirecional no acesso aleatório em redes celulares mmWave. A configuração de antenas 2×32 com o Esquema de Alamouti foi a configuração que obteve o desempenho mais próximo do desempenho alcançado pela transmissão direcional com conformação de feixe. Foi possível perceber que seria necessário um ganho de 10 dB para o PHY-CC omnidirecional com o Esquema de Alamouti na configuração 2×32 alcançar o mesmo desempenho do PHY-CC direcional com conformação de feixe, em termos de probabilidade de conexão do UE na rede e da SNR recebida no canal de controle *uplink*. Nossos resultados mostraram que o PHY-CC omnidirecional com o Esquema de Alamouti proporciona cobertura significativamente maior em distância que o PHY-CC direcional SISO (sem diversidade de transmissão) com um processamento linear simples e com pouca complexidade computacional, sem requerer a sobrecarga de busca de feixe que o PHY-CC direcional traz para o acesso inicial, provando ser uma alternativa viável para realizar o PHY-CC para o acesso aleatório em redes celulares mmWave.

Como trabalhos futuros sugerimos que sejam analisados cenários com mais de um par de comunicação BS-UE. Seria proveitoso analisar a porcentagem de nós que conseguem realizar o acesso aleatório com sucesso, distribuídos em diferentes posições na célula. Também seria produtivo analisar um cenário com os dispositivos móveis em movimentando dentro da célula ou em operação de *handover* entre diferentes células. Outra abordagem para trabalhos futuros seria medir o atraso demandado por uma conexão do usuário à uma rede celular mmWave com PHY-CC em modo omnidirecional e em modo direcional com conformação de feixe. Também seria benéfico considerar a interferência intra e inter-celular durante o acesso aleatório em redes celulares mmWave. Esses cenários de simulação são apresentados como futuros trabalhos de forma a complementar o nosso trabalho, contribuindo para o avanço dos estudos referentes ao acesso inicial em redes celulares mmWave.

REFERÊNCIAS BIBLIOGRÁFICAS

1 SUN, S.; RAPPAPORT, T. S.; HEATH, R. W.; NIX, A.; RANGAN, S. Mimo for millimeter-wave wireless communications: beamforming, spatial multiplexing, or both? *IEEE Communications Magazine*, IEEE, v. 52, n. 12, p. 110–121, 2014.

2 QIAO, J. Enabling millimeter wave communication for 5G cellular networks: MAC-layer perspective. *University of Waterloo*, University of Waterloo, 2015.

3 PI, Z.; KHAN, F. An introduction to millimeter-wave mobile broadband systems. *IEEE Communications Magazine*, IEEE, v. 49, n. 6, 2011.

4 SHOKRI-GHADIKOLAEI, H.; FISCHIONE, C.; FODOR, G.; POPOVSKI, P.; ZORZI, M. Millimeter wave cellular networks: A MAC layer perspective. *IEEE Transactions on Communications*, IEEE, v. 63, n. 10, p. 3437–3458, 2015.

5 AKDENIZ, M. R.; LIU, Y.; SAMIMI, M. K.; SUN, S.; RANGAN, S.; RAPPAPORT, T. S.; ERKIP, E. Millimeter wave channel modeling and cellular capacity evaluation. *IEEE Journal on Selected Areas in Communications*, IEEE, v. 32, n. 6, p. 1164–1179, 2014.

6 ROH, W.; SEOL, J.-Y.; PARK, J.; LEE, B.; LEE, J.; KIM, Y.; CHO, J.; CHEUN, K.; ARYANFAR, F. Millimeter-wave beamforming as an enabling technology for 5G cellular communications: theoretical feasibility and prototype results. *IEEE Communications Magazine*, IEEE, v. 52, n. 2, p. 106–113, 2014.

7 BAI, T.; ALKHATEEB, A.; HEATH, R. W. Coverage and capacity of millimeter-wave cellular networks. *IEEE Communications Magazine*, IEEE, v. 52, n. 9, p. 70–77, 2014.

8 SUNG, N. W.; CHOI, Y. S. Fast intra-beam switching scheme using common contention channels in millimeter-wave based cellular systems. In: IEEE. *Advanced Communication Technology (ICACT), 2016 18th International Conference on*. [S.l.], 2016. p. 760–765.

9 NITSCHE, T.; FLORES, A. B.; KNIGHTLY, E. W.; WIDMER, J. Steering with eyes closed: mm-wave beam steering without in-band measurement. In: IEEE. *Computer Communications (INFOCOM), 2015 IEEE Conference on*. [S.l.], 2015. p. 2416–2424.

10 WANG, J. Beam codebook based beamforming protocol for multi-Gbps millimeter-wave WPAN systems. *IEEE Journal on Selected Areas in Communications*, IEEE, v. 27, n. 8, 2009.

11 LI, B.; ZHOU, Z.; ZOU, W.; SUN, X.; DU, G. On the efficient beam-forming training for 60GHz wireless personal area networks. *IEEE Transactions on Wireless Communications*, IEEE, v. 12, n. 2, p. 504–515, 2013.

12 JEONG, C.; PARK, J.; YU, H. Random access in millimeter-wave beamforming cellular networks: issues and approaches. *IEEE Communications Magazine*, IEEE, v. 53, n. 1, p. 180–185, 2015.

13 GIORDANI, M.; MEZZAVILLA, M.; BARATI, C. N.; RANGAN, S.; ZORZI, M. Comparative analysis of initial access techniques in 5g mmwave cellular networks. In: IEEE. *Information Science and Systems (CISS), 2016 Annual Conference on*. [S.l.], 2016. p. 268–273.

14 BARATI, C. N.; HOSSEINI, S. A.; MEZZAVILLA, M.; AMIRI-ELIASI, P.; RANGAN, S.; KORAKIS, T.; PANWAR, S. S.; ZORZI, M. Directional initial access for millimeter wave cellular systems. In: IEEE. *Signals, Systems and Computers, 2015 49th Asilomar Conference on*. [S.l.], 2015. p. 307–311.

15 GIORDANI, M.; MEZZAVILLA, M.; ZORZI, M. Initial access in 5g mmwave cellular networks. *IEEE Communications Magazine*, IEEE, v. 54, n. 11, p. 40–47, 2016.

16 ALAMOUTI, S. M. A simple transmit diversity technique for wireless communications. *IEEE Journal on selected areas in communications*, IEEE, v. 16, n. 8, p. 1451–1458, 1998.

17 BOCCARDI, F.; HEATH, R. W.; LOZANO, A.; MARZETTA, T. L.; POPOVSKI, P. Five disruptive technology directions for 5g. *IEEE Communications Magazine*, IEEE, v. 52, n. 2, p. 74–80, 2014.

18 RAPPAPORT, T. S.; SUN, S.; MAYZUS, R.; ZHAO, H.; AZAR, Y.; WANG, K.; WONG, G. N.; SCHULZ, J. K.; SAMIMI, M.; GUTIERREZ, F. Millimeter wave mobile communications for 5g cellular: It will work! *IEEE access*, IEEE, v. 1, p. 335–349, 2013.

19 LARSSON, E. G.; EDFORS, O.; TUFVESSON, F.; MARZETTA, T. L. Massive mimo for next generation wireless systems. *IEEE Communications Magazine*, IEEE, v. 52, n. 2, p. 186–195, 2014.

20 SHOKRI-GHADIKOLAEI, H.; FISCHIONE, C.; POPOVSKI, P.; ZORZI, M. Design aspects of short-range millimeter-wave networks: A MAC layer perspective. *IEEE Network*, IEEE, v. 30, n. 3, p. 88–96, 2016.

21 PRELCIC, N. G.; ALI, A.; VA, V.; JR, R. W. H. Millimeter wave communication with out-of-band information. 2017.

22 NIU, Y.; LI, Y.; JIN, D.; SU, L.; VASILAKOS, A. V. A survey of millimeter wave (mmwave) communications for 5g: Opportunities and challenges. *Computer Science-Networking and Internet Architecture*, 2015.

23 NITSCHE, T.; CORDEIRO, C.; FLORES, A. B.; KNIGHTLY, E. W.; PERAHIA, E.; WIDMER, J. C. Ieee 802.11 ad: directional 60 ghz communication for multi-gigabit-per-second wi-fi [invited paper]. *IEEE Communications Magazine*, IEEE, v. 52, n. 12, p. 132–141, 2014.

24 GOLDSMITH, A. *Wireless communications*. [S.l.]: Cambridge university press, 2005.

25 ROSENBROCK, H. An automatic method for finding the greatest or least value of a function. *The Computer Journal*, Oxford University Press, v. 3, n. 3, p. 175–184, 1960.

26 MADUENO, G. C.; STEFANOVIĆ, Č.; POPOVSKI, P. Efficient LTE access with collision resolution for massive M2M communications. In: IEEE. *Globecom Workshops (GC Wkshps), 2014*. [S.l.], 2014. p. 1433–1438.

27 LAYA, A.; ALONSO, L.; ALONSO-ZARATE, J. Is the random access channel of lte and lte-a suitable for m2m communications? a survey of alternatives. *IEEE Communications Surveys and Tutorials*, v. 16, n. 1, p. 4–16, 2014.

28 HAMPTON, J. R. *Introduction to MIMO communications*. [S.l.]: Cambridge university press, 2013.

29 CARVALHO, M. M.; GARCIA-LUNA-ACEVES, J. Analytical modeling of ad hoc networks that utilize space-time coding. In: IEEE. *Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks, 2006 4th International Symposium on*. [S.l.], 2006. p. 1–11.

30 MEZZAVILLA, M.; DUTTA, S.; ZHANG, M.; AKDENIZ, M. R.; RANGAN, S. 5G mmWave module for the ns-3 network simulator. In: ACM. *Proceedings of the 18th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*. [S.l.], 2015. p. 283–290.

31 NS-3 Network Simulator. Acessado em Maio, 2017. Disponível em: <https://www.nsnam.org/>.

32 SUN, S.; RAPPAPORT, T. S. Multi-beam antenna combining for 28 ghz cellular link improvement in urban environments. In: IEEE. *Global Communications Conference (GLOBECOM), 2013 IEEE*. [S.l.], 2013. p. 3754–3759.

33 NIE, S.; MACCARTNEY, G. R.; SUN, S.; RAPPAPORT, T. S. 28 ghz and 73 ghz signal outage study for millimeter wave cellular and backhaul communications. In: IEEE. *Communications (ICC), 2014 IEEE International Conference on*. [S.l.], 2014. p. 4856–4861.

34 RAPPAPORT, T. S. et al. *Wireless communications: principles and practice*. [S.l.]: prentice hall PTR New Jersey, 1996. v. 2.

35 BALDO, N. *The ns-3 LTE module by the LENA project*. Acessado em Maio, 2017. Disponível em: <https://www.nsnam.org/tutorials/consortium13/lte-tutorial.pdf>.

36 MEZZAVILLA, M.; MIOZZO, M.; ROSSI, M.; BALDO, N.; ZORZI, M. A lightweight and accurate link abstraction model for the simulation of lte networks in ns-3. In: ACM. *Proceedings of the 15th ACM international conference on Modeling, analysis and simulation of wireless and mobile systems*. [S.l.], 2012. p. 55–60.

37 FORD, R.; ZHANG, M.; DUTTA, S.; MEZZAVILLA, M.; RANGAN, S.; ZORZI, M. A framework for cross-layer evaluation of 5g mmwave cellular networks in ns-3.

38 KHAN, F.; PI, Z. mmwave mobile broadband (mmb): Unleashing the 3–300ghz spectrum. In: IEEE. *Sarnoff Symposium, 2011 34th IEEE*. [S.l.], 2011. p. 1–6.

APÊNDICES

Anexo I: Script de execução e classes alteradas no código fonte do simulador ns-3.

Script de simulação:

```
1   //Author: Thayane Viana <thayaneviana@hotmail.com>
2
3  /* Script de um cen rio com uma c lula mmWave com 1 dispositivo e 1 eNB.
4   * Dispositivo em posi o est tica.
5   * Transmiss o de pacotes de controle e dados.
6   *
7
8   /* Information regarding the traces generated:
9     *
10    * 1. UE_1_SINR.txt : Gives the SINR for each sub-band
11    *    Subframe no.  | Slot No. | Sub-band  | SINR (db)
12    *
13    * 2. UE_1_Tb_size.txt : Allocated transport block size
14    *    Time (micro-sec)  |  Tb-size in bytes
15    * */
16
17
18 #include "ns3/mmwave-helper.h"
19 #include "ns3/epc-helper.h"
20 #include "ns3/core-module.h"
21 #include "ns3/network-module.h"
22 #include "ns3/ipv4-global-routing-helper.h"
23 #include "ns3/internet-module.h"
24 #include "ns3/mobility-module.h"
25 #include "ns3/applications-module.h"
26 #include "ns3/point-to-point-helper.h"
27 #include "ns3/config-store.h"
28 #include "ns3/mmwave-point-to-point-epc-helper.h"
29 //#include "ns3/gtk-config-store.h"
30 #include "ns3/flow-monitor-module.h"
31 #include "ns3/pointer.h"
32 #include <ns3/buildings-helper.h>
33 #include "ns3/log.h"
34 #include <ns3/buildings-module.h>
35 #include "ns3/radio-environment-map-helper.h"
36 //#include "ns3/radio-environment-map-helper.h"
37
38 #include <iostream>
39 #include <vector>
40 #include <cstdio>
41 #include <cstdlib>
42 #include <stdexcept>
43 #include <ctime>
44 #include <iomanip>
45 #include <fstream>
46 #include <string.h>
47
48 using namespace ns3;
49 using namespace std;
50
51
52 NS_LOG_COMPONENT_DEFINE ("mmWave");
53
54 double stop = 1;
55 double warmup = 0.01;
56 double dist = 20;
57
58
59 void ComputeResults (void);
60 struct sim_Result
61 {
62   uint64_t sumRxBytesByFlow;
63   uint64_t sumRxBytesQuadByFlow;
64   uint64_t sumLostPktsByFlow;
65   uint64_t sumRxPktsByFlow;
66   uint64_t sumTxPktsByFlow;
67   uint64_t sumDelayFlow;
68   uint64_t nFlows;
69
70        /* Throughput Average by Flow (bps) = sumRxBytesByFlow * 8 / (nFlows * time)
71        * Throughput Quadratic Average by Flow (bps) = sumRxBytesQuadByFlow * 64 / (nFlows * time * time)
72        * Net Aggregated Throughput Average by Node (bps) = sumRxBytesByFlow * 8 / (nodes * time)
73        * Fairness = sumRxBytesByFlow^2 / (nFlows * sumRxBytesQuadByFlow)
74        * Delay per Packet (seconds/packet) = sumDelayFlow / sumRxPktsByFlow
75        * Lost Ratio (%) = 100 * sumLostPktsByFlow / sumTxPktsByFlow
76        */
77   double thrpAvgByFlow;
```

```
78    double thrpAvgQuadByFlow;
79    double thrpVarByFlow;
80    double netThrpAvgByNode;
81    double fairness;
82    double delayByPkt;
83    double lostRatio;
84    double pdr;
85
86    sim_Result ()
87    {
88      sumRxBytesByFlow = 0;
89      sumRxBytesQuadByFlow = 0;
90      sumLostPktsByFlow = 0;
91      sumRxPktsByFlow = 0;
92      sumTxPktsByFlow = 0;
93      sumDelayFlow = 0;
94      nFlows = 0;
95    }
96  } data;
97
98  int
99  main (int argc, char *argv[])
100 {
101       //LogComponentEnable("MmWaveChannelMatrix",LOG_LEVEL_DEBUG);
102       //LogComponentEnable ("LteUeRrc", LOG_LEVEL_ALL);
103       //LogComponentEnable ("LteEnbRrc", LOG_LEVEL_ALL);
104       //   LogComponentEnable("MmWavePointToPointEpcHelper",LOG_LEVEL_ALL);
105       //   LogComponentEnable("EpcUeNas",LOG_LEVEL_ALL);
106       //LogComponentEnable ("MmWaveSpectrumPhy", LOG_LEVEL_DEBUG);
107           //LogComponentEnable ("MmWaveSpectrumPhy", LOG_LEVEL_FUNCTION);
108       //LogComponentEnable ("MmWaveUePhy", LOG_LEVEL_DEBUG);
109       //LogComponentEnable ("MmWaveUePhy", LOG_LEVEL_DEBUG);
110       //LogComponentEnable ("MmWavePhy", LOG_LEVEL_DEBUG);
111       //LogComponentEnable ("MmWavePhy", LOG_LEVEL_FUNCTION);
112       //   LogComponentEnable ("MmWaveEnbPhy", LOG_LEVEL_DEBUG);
113       //LogComponentEnable ("MmWaveEnbMac", LOG_LEVEL_DEBUG);
114       //LogComponentEnable ("MmWaveRrMacScheduler", LOG_LEVEL_ALL);
115       //LogComponentEnable ("MmWaveUeMac", LOG_LEVEL_DEBUG);
116       //LogComponentEnable ("MmWaveUeMac", LOG_LEVEL_FUNCTION);
117       //LogComponentEnable ("MmWaveChannelMatrix", LOG_LEVEL_FUNCTION);
118       //LogComponentEnable ("UdpClient", LOG_LEVEL_INFO);
119       //LogComponentEnable ("PacketSink", LOG_LEVEL_INFO);
120       //LogComponentEnable("PropagationLossModel",LOG_LEVEL_ALL);
121       //LogComponentEnable("PropagationLossModel",LOG_LEVEL_DEBUG);
122       //LogComponentEnable ("MmWaveBeamforming", LOG_LEVEL_DEBUG);
123       //LogComponentEnable ("MmWaveAmc", LOG_LEVEL_LOGIC);
124
125
126       //LogComponentEnable ("MmWaveMiErrorModel", LOG_LEVEL_LOGIC);
127       //LogComponentEnable ("mmWaveAmc", LOG_LEVEL_DEBUG);
128
129
130
131       uint16_t numEnb = 1;
132       uint16_t numUe = 1;
133       double simTime = 1;
134       double interPacketInterval = 1; //(ms)
135
136
137       // Command line arguments
138       CommandLine cmd;
139       cmd.AddValue("numEnb", "Number of eNBs", numEnb);
140       cmd.AddValue("numUe", "Number of UEs per eNB", numUe);
141       cmd.AddValue("simTime", "Total duration of the simulation [s])", simTime);
142       cmd.AddValue("interPacketInterval", "Inter packet interval [ms])", interPacketInterval);
143       cmd.AddValue("dist", "Distance eNB-node", dist);
144       cmd.Parse(argc, argv);
145
146
147       //The number of TTIs a CQI is valid (default 1000 - 1 sec.)
148       //Config::SetDefault ("ns3::MmWaveRrMacScheduler::CqiTimerThreshold", UintegerValue (100));
149
150
151       //The carrier frequency (in Hz) at which propagation occurs  (default is 28 GHz)
152        Config::SetDefault ("ns3::MmWavePropagationLossModel::Frequency",  DoubleValue (28e9));
153        Config::SetDefault ("ns3::MmWaveEnbPhy::TxPower",  DoubleValue (30));
154        Config::SetDefault ("ns3::MmWaveUePhy::TxPower",  DoubleValue (20));
155
156        Config::SetDefault ("ns3::MmWaveEnbNetDevice::AntennaNum", UintegerValue(64));
157        Config::SetDefault ("ns3::MmWaveUeNetDevice::AntennaNum", UintegerValue(16));
158
```

```
159       //The minimum value (dB) of the total loss, used at short ranges (distance<0)
160      // Config::SetDefault ("ns3::MmWavePropagationLossModel::MinLoss",  DoubleValue (20.0));
161
162       //Loss (dB) in the Signal-to-Noise-Ratio due to non-idealities in the receiver.
163        Config::SetDefault ("ns3::MmWaveEnbPhy::NoiseFigure",  DoubleValue (5.0));
164
165       //The no. of packets received and transmitted by the Base Station
166       //Config::SetDefault ("ns3::MmWaveSpectrumPhy::ReportEnbTxRxPacketCount",  MakeTraceSourceAccessor (&MmWaveSpectrumPhy::
              m_reportEnbPacketCount));
167
168       //Activate/Deactivate the HARQ [by default is active].
169       //Config::SetDefault ("ns3::MmWaveRrMacScheduler::HarqEnabled", BooleanValue(false));
170       //A classe MmWavePhyMacCommon tem v rios atributos, olhar c digo da classe
171       Config::SetDefault ("ns3::MmWavePhyMacCommon::ResourceBlockNum", UintegerValue(1));
172       Config::SetDefault ("ns3::MmWavePhyMacCommon::ChunkPerRB", UintegerValue(72));
173
174       Ptr<MmWaveHelper> mmwaveHelper = CreateObject<MmWaveHelper> ();
175       Ptr<MmWavePointToPointEpcHelper>  epcHelper = CreateObject<MmWavePointToPointEpcHelper> ();
176
177       mmwaveHelper->SetAttribute("PathlossModel", StringValue ("ns3::MmWavePropagationLossModel"));
178       // Configure number of antennas
179       mmwaveHelper->SetAntenna (16,64);
180       mmwaveHelper->SetEpcHelper (epcHelper);
181
182       ConfigStore inputConfig;
183       inputConfig.ConfigureDefaults();
184
185       // parse again so you can override default values from the command line
186       cmd.Parse(argc, argv);
187
188       Ptr<Node> pgw = epcHelper->GetPgwNode ();
189
190       // Create a single RemoteHost
191       NodeContainer remoteHostContainer;
192       remoteHostContainer.Create (1);
193       Ptr<Node> remoteHost = remoteHostContainer.Get (0);
194       InternetStackHelper internet;
195       internet.Install (remoteHostContainer);
196
197       // Create the Internet
198       PointToPointHelper p2ph;
199       p2ph.SetDeviceAttribute ("DataRate", DataRateValue (DataRate ("100Gb/s")));
200       p2ph.SetDeviceAttribute ("Mtu", UintegerValue (1500));
201       p2ph.SetChannelAttribute ("Delay", TimeValue (Seconds (0.010)));
202       NetDeviceContainer internetDevices = p2ph.Install (pgw, remoteHost);
203       Ipv4AddressHelper ipv4h;
204       ipv4h.SetBase ("1.0.0.0", "255.0.0.0");
205       Ipv4InterfaceContainer internetIpIfaces = ipv4h.Assign (internetDevices);
206       // interface 0 is localhost, 1 is the p2p device
207       //Ipv4Address remoteHostAddr = internetIpIfaces.GetAddress (1);
208
209       Ipv4StaticRoutingHelper ipv4RoutingHelper;
210       Ptr<Ipv4StaticRouting> remoteHostStaticRouting = ipv4RoutingHelper.GetStaticRouting (remoteHost->GetObject<Ipv4> ());
211       remoteHostStaticRouting->AddNetworkRouteTo (Ipv4Address ("7.0.0.0"), Ipv4Mask ("255.0.0.0"), 1);
212
213       NodeContainer ueNodes;
214       NodeContainer enbNodes;
215       enbNodes.Create(numEnb);
216       ueNodes.Create(numUe);
217       NodeContainer allNodes;
218
219       for (uint16_t i = 0; i < numUe; i++)
220     {
221         allNodes.Add(ueNodes.Get(i));
222     }
223
224       //allNodes.Add(enbNodes.Get(0));
225       allNodes.Add(remoteHostContainer.Get(0));
226
227
228
229       // Install Mobility Model
230       Ptr<ListPositionAllocator> enbPositionAlloc = CreateObject<ListPositionAllocator> ();
231       enbPositionAlloc->Add (Vector (0.0, 0.0, 30.0));
232       MobilityHelper enbmobility;
233       enbmobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
234       enbmobility.SetPositionAllocator(enbPositionAlloc);
235       enbmobility.Install (enbNodes);
236
237
238
```

```
239       MobilityHelper uemobility;
240       Ptr<ListPositionAllocator> uePositionAlloc = CreateObject<ListPositionAllocator> ();
241
242       //Gera  o de posi  es aleat rias dentro de um raio m ximo
243       //Obs: n o est  funcionando para mais que 4 n s
244       /*int n;
245       int raio = 100;
246       int dist[numUe];
247       double x[numUe];
248       double y[numUe];
249       srand( (unsigned)time(NULL) );
250       for (n=0 ; n < numUe; n++)
251       {
252           dist[n] = rand() % (raio+1);
253           x[n] = rand () % (dist[n]+1);
254           y[n] = sqrt (pow(dist[n],2) - pow(x[n],2));
255           //std::cout << "N =" << n << endl;
256           //std::cout << "dist=" << dist[n] << endl;
257           //std::cout << "x=" << x[n] << endl;
258           //std::cout << "y=" << y[n] << endl;
259           uePositionAlloc->Add (Vector (x[n], y[n], 1.5));
260           printf("uePositionAlloc->Add (Vector (%f, %f, 1.5)); \n", x[n], y[n]);
261           //std::cout << "-----------------------------------" << endl;
262       }*/
263
264       uePositionAlloc->Add (Vector (0.0, dist, 1.5));
265
266       //uePositionAlloc->Add (Vector (100.0, 0.0, 1.5));
267       //uePositionAlloc->Add (Vector (30.0, 40.0, 1.5));
268       //uePositionAlloc->Add (Vector (230.0, 230.0, 1.5));
269       //uePositionAlloc->Add (Vector (250.0, 250.0, 1.5));
270       //uePositionAlloc->Add (Vector (20.0, 0.0, 1.5));
271       //uePositionAlloc->Add (Vector (80.0, 0.0, 1.5));
272
273
274
275
276
277       uemobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
278       uemobility.SetPositionAllocator(uePositionAlloc);
279       uemobility.Install (ueNodes);
280
281
282       // Install mmWave Devices to the nodes
283       NetDeviceContainer enbmmWaveDevs = mmwaveHelper->InstallEnbDevice (enbNodes);
284       NetDeviceContainer uemmWaveDevs = mmwaveHelper->InstallUeDevice (ueNodes);
285
286       // Install the IP stack on the UEs
287       internet.Install (ueNodes);
288       Ipv4InterfaceContainer ueIpIface;
289       ueIpIface = epcHelper->AssignUeIpv4Address (NetDeviceContainer (uemmWaveDevs));
290       // Assign IP address to UEs, and install applications
291       for (uint32_t u = 0; u < ueNodes.GetN (); ++u)
292       {
293           Ptr<Node> ueNode = ueNodes.Get (u);
294           // Set the default gateway for the UE
295           Ptr<Ipv4StaticRouting> ueStaticRouting = ipv4RoutingHelper.GetStaticRouting (ueNode->GetObject<Ipv4> ());
296           ueStaticRouting->SetDefaultRoute (epcHelper->GetUeDefaultGatewayAddress (), 1);
297       }
298
299       mmwaveHelper->AttachToClosestEnb (uemmWaveDevs, enbmmWaveDevs);
300
301       mmwaveHelper->EnableTraces ();
302       //mmwaveHelper->EnableDlPhyTrace (); // Uncomment to enable PCAP tracing
303       mmwaveHelper->Initialize();
304
305
306
307   /*Ptr<RadioEnvironmentMapHelper> remHelper = CreateObject<RadioEnvironmentMapHelper> ();
308   remHelper->SetAttribute ("ChannelPath", StringValue ("/ChannelList/0"));
309   remHelper->SetAttribute ("OutputFile", StringValue ("rem.out"));
310   remHelper->SetAttribute ("XMin", DoubleValue (-400.0));
311   remHelper->SetAttribute ("XMax", DoubleValue (400.0));
312   remHelper->SetAttribute ("XRes", UintegerValue (100));
313   remHelper->SetAttribute ("YMin", DoubleValue (-300.0));
314   remHelper->SetAttribute ("YMax", DoubleValue (300.0));
315   remHelper->SetAttribute ("YRes", UintegerValue (75));
316   remHelper->SetAttribute ("Z", DoubleValue (0.0));
317   remHelper->SetAttribute ("UseDataChannel", BooleanValue (true));
318   remHelper->SetAttribute ("RbId", IntegerValue (10));
319   remHelper->Install ();*/
```

```
320
321
322       // Install and start applications on UEs and remote host
323       uint16_t dlPort = 80;
324       //uint16_t ulPort = 80;
325       ApplicationContainer clientApps;
326       ApplicationContainer serverApps;
327       for (uint32_t u = 0; u < ueNodes.GetN (); ++u)
328       {
329           //++ulPort;
330
331           PacketSinkHelper dlPacketSinkHelper ("ns3::UdpSocketFactory", InetSocketAddress (Ipv4Address::GetAny (), dlPort));
332           //PacketSinkHelper ulPacketSinkHelper ("ns3::UdpSocketFactory", InetSocketAddress (Ipv4Address::GetAny (), ulPort));
333
334           serverApps.Add (dlPacketSinkHelper.Install (ueNodes.Get(u)));
335           //serverApps.Add (ulPacketSinkHelper.Install (remoteHost));
336
337
338           UdpClientHelper dlClient (ueIpIface.GetAddress (u), dlPort);
339           dlClient.SetAttribute ("Interval", TimeValue (MilliSeconds(interPacketInterval)));
340           dlClient.SetAttribute ("MaxPackets", UintegerValue(1000000));
341
342           //UdpClientHelper ulClient (remoteHostAddr, ulPort);
343           //ulClient.SetAttribute ("Interval", TimeValue (MilliSeconds(interPacketInterval)));
344           //ulClient.SetAttribute ("MaxPackets", UintegerValue(1000000));
345
346
347
348           clientApps.Add (dlClient.Install (remoteHost));
349           //clientApps.Add (ulClient.Install (ueNodes.Get(u)));
350
351       }
352
353
354        p2ph.EnablePcapAll("cenario15");
355         // FlowMonitor
356
357       // Activate a data radio bearer
358       /*enum EpsBearer::Qci q = EpsBearer::GBR_CONV_VOICE;
359       EpsBearer bearer (q);
360       mmwaveHelper->ActivateDataRadioBearer (uemmWaveDevs, bearer);*/
361
362       FlowMonitorHelper flowmon;
363       Ptr<FlowMonitor> monitor = flowmon.Install(allNodes);
364       monitor->Start (Seconds (0.0)); // start monitoring after network warm up
365       monitor->Stop (Seconds (simTime)); // stop monitoring
366
367       Simulator::Stop(Seconds(simTime));
368       Simulator::Run ();
369
370       Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier> (flowmon.GetClassifier());
371
372       std::map<FlowId, FlowMonitor::FlowStats> stats = monitor->GetFlowStats();
373
374
375       for (std::map<FlowId, FlowMonitor::FlowStats>::const_iterator i = stats.begin (); i != stats.end (); ++i)
376       {
377           Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow (i->first);
378           if (t.destinationPort == 80) // only http flows
379         {
380           std::cout << "Flow " << i->first << " (" << t.sourceAddress << " -> " << t.destinationAddress << "Port:"<< t.
                destinationPort<<")\n";
381           std::cout << "  Tx Bytes:   " << i->second.txBytes << "\n";
382           std::cout << "  Rx Bytes:   " << i->second.rxBytes << "\n";
383           std::cout << "  Throughput: " << i->second.rxBytes * 8.0 / (simTime - 0.01) / 1024 / 1024  << " Mbps\n";
384           std::cout << "  Packet Delivery Ratio: " << ((double)(i->second.rxPackets)/(double)(i->second.txPackets))*100 << "%\n"
                ;
385
386       }
387
388         if (t.destinationPort == 80) // only http flows
389         {
390       data.nFlows++;
391       data.sumRxBytesByFlow += i->second.rxBytes; // sum flows
392       data.sumRxBytesQuadByFlow += i->second.rxBytes * i->second.rxBytes; // sum flows
393       data.sumDelayFlow += i->second.delaySum.GetInteger (); // sum delays
394       data.sumRxPktsByFlow += i->second.rxPackets; // sum rx pkts
395       data.sumTxPktsByFlow += i->second.txPackets; // sum tx pkts
396       data.sumLostPktsByFlow += i->second.lostPackets; // sum lost pkts
397
398
```

```
399       }
400
401
402
403     }
404       serverApps.Start (Seconds (0.0));
405       clientApps.Start (Seconds (0.01));
406
407
408
409
410
411
412
413
414
415   //   Simulator::Stop(Seconds(simTime));
416   //   Simulator::Run();
417
418       /*GtkConfigStore config;
419     config.ConfigureAttributes();*/
420
421       Simulator::Destroy();
422
423       ComputeResults ();
424       return 0;
425
426   }
427
428   void
429   ComputeResults (void)
430   {
431     double deltaT = (stop - warmup);
432       // Throughput Average by Flow (bps)
433     data.thrpAvgByFlow = (double) data.sumRxBytesByFlow * 8 / (data.nFlows * deltaT);
434     // Throughput Quadratic Average by Flow (bps )
435     data.thrpAvgQuadByFlow = (double) data.sumRxBytesQuadByFlow * 8*8 / (data.nFlows * deltaT*deltaT);
436     // Throughput Variance by Flow (bps )
437     data.thrpVarByFlow = data.thrpAvgQuadByFlow - data.thrpAvgByFlow * data.thrpAvgByFlow;
438     // Network Aggregated Throughput Average by Node (bps)
439     //data.netThrpAvgByNode = (double) data.sumRxBytesByFlow * 8 / (numFluxos * deltaT);
440     // Fairness Jain s Index
441     data.fairness = (double) data.sumRxBytesByFlow * data.sumRxBytesByFlow / (data.nFlows * data.sumRxBytesQuadByFlow);
442     // Delay Mean by Packet (nanoseconds)
443     data.delayByPkt = (double) data.sumDelayFlow / data.sumRxPktsByFlow;
444     // Lost Ratio (%)
445     data.lostRatio = (double) 100 * data.sumLostPktsByFlow / data.sumTxPktsByFlow;
446     data.pdr = (double) 100 * data.sumRxPktsByFlow / data.sumTxPktsByFlow;
447
448
449     cout << "================================================================================" << endl
450          << "Simulation results:" << endl
451          << "Throughput Average by Flow (kbps):\t" << data.thrpAvgByFlow / 1024.0 << endl
452          << "Throughput Deviation by Flow (kbps):\t" << sqrt (data.thrpVarByFlow) / 1024.0 << endl
453         // << "Network Aggregated Throughput Average by Node (kbps):\t" << data.netThrpAvgByNode / 1024.0 << endl
454          << "Fairness Jain s Index:\t" << data.fairness << endl
455          << "Delay Mean by Packet (seconds):\t" << data.delayByPkt / 1e9 << endl
456          << "Packet Lost Ratio (%):\t" << data.lostRatio << endl
457          << "Packet Delivery Ratio (%):\t" << data.pdr<<endl<< endl<< endl;
458
459     cout << "Flows: " << data.nFlows << endl;
460
461
462       /*ofstream throughput;
463       throughput.open ("Resultados/throughput", ios::app);
464       throughput << "\t" << data.nFlows << "\t"<< std::setprecision(4) << data.thrpAvgByFlow / 1024.0 << endl;
465       throughput.close ();*/
466
467
468       /*ofstream delay;
469       delay.open ("Resultados/delay", ios::app);
470       delay << "\t" << data.nFlows << "\t" << std::setprecision(4) << data.delayByPkt / 1e6 <<  endl;
471       delay.close ();
472
473
474       ofstream fairness;
475       fairness.open ("Resultados/fairness", ios::app);
476       fairness << "\t" << data.nFlows << "\t"  << std::setprecision(4) << data.fairness << endl;
477       fairness.close ();*/
478
479       ofstream pdr;
```

```
480       pdr.open ("Resultados/pdr", ios::app);
481       pdr << "dist=" << dist << "\t" << std::setprecision(4) << data.pdr << endl;
482       pdr.close ();
483
484       /*ofstream packets;
485       packets.open ("Resultados/packets", ios::app);
486       packets << "\t" << data.sumTxPktsByFlow << "\t" << std::setprecision (4) << data.sumRxPktsByFlow << endl;
487       packets.close ();*/
488
489
490
491   }
```

Alterações no código fonte do simulador ns-3. Classe: mmwave-beamforming.cc

```
1
2
3   ------------------------------------------------------------------------------------------------------
4   /*
5    * MmWaveBeamforming.cc
6    *
7    *  Created on: 2014      1125
8    *      Author: menglei
9    *
10   * Modified on: 07/19/2017 (Alamouti squeme inclusion for control channel)
11   * Author: Thayane Viana
12   */
13  // ---------------------------------- ALAMOUTI 2x32 ----------------------------------------------------
14  #include "mmwave-beamforming.h"
15
16  #include <ns3/log.h>
17  #include <fstream>
18  #include <ns3/simulator.h>
19  #include <ns3/abort.h>
20  #include <ns3/mmwave-enb-net-device.h>
21  #include <ns3/mmwave-ue-net-device.h>
22  #include <ns3/mmwave-ue-phy.h>
23  #include <ns3/antenna-array-model.h>
24  #include <ns3/node.h>
25  #include <algorithm>
26  #include <ns3/double.h>
27  #include <ns3/boolean.h>
28
29  namespace ns3{
30
31  NS_LOG_COMPONENT_DEFINE ("MmWaveBeamforming");
32
33  NS_OBJECT_ENSURE_REGISTERED (MmWaveBeamforming);
34
35  // number of channel matrix instance in beamforming files
36  // period of updating channel matrix
37  static const uint32_t g_numInstance = 100;
38
39  complex2DVector_t g_enbAntennaInstance; //100 instance of txW
40  complex2DVector_t g_ueAntennaInstance; //100 instance of rxW
41  complex3DVector_t g_enbSpatialInstance; //this stores 100 instance of txE
42  complex3DVector_t g_ueSpatialInstance; //this stores 100 instance of rxE
43  double2DVector_t g_smallScaleFadingInstance;     //this stores 100 instance of sigma vector
44
45  /*
46   * The delay spread and Doppler shift is not based on measurement data at this time
47   */
48  static const double DelaySpread[20]  = {0, 3e-9, 4e-9, 5e-9, 5e-9, 6e-9, 7e-9, 7e-9, 7e-9, 17e-9,
49              18e-9, 20e-9, 23e-9, 24e-9, 26e-9, 38e-9, 40e-9, 42e-9, 45e-9, 50e-9};
50
51  static const double DopplerShift[20] = {0.73, 0.78, 0.68, 0.71, 0.79, 0.69, 0.66, 0.70, 0.69, 0.44,
52          0.48, 0.43, 0.42, 0.47,0.50, 0.53, 0.52, 0.49, 0.55, 0.52};
53
54  //Quantidade de antenas utilizadas pelo Alamouti
55  uint32_t txAntennaSizeAlamouti = 2; //2 antenas
56  uint32_t rxAntennaSizeAlamouti = 32; //2 antenas
57
58  MmWaveBeamforming::MmWaveBeamforming (uint32_t enbAntenna, uint32_t ueAntenna)
59      :m_pathNum (20),
60      m_enbAntennaSize(enbAntenna),
61      m_ueAntennaSize(ueAntenna),
62      m_longTermUpdatePeriod (0),
```

```
63      m_smallScale (true),
64      m_fixSpeed (false),
65      m_ueSpeed (0.0),
66      m_update(true)
67  {
68      if (g_smallScaleFadingInstance.empty ())
69      LoadFile();
70      m_uniformRV = CreateObject<UniformRandomVariable> ();
71  }
72
73
74  TypeId
75  MmWaveBeamforming::GetTypeId (void)
76  {
77      static TypeId tid = TypeId ("ns3::MmWaveBeamforming")
78          .SetParent<Object> ()
79          .AddAttribute ("LongTermUpdatePeriod",
80                         "Time (ms) between periodic updating of channel matrix/beamforming vectors",
81              TimeValue (MilliSeconds (100.0)),
82              MakeTimeAccessor (&MmWaveBeamforming::m_longTermUpdatePeriod),
83              MakeTimeChecker ())
84          .AddAttribute ("SmallScaleFading",
85                                      "Enable small scale fading",
86                                      BooleanValue (true),
87                                      MakeBooleanAccessor (&MmWaveBeamforming::m_smallScale),
88                                      MakeBooleanChecker ())
89          .AddAttribute ("FixSpeed",
90                                      "Set a fixed speed (even if constant position) so doppler > 0 for testing",
91                                      BooleanValue (false),
92                                      MakeBooleanAccessor (&MmWaveBeamforming::m_fixSpeed),
93                                      MakeBooleanChecker ())
94          .AddAttribute ("UeSpeed",
95                                      "UE speed (m/s) for fixed speed test",
96                                      DoubleValue (0.0),
97                                      MakeDoubleAccessor (&MmWaveBeamforming::m_ueSpeed),
98                                      MakeDoubleChecker<double> ())
99      ;
100     return tid;
101 }
102
103 MmWaveBeamforming::~MmWaveBeamforming ()
104 {
105
106 }
107
108 void
109 MmWaveBeamforming::DoDispose ()
110 {
111     NS_LOG_FUNCTION (this);
112 }
113
114 void
115 MmWaveBeamforming::SetCofigurationParameters (Ptr<MmWavePhyMacCommon> ptrConfig)
116 {
117     m_phyMacConfig = ptrConfig;
118 }
119
120 Ptr<MmWavePhyMacCommon>
121 MmWaveBeamforming::GetConfigurationParameters (void) const
122 {
123     return m_phyMacConfig;
124 }
125
126 std::complex<double>
127 MmWaveBeamforming::ParseComplex (std::string strCmplx)
128 {
129     double re = 0.00;
130     double im = 0.00;
131     size_t findj = 0;
132     std::complex<double> out_complex;
133
134     findj = strCmplx.find("i");
135     if( findj == std::string::npos )
136     {
137         im = -1.00;
138     }
139     else
140     {
141         strCmplx[findj] = \0 ;
142     }
143     if( ( strCmplx.find("+",1) == std::string::npos && strCmplx.find("-",1) == std::string::npos ) && im != -1 )
```

```
144         {
145             /* No real value */
146             re = -1.00;
147         }
148         std::stringstream stream( strCmplx );
149         if( re != -1.00 )
150         {
151             stream>>re;
152         }
153         else
154         {
155             re = 0;
156         }
157         if( im != -1 )
158         {
159             stream>>im;
160         }
161         else
162         {
163             im = 0.00;
164         }
165         //   std::cout<<" ----  "<<re<<"  "<<im<<std::endl;
166         out_complex = std::complex<double>(re,im);
167         return out_complex;
168     }
169
170     void
171     MmWaveBeamforming::LoadFile()
172     {
173         LoadSmallScaleFading ();
174         LoadEnbAntenna ();
175         LoadUeAntenna ();
176         LoadEnbSpatialSignature ();
177         LoadUeSpatialSignature ();
178     }
179
180
181     void
182     MmWaveBeamforming::LoadSmallScaleFading ()
183     {
184         std::string filename = "src/mmwave/model/BeamFormingMatrix/SmallScaleFading.txt";
185         NS_LOG_FUNCTION (this << "Loading SmallScaleFading file " << filename);
186         std::ifstream singlefile;
187         singlefile.open (filename.c_str (), std::ifstream::in);
188
189         NS_LOG_INFO (this << " File: " << filename);
190         NS_ASSERT_MSG(singlefile.good (), " SmallScaleFading file not found");
191         std::string line;
192         std::string token;
193         while( std::getline(singlefile, line) ) //Parse each line of the file
194         {
195             doubleVector_t path;
196             std::istringstream stream(line);
197             while( getline(stream,token, ',') ) //Parse each comma separated string in a line
198             {
199                 double sigma = 0.00;
200                 std::stringstream stream( token );
201                 stream>>sigma;
202                 path.push_back(sigma);
203             }
204             g_smallScaleFadingInstance.push_back (path);
205         }
206         NS_LOG_INFO ("SmallScaleFading[instance:"<<g_smallScaleFadingInstance.size()<<"][path:"<<g_smallScaleFadingInstance[0].
                size()<<"]");
207     }
208
209     void
210     MmWaveBeamforming::LoadEnbAntenna ()
211     {
212         std::string filename = "src/mmwave/model/BeamFormingMatrix/TxAntenna.txt";
213         NS_LOG_FUNCTION (this << "Loading TxAntenna file " << filename);
214         std::ifstream singlefile;
215         std::complex<double> complexVar;
216         singlefile.open (filename.c_str (), std::ifstream::in);
217
218         NS_LOG_INFO (this << " File: " << filename);
219         NS_ASSERT_MSG(singlefile.good (), " TxAntenna file not found");
220         std::string line;
221         std::string token;
222         while( std::getline(singlefile, line) ) //Parse each line of the file
223         {
```

```
224             complexVector_t txAntenna;
225             std::istringstream stream(line);
226             while( getline(stream,token, ',') ) //Parse each comma separated string in a line
227             {
228                 complexVar = ParseComplex(token);
229                 txAntenna.push_back(complexVar);
230             }
231             g_enbAntennaInstance.push_back(txAntenna);
232         }
233         NS_LOG_INFO ("TxAntenna[instance:"<<g_enbAntennaInstance.size()<<"][antennaSize:"<<g_enbAntennaInstance[0].size()<<"]");
234     }
235
236
237     void
238     MmWaveBeamforming::LoadUeAntenna ()
239     {
240         std::string filename = "src/mmwave/model/BeamFormingMatrix/RxAntenna.txt";
241         NS_LOG_FUNCTION (this << "Loading RxAntenna file " << filename);
242         std::ifstream singlefile;
243         std::complex<double> complexVar;
244         singlefile.open (filename.c_str (), std::ifstream::in);
245
246         NS_LOG_INFO (this << " File: " << filename);
247         NS_ASSERT_MSG(singlefile.good (), " RxAntenna file not found");
248
249         std::string line;
250         std::string token;
251         while( std::getline(singlefile, line) ) //Parse each line of the file
252         {
253             complexVector_t rxAntenna;
254             std::istringstream stream(line);
255             while( getline(stream,token, ',') ) //Parse each comma separated string in a line
256             {
257                 complexVar = ParseComplex(token);
258                 rxAntenna.push_back(complexVar);
259             }
260             g_ueAntennaInstance.push_back(rxAntenna);
261         }
262         NS_LOG_INFO ("RxAntenna[instance:"<<g_ueAntennaInstance.size()<<"][antennaSize:"<<g_ueAntennaInstance[0].size()<<"]");
263     }
264
265     void
266     MmWaveBeamforming::LoadEnbSpatialSignature ()
267     {
268         std::string filename = "src/mmwave/model/BeamFormingMatrix/TxSpatialSigniture.txt";
269         NS_LOG_FUNCTION (this << "Loading TxspatialSigniture file " << filename);
270         std::ifstream singlefile;
271         std::string line;
272         std::string token;
273
274         uint16_t counter = 1;
275         std::complex<double> complexVar;
276         complex2DVector_t txSpatialMatrix;
277         singlefile.open (filename.c_str (), std::ifstream::in);
278
279         NS_ASSERT_MSG (singlefile.good (), "TxSpatialSigniture file not found");
280
281         while( std::getline(singlefile, line) ) //Parse each line of the file
282         {
283             complexVector_t txSpatialElement;
284             std::istringstream stream(line);
285             while( getline(stream,token, ',') ) //Parse each comma separated string in a line
286             {
287                 complexVar = ParseComplex(token);
288                 txSpatialElement.push_back(complexVar);
289             }
290             txSpatialMatrix.push_back(txSpatialElement);
291             if(counter % m_pathNum ==0 )
292             {
293                 g_enbSpatialInstance.push_back(txSpatialMatrix);
294                 txSpatialMatrix.clear();
295             }
296             counter++;
297         }
298         NS_LOG_INFO ("TxspatialSigniture[instance:"<<g_enbSpatialInstance.size()<<"][path:"<<g_enbSpatialInstance[0].size()<<"][
                antennaSize:"<<g_enbSpatialInstance[0][0].size()<<"]");
299     }
300
301     void
302     MmWaveBeamforming::LoadUeSpatialSignature ()
303     {
```

```
304     std::string strFilename = "src/mmwave/model/BeamFormingMatrix/RxSpatialSigniture.txt";
305     NS_LOG_FUNCTION (this << "Loading RxspatialSigniture file " << strFilename);
306     std::ifstream singlefile;
307     std::complex<double> complexVar;
308     complex2DVector_t rxSpatialMatrix;
309     singlefile.open (strFilename.c_str (), std::ifstream::in);
310
311     NS_LOG_INFO (this << " File: " << strFilename);
312     NS_ASSERT_MSG (singlefile.good (), " RxSpatialSigniture file not found");
313
314     std::string line;
315     std::string token;
316     int counter = 1;
317     while (std::getline(singlefile, line) ) //Parse each line of the file
318     {
319         complexVector_t rxSpatialElement;
320         std::istringstream stream(line);
321         while (getline(stream,token, ',') ) //Parse each comma separated string in a line
322         {
323             complexVar = ParseComplex (token);
324             rxSpatialElement.push_back (complexVar);
325         }
326         rxSpatialMatrix.push_back (rxSpatialElement);
327         if (counter % m_pathNum == 0)
328         {
329             g_ueSpatialInstance.push_back (rxSpatialMatrix);
330             rxSpatialMatrix.clear ();
331         }
332         counter++;
333     }
334     NS_LOG_INFO ("RxspatialSigniture[instance:"<<g_ueSpatialInstance.size()<<"][path:"<<g_ueSpatialInstance[0].size()<<"][
            antennaSize:"<<g_ueSpatialInstance[0][0].size()<<"]");
335 }
336
337
338 void
339 MmWaveBeamforming::Initial(NetDeviceContainer ueDevices, NetDeviceContainer enbDevices)
340 {
341     for (NetDeviceContainer::Iterator i = ueDevices.Begin(); i != ueDevices.End(); i++)
342     {
343         for (NetDeviceContainer::Iterator j = enbDevices.Begin(); j != enbDevices.End(); j++)
344         {
345             if(m_update)
346             {
347                 SetChannelMatrix (*i,*j);
348             }
349         }
350
351     }
352
353     for (NetDeviceContainer::Iterator i = ueDevices.Begin(); i != ueDevices.End(); i++)
354     {
355
356         Ptr<MmWaveUeNetDevice> UeDev =
357                         DynamicCast<MmWaveUeNetDevice> (*i);
358         if (UeDev->GetTargetEnb ())
359         {
360             Ptr<NetDevice> targetBs = UeDev->GetTargetEnb();
361             SetBeamformingVector(*i,targetBs);
362
363         }
364     }
365     Simulator::Schedule (m_longTermUpdatePeriod, &MmWaveBeamforming::Initial,this,ueDevices,enbDevices);
366
367     /*if (!m_nextLongTermUpdate)
368     {
369         m_nextLongTermUpdate = CreateObject<ExponentialRandomVariable> ();
370         m_nextLongTermUpdate->SetAttribute ("Mean", DoubleValue (m_longTermUpdatePeriod.GetMicroSeconds ()));
371         m_nextLongTermUpdate->SetAttribute ("Bound", DoubleValue (m_longTermUpdatePeriod.GetMicroSeconds () * 10));
372     }
373     Simulator::Schedule (MicroSeconds (m_nextLongTermUpdate->GetValue()), &MmWaveBeamforming::Initial,this,ueDevices,
            enbDevices);*/
374 }
375
376
377 void
378 MmWaveBeamforming::SetChannelMatrix (Ptr<NetDevice> ueDevice, Ptr<NetDevice> enbDevice)
379 {
380     key_t key = std::make_pair(ueDevice,enbDevice);
381     int randomInstance = m_uniformRV->GetValue (0, g_numInstance-1);
382     NS_LOG_DEBUG ("************ UPDATING CHANNEL MATRIX (instance " << randomInstance << ") ************");
```

```
383
384     Ptr<BeamformingParams> bfParams = Create<BeamformingParams> ();
385     bfParams->m_enbW = g_enbAntennaInstance.at (randomInstance);
386     bfParams->m_ueW = g_ueAntennaInstance.at (randomInstance);
387     bfParams->m_channelMatrix.m_enbSpatialMatrix = g_enbSpatialInstance.at (randomInstance);
388     bfParams->m_channelMatrix.m_ueSpatialMatrix = g_ueSpatialInstance.at (randomInstance);
389     bfParams->m_channelMatrix.m_powerFraction = g_smallScaleFadingInstance.at (randomInstance);
390     bfParams->m_beam = GetLongTermFading (bfParams);
391     std::map< key_t, Ptr<BeamformingParams> >::iterator iter = m_channelMatrixMap.find(key);
392     if (iter != m_channelMatrixMap.end ())
393     {
394         m_channelMatrixMap.erase (iter);
395     }
396     m_channelMatrixMap.insert(std::make_pair(key,bfParams));
397     //update channel matrix periodically
398     //Simulator::Schedule (Seconds (m_longTermUpdatePeriod), &MmWaveBeamforming::SetChannelMatrix,this,ueDevice,enbDevice);
399 }
400
401 void
402 MmWaveBeamforming::SetBeamformingVector (Ptr<NetDevice> ueDevice, Ptr<NetDevice> enbDevice)
403 {
404     key_t key = std::make_pair(ueDevice,enbDevice);
405     std::map< key_t, Ptr<BeamformingParams> >::iterator it = m_channelMatrixMap.find(key);
406     NS_ASSERT_MSG (it != m_channelMatrixMap.end (), "could not find");
407     Ptr<BeamformingParams> bfParams = it->second;
408     Ptr<MmWaveEnbNetDevice> EnbDev =
409                 DynamicCast<MmWaveEnbNetDevice> (enbDevice);
410     Ptr<MmWaveUeNetDevice> UeDev =
411                 DynamicCast<MmWaveUeNetDevice> (ueDevice);
412
413     Ptr<AntennaArrayModel> ueAntennaArray = DynamicCast<AntennaArrayModel> (
414             UeDev->GetPhy ()->GetDlSpectrumPhy ()->GetRxAntenna ());
415     Ptr<AntennaArrayModel> enbAntennaArray = DynamicCast<AntennaArrayModel> (
416             EnbDev->GetPhy ()->GetDlSpectrumPhy ()->GetRxAntenna ());
417
418     /*double variable = m_uniformRV->GetValue (0, 1);
419     if(m_update && variable<0.08)
420     {
421         ueAntennaArray->SetBeamformingVectorWithDelay (bfParams->m_ueW);
422         enbAntennaArray->SetBeamformingVectorWithDelay (bfParams->m_enbW, ueDevice);
423
424     }
425     else
426     {*/
427         ueAntennaArray->SetBeamformingVector (bfParams->m_ueW);
428         enbAntennaArray->SetBeamformingVector (bfParams->m_enbW, ueDevice);
429
430     //}
431 }
432
433 complexVector_t*
434 MmWaveBeamforming::GetLongTermFading (Ptr<BeamformingParams> bfParams) const
435 {
436     complexVector_t* longTerm = new complexVector_t();
437     for (unsigned pathIndex = 0; pathIndex < m_pathNum; pathIndex++)
438     {
439         std::complex<double> txsum (0,0);
440         for (unsigned txAntennaIndex = 0; txAntennaIndex < m_enbAntennaSize; txAntennaIndex++)
441         {
442             txsum = txsum +
443                     bfParams->m_enbW.at (txAntennaIndex)*
444                     bfParams->m_channelMatrix.m_enbSpatialMatrix.at (pathIndex).at (txAntennaIndex);
445         }
446
447         std::complex<double> rxsum (0,0);
448         for (unsigned rxAntennaIndex = 0; rxAntennaIndex < m_ueAntennaSize; rxAntennaIndex++)
449         {
450             rxsum = rxsum +
451                     bfParams->m_ueW.at (rxAntennaIndex)*
452                     bfParams->m_channelMatrix.m_ueSpatialMatrix.at (pathIndex).at (rxAntennaIndex);
453         }
454         NS_LOG_INFO ("rxsum="<<rxsum.real ()<<" "<<rxsum.imag ());
455         longTerm->push_back (txsum*rxsum);
456     }
457     return longTerm;
458 }
459
460
461 complexVector_t*
462 MmWaveBeamforming::GetLongTermFadingOmni (Ptr<BeamformingParams> bfParams) const
463 {
```

```
464       complexVector_t* longTerm = new complexVector_t();
465       for (unsigned pathIndex = 0; pathIndex < m_pathNum; pathIndex++)
466       {
467           std::complex<double> txsum (0,0);
468           for (unsigned txAntennaIndex = 0; txAntennaIndex < m_enbAntennaSize; txAntennaIndex++)
469           {
470               txsum = txsum +
471                       bfParams->m_channelMatrix.m_enbSpatialMatrix.at (pathIndex).at (txAntennaIndex);
472           }
473
474           std::complex<double> rxsum (0,0);
475           for (unsigned rxAntennaIndex = 0; rxAntennaIndex < m_ueAntennaSize; rxAntennaIndex++)
476           {
477               rxsum = rxsum +
478                       bfParams->m_channelMatrix.m_ueSpatialMatrix.at (pathIndex).at (rxAntennaIndex);
479           }
480           NS_LOG_INFO ("rxsum="<<rxsum.real ()<<" "<<rxsum.imag ());
481           longTerm->push_back (txsum*rxsum);
482       }
483       return longTerm;
484   }
485
486   Ptr<SpectrumValue>
487   MmWaveBeamforming::GetChannelGainVector (Ptr<const SpectrumValue> txPsd, Ptr<BeamformingParams> bfParams, double speed)
          const
488   {
489
490       //std::cout << "GetChannelGainVector \n";
491       NS_LOG_FUNCTION (this);
492       Ptr<SpectrumValue> tempPsd = Copy<SpectrumValue> (txPsd);
493       if(m_fixSpeed)
494       {
495           speed = m_ueSpeed;
496       }
497
498       bool noSpeed = false;
499       if (speed == 0)
500       {
501           noSpeed = true;
502       }
503       Time time = Simulator::Now ();
504       double t = time.GetSeconds ();
505
506       Values::iterator vit = tempPsd->ValuesBegin ();
507       uint16_t iSubband = 0;
508       while (vit != tempPsd->ValuesEnd ())
509       {
510           std::complex<double> subsbandGain (0.0,0.0);
511           if ((*vit) != 0.00)
512           {
513               double fsb = m_phyMacConfig->GetCentreFrequency () - GetSystemBandwidth ()/2 + m_phyMacConfig->GetChunkWidth ()*
                      iSubband ;
514               for (unsigned int pathIndex = 0; pathIndex < m_pathNum; pathIndex++)
515               {
516                   double sigma = bfParams->m_channelMatrix.m_powerFraction.at (pathIndex);
517                   double temp_delay = -2*M_PI*fsb*DelaySpread[pathIndex];
518                   std::complex<double> delay (cos (temp_delay), sin (temp_delay));
519                   std::complex<double> doppler;
520                   if (noSpeed)
521                   {
522                       doppler = std::complex<double> (1,0);
523                   }
524                   else
525                   {
526                       double f_d = speed*m_phyMacConfig->GetCentreFrequency ()/3e8;
527                       double temp_Doppler = 2*M_PI*t*f_d*DopplerShift[pathIndex];
528
529                       doppler = std::complex<double> (cos (temp_Doppler), sin (temp_Doppler));
530                   }
531
532                   std::complex<double> smallScaleFading = m_smallScale ? sqrt(2)*sigma*doppler*delay : sqrt(2)*sigma;
533                   subsbandGain = subsbandGain + (*bfParams->m_beam).at (pathIndex)*smallScaleFading;
534               }
535               *vit = (*vit)*(norm (subsbandGain));
536               //std::cout<<"Beamforming= \n"<< std::endl;
537               //std::cout<<"Ganho beamforming="<< norm (subsbandGain) << "\n" << std::endl;
538           }
539           vit++;
540           iSubband++;
541       }
542       return tempPsd;
```

```
543
544 }
545 Ptr<SpectrumValue>
546 MmWaveBeamforming::GetChannelGainVectorOmni (Ptr<const SpectrumValue> txPsd, Ptr<BeamformingParams> bfParams, double speed)
        const
547 {
548     //std::cout << "GetChannelGainVectorOmni \n";
549     NS_LOG_FUNCTION (this);
550     Ptr<SpectrumValue> tempPsd = Copy<SpectrumValue> (txPsd);
551     if(m_fixSpeed)
552     {
553         speed = m_ueSpeed;
554     }
555
556     bool noSpeed = false;
557     if (speed == 0)
558     {
559         noSpeed = true;
560     }
561     Time time = Simulator::Now ();
562     double t = time.GetSeconds ();
563
564     Values::iterator vit = tempPsd->ValuesBegin ();
565     uint16_t iSubband = 0;
566     while (vit != tempPsd->ValuesEnd ())
567     {
568         double hnorm=0;
569         std::complex<double> subsbandGain (0.0,0.0);
570         if ((*vit) != 0.00)
571         {
572             double fsb = m_phyMacConfig->GetCentreFrequency () - GetSystemBandwidth ()/2 + m_phyMacConfig->GetChunkWidth ()*
                  iSubband ;
573             for (unsigned rxAntennaIndex = 0; rxAntennaIndex < rxAntennaSizeAlamouti; rxAntennaIndex++)
574                 {
575                     for (unsigned txAntennaIndex = 0; txAntennaIndex < txAntennaSizeAlamouti; txAntennaIndex++)
576                     {
577                         std::complex<double> hsum (0,0);
578                         for (unsigned int pathIndex = 0; pathIndex < m_pathNum; pathIndex++)
579                         {
580                             double sigma = bfParams->m_channelMatrix.m_powerFraction.at (pathIndex);
581                             double temp_delay = -2*M_PI*fsb*DelaySpread[pathIndex];
582                             std::complex<double> delay (cos (temp_delay), sin (temp_delay));
583                             std::complex<double> doppler;
584                             if (noSpeed)
585                             {
586                             doppler = std::complex<double> (1,0);
587                             }
588                             else
589                             {
590                             double f_d = speed*m_phyMacConfig->GetCentreFrequency ()/3e8;
591                             double temp_Doppler = 2*M_PI*t*f_d*DopplerShift[pathIndex];
592
593                             doppler = std::complex<double> (cos (temp_Doppler), sin (temp_Doppler));
594                             }
595
596                             std::complex<double> smallScaleFading = m_smallScale ? sqrt(2)*sigma*doppler*delay : sqrt(2)*
                                  sigma;
597                             //subsbandGain = subsbandGain + (*bfParams->m_beam).at (pathIndex)*smallScaleFading;
598
599                             //somat rio para o n mero de percursos
600                             //hsum = hsum + bfParams->m_channelMatrix.m_ueSpatialMatrix.at (pathIndex).at (rxAntennaIndex)*
601                             //bfParams->m_channelMatrix.m_enbSpatialMatrix.at (pathIndex).at (txAntennaIndex)*
                                      smallScaleFading;
602                             //std::cout << "rxAntennaSizeAlamouti=" << rxAntennaSizeAlamouti << "\n";
603                              hsum = hsum + bfParams->m_channelMatrix.m_ueSpatialMatrix.at (pathIndex).at (txAntennaIndex)*
604                              bfParams->m_channelMatrix.m_enbSpatialMatrix.at (pathIndex).at (rxAntennaIndex)*
                                      smallScaleFading;
605                         }
606                     hnorm = hnorm + norm(hsum); //soma dos quadrados de cada elemento da matriz H, norma de Frobenius.
607                         //std::cout<<"Alamouti";
608                     }
609                 }
610             //*vit = (*vit)*(norm (subsbandGain));
611             *vit = (*vit)*(hnorm)/txAntennaSizeAlamouti;
612             //std::cout<<"Ganho alamouti="<< hnorm <<"\n"<< std::endl;
613         }
614         vit++;
615         iSubband++;
616     }
617     return tempPsd;
618 }
```

```
619
620 Ptr<SpectrumValue>
621 MmWaveBeamforming::GetChannelGainVectorOmniDownlink (Ptr<const SpectrumValue> txPsd, Ptr<BeamformingParams> bfParams, double
        speed) const
622 {
623     //std::cout << "GetChannelGainVectorOmniDownlink \n";
624     NS_LOG_FUNCTION (this);
625     Ptr<SpectrumValue> tempPsd = Copy<SpectrumValue> (txPsd);
626     if(m_fixSpeed)
627     {
628         speed = m_ueSpeed;
629     }
630
631     bool noSpeed = false;
632     if (speed == 0)
633     {
634         noSpeed = true;
635     }
636     Time time = Simulator::Now ();
637     double t = time.GetSeconds ();
638
639     Values::iterator vit = tempPsd->ValuesBegin ();
640     uint16_t iSubband = 0;
641     while (vit != tempPsd->ValuesEnd ())
642     {
643         double hnorm=0;
644         std::complex<double> subsbandGain (0.0,0.0);
645         if ((*vit) != 0.00)
646         {
647             double fsb = m_phyMacConfig->GetCentreFrequency () - GetSystemBandwidth ()/2 + m_phyMacConfig->GetChunkWidth ()*
                  iSubband ;
648             for (unsigned rxAntennaIndex = 0; rxAntennaIndex < rxAntennaSizeAlamouti; rxAntennaIndex++)
649                 {
650                     for (unsigned txAntennaIndex = 0; txAntennaIndex < txAntennaSizeAlamouti; txAntennaIndex++)
651                     {
652                         std::complex<double> hsum (0,0);
653                         for (unsigned int pathIndex = 0; pathIndex < m_pathNum; pathIndex++)
654                         {
655                             double sigma = bfParams->m_channelMatrix.m_powerFraction.at (pathIndex);
656                             double temp_delay = -2*M_PI*fsb*DelaySpread[pathIndex];
657                             std::complex<double> delay (cos (temp_delay), sin (temp_delay));
658                             std::complex<double> doppler;
659                             if (noSpeed)
660                             {
661                             doppler = std::complex<double> (1,0);
662                             }
663                             else
664                             {
665                             double f_d = speed*m_phyMacConfig->GetCentreFrequency ()/3e8;
666                             double temp_Doppler = 2*M_PI*t*f_d*DopplerShift[pathIndex];
667
668                             doppler = std::complex<double> (cos (temp_Doppler), sin (temp_Doppler));
669                             }
670
671                             std::complex<double> smallScaleFading = m_smallScale ? sqrt(2)*sigma*doppler*delay : sqrt(2)*
                                  sigma;
672                             //subsbandGain = subsbandGain + (*bfParams->m_beam).at (pathIndex)*smallScaleFading;
673
674                             //somat rio para o n mero de percursos
675                             //hsum = hsum + bfParams->m_channelMatrix.m_ueSpatialMatrix.at (pathIndex).at (rxAntennaIndex)*
676                             //bfParams->m_channelMatrix.m_enbSpatialMatrix.at (pathIndex).at (txAntennaIndex)*
                                      smallScaleFading;
677
678                              hsum = hsum + bfParams->m_channelMatrix.m_ueSpatialMatrix.at (pathIndex).at (rxAntennaIndex)*
679                              bfParams->m_channelMatrix.m_enbSpatialMatrix.at (pathIndex).at (txAntennaIndex)*
                                      smallScaleFading;
680                         }
681                     hnorm = hnorm + norm(hsum); //soma dos quadrados de cada elemento da matriz H, norma de Frobenius.
682                         //std::cout<<"Alamouti";
683                     }
684                 }
685             //*vit = (*vit)*(norm (subsbandGain));
686             *vit = (*vit)*(hnorm)/txAntennaSizeAlamouti;
687             //std::cout<<"Ganho alamouti="<< hnorm <<"\n"<< std::endl;
688         }
689         vit++;
690         iSubband++;
691     }
692     return tempPsd;
693 }
694
```

```
695 Ptr<SpectrumValue>
696 MmWaveBeamforming::DoCalcRxPowerSpectralDensity (Ptr<const SpectrumValue> txPsd,
697                                                  Ptr<const MobilityModel> a,
698                                                  Ptr<const MobilityModel> b) const
699 {
700     bool downlink;
701     Ptr<NetDevice> enbDevice, ueDevice;
702
703     Ptr<NetDevice> txDevice = a->GetObject<Node> ()->GetDevice (0);
704     Ptr<NetDevice> rxDevice = b->GetObject<Node> ()->GetDevice (0);
705     Ptr<SpectrumValue> rxPsd = Copy (txPsd);
706     key_t dlkey = std::make_pair(rxDevice,txDevice);
707     key_t ulkey = std::make_pair(txDevice,rxDevice);
708
709     std::map< key_t, Ptr<BeamformingParams> >::iterator it;
710     if (m_channelMatrixMap.find(dlkey) != m_channelMatrixMap.end ())
711     {
712         // this is downlink case
713         downlink = true;
714         enbDevice = txDevice;
715         ueDevice = rxDevice;
716         it = m_channelMatrixMap.find(dlkey);
717     }
718     else if (m_channelMatrixMap.find(ulkey) != m_channelMatrixMap.end ())
719     {
720         // this is uplink case
721         downlink = false;
722         ueDevice = txDevice;
723         enbDevice = rxDevice;
724         it = m_channelMatrixMap.find(ulkey);
725     }
726     else
727     {
728         // enb to enb or ue to ue transmission, set to 0. Do no consider such scenarios.
729         return 0;
730     }
731
732     Ptr<BeamformingParams> bfParams = it->second;
733
734     Ptr<MmWaveUeNetDevice> UeDev =
735                 DynamicCast<MmWaveUeNetDevice> (ueDevice);
736     Ptr<MmWaveUePhy> uePhy = UeDev->GetPhy ();
737     Ptr<MmWaveEnbNetDevice> EnbDev =
738                 DynamicCast<MmWaveEnbNetDevice> (enbDevice);
739     Ptr<AntennaArrayModel> ueAntennaArray = DynamicCast<AntennaArrayModel> (
740             UeDev->GetPhy ()->GetDlSpectrumPhy ()->GetRxAntenna ());
741     Ptr<AntennaArrayModel> enbAntennaArray = DynamicCast<AntennaArrayModel> (
742             EnbDev->GetPhy ()->GetDlSpectrumPhy ()->GetRxAntenna ());
743
744     if (enbAntennaArray->IsOmniTx ())
745     {
746         //return rxPsd;//zml do not apply fading to omi
747
748         //std::cout<<"entrei Omni";
749         complexVector_t vec;
750         for (unsigned int i=0; i<m_pathNum; i++)
751         {
752             vec.push_back(std::complex<double> (1,0));
753         }
754         (*bfParams->m_beam) = vec;
755         //bfParams->m_beam = GetLongTermFadingOmni (bfParams);// Ganho ALAMOUTI
756     }
757     else
758     {
759         complexVector_t ueW = ueAntennaArray->GetBeamformingVector();
760         complexVector_t enbW = enbAntennaArray->GetBeamformingVector();
761
762         if (!ueW.empty() && !enbW.empty())
763         {
764             bfParams->m_ueW = ueW;
765             bfParams->m_enbW = enbW;
766             bfParams->m_beam = GetLongTermFading (bfParams);
767         }
768         else if(ueW.empty())
769         {
770             NS_LOG_ERROR ("UE beamforming vector is not configured, make sure this UE is registered to ENB");
771             *rxPsd = (*rxPsd)*0;
772             return rxPsd;
773         }
774         else if(enbW.empty())
775         {
```

```
776             NS_LOG_ERROR ("ENB beamforming vector is not configured, make sure UE is registered to this ENB");
777             *rxPsd = (*rxPsd)*0;
778             return rxPsd;
779         }
780     }
781
782     Vector rxSpeed = b->GetVelocity();
783     Vector txSpeed = a->GetVelocity();
784     double relativeSpeed = (rxSpeed.x-txSpeed.x)
785             +(rxSpeed.y-txSpeed.y)+(rxSpeed.z-txSpeed.z);
786
787     //se for Omnidirecional, calcula o bfPsd com outra fun   o
788     if (enbAntennaArray->IsOmniTx ())
789     {
790         //std::cout<<"Omni Alamouti \n";
791
792         if (downlink)
793         {
794         //std::cout << "DOWNLINK \n";
795         Ptr<SpectrumValue> bfPsd = GetChannelGainVectorOmniDownlink (rxPsd, bfParams,  relativeSpeed);
796         //std::cout << "rxAntennaSizeAlamouti=" << rxAntennaSizeAlamouti << "\n";
797         return bfPsd;
798         }
799         else
800         {
801         //  std::cout << "UPLINK \n";
802
803
804         Ptr<SpectrumValue> bfPsd = GetChannelGainVectorOmni (rxPsd, bfParams,  relativeSpeed);
805         //std::cout << "rxAntennaSizeAlamouti=" << rxAntennaSizeAlamouti << "\n";
806         return bfPsd;
807         }
808         //std::cout << "GetChannelGainVectorOmni";
809         //NS_LOG_DEBUG ((*bfPsd)/(*rxPsd));
810         //SpectrumValue bfGain = (*bfPsd)/(*rxPsd);
811         //int nbands = bfGain.GetSpectrumModel ()->GetNumBands ();
812         //std::cout<<"\t Gain Alamouti = \t" << Sum (bfGain)/nbands << "\n";
813         /*if (downlink)
814         {
815         NS_LOG_DEBUG ("DL Alamouti \t  RNTI: \t" << uePhy->GetRnti() << "\t Gain \t" << Sum (bfGain)/nbands << "\t RX PSD \t
                " << Sum(*bfPsd)/nbands); // print avg bf gain
816         }
817         else
818         {
819             //std::cout<< "UPLINK";
820         NS_LOG_DEBUG ("UL Alamouti \t  RNTI: \t" << uePhy->GetRnti() << "\t Gain \t" << Sum (bfGain)/nbands << "\t RX PSD \t
                " << Sum(*bfPsd)/nbands);
821         }*/
822
823
824
825     }
826     else
827     {
828         //std::cout<<"Beamforming \n";
829         Ptr<SpectrumValue> bfPsd = GetChannelGainVector (rxPsd, bfParams,  relativeSpeed);
830         //NS_LOG_DEBUG ((*bfPsd)/(*rxPsd));
831         SpectrumValue bfGain = (*bfPsd)/(*rxPsd);
832         int nbands = bfGain.GetSpectrumModel ()->GetNumBands ();
833         //std::cout<<"\t Gain Beamforming= \t" << Sum (bfGain)/nbands << "\n";
834         if (downlink)
835         {
836             NS_LOG_DEBUG ("DL BF \t  RNTI: \t" << uePhy->GetRnti() << "\t Gain \t" << Sum (bfGain)/nbands << "\t RX PSD \t"
                    << Sum(*bfPsd)/nbands); // print avg bf gain
837         }
838         else
839         {
840             //std::cout<< "UPLINK";
841             NS_LOG_DEBUG ("UL BF \t  RNTI: \t" << uePhy->GetRnti() << "\t Gain \t" << Sum (bfGain)/nbands << "\t RX PSD \t"
                    << Sum(*bfPsd)/nbands);
842         }
843
844         return bfPsd;
845
846     }
847
848
849     //SpectrumValue bfGain = (*bfPsd)/(*rxPsd);
850     /*int nbands = bfGain.GetSpectrumModel ()->GetNumBands ();
851
852     if (downlink)
```

```
853         {
854             NS_LOG_DEBUG ("****** DL BF gain (RNTI " << uePhy->GetRnti() << ") == " << Sum (bfGain)/nbands << " RX PSD " << Sum
                    (*rxPsd)/nbands); // print avg bf gain
855         }
856         else
857         {
858             NS_LOG_DEBUG ("****** UL BF gain (RNTI " << uePhy->GetRnti() << ") == " << Sum (bfGain)/nbands << " RX PSD " << Sum
                    (*rxPsd)/nbands);
859         }*/
860         //return bfPsd;
861     }
862
863     double
864     MmWaveBeamforming::GetSystemBandwidth () const
865     {
866         double bw = 0.00;
867         bw = m_phyMacConfig->GetChunkWidth () * m_phyMacConfig->GetNumChunkPerRb () * m_phyMacConfig->GetNumRb ();
868         return bw;
869     }
870     void
871     MmWaveBeamforming::UpdateMatrices (bool update)
872     {
873         m_update = update;
874     }
875
876     }// namespace ns3
```

Classe: mmWave-beamforming.h

```
1
2
3   /*
4    * mmWave-beamforming.h
5    *
6    *  Created on: 2014      1125
7    *      Author: menglei
8    */
9
10  #ifndef MMWAVE_BEAMFORMING_H_
11  #define MMWAVE_BEAMFORMING_H_
12
13  #include "ns3/object.h"
14  #include <ns3/spectrum-value.h>
15  #include <string.h>
16  #include "ns3/uinteger.h"
17  #include <complex>
18  #include <ns3/nstime.h>
19  #include <ns3/simple-ref-count.h>
20  #include <ns3/ptr.h>
21  #include <ns3/net-device-container.h>
22  #include <map>
23  #include <ns3/spectrum-signal-parameters.h>
24  #include <ns3/mobility-model.h>
25  #include <ns3/spectrum-propagation-loss-model.h>
26  #include <ns3/mmwave-phy-mac-common.h>
27  #include <ns3/random-variable-stream.h>
28
29
30
31  namespace ns3{
32
33  typedef std::vector<double> doubleVector_t;
34  typedef std::vector<doubleVector_t> double2DVector_t;
35
36  typedef std::vector< std::complex<double> > complexVector_t;
37  typedef std::vector<complexVector_t> complex2DVector_t;
38  typedef std::vector<complex2DVector_t> complex3DVector_t;
39  typedef std::vector<uint32_t> allocatedUeImsiVector_t;
40  typedef std::pair<Ptr<NetDevice>, Ptr<NetDevice> > key_t;
41
42  /**
43  * \store Spatial Signature and small scale fading matrices
44  */
45  struct channelMatrix : public SimpleRefCount<channelMatrix>
46  {
47      complex2DVector_t   m_enbSpatialMatrix; // enb side spatial matrix
48      complex2DVector_t   m_ueSpatialMatrix; // ue side spatial matrix
```

```
49      doubleVector_t       m_powerFraction; // store power fraction vector of 20 paths
50  };
51  /**
52  * \store beamforming vectors to calculate beamforming gain and fading
53  */
54  struct BeamformingParams : public SimpleRefCount<BeamformingParams>
55  {
56      complexVector_t      m_enbW; // enb beamforming vector
57      complexVector_t      m_ueW; // ue beamforming vector
58      channelMatrix        m_channelMatrix;
59      complexVector_t*     m_beam; // product of beamforming vectors and spatial matrices
60  };
61
62  /**
63  * \ingroup mmWave
64  * \MmWaveBeamforming models the beamforming gain and fading distortion in frequency and time for the mmWave channel
65  */
66  class MmWaveBeamforming : public SpectrumPropagationLossModel
67  {
68  public:
69      /**
70      * \breif Set the pathNum and load files that store beamforming vector
71      * \param enbAntenna antenna number of enb
72      * \param ueAntenna antenna number of ue
73      */
74      MmWaveBeamforming (uint32_t enbAntenna, uint32_t ueAntenna);
75      virtual ~MmWaveBeamforming();
76
77      static TypeId GetTypeId (void);
78      void DoDispose();
79      void SetCofigurationParameters (Ptr<MmWavePhyMacCommon> ptrConfig);
80      Ptr<MmWavePhyMacCommon> GetConfigurationParameters (void) const;
81
82      void LoadFile();
83      /**
84      * \breif Set the channel matrix for each link
85      * \param ueDevices a pointer to ueNetDevice container
86      * \param enbDevices a pointer to enbNetDevice container
87      */
88      void Initial(NetDeviceContainer ueDevices, NetDeviceContainer enbDevices);
89
90      void UpdateMatrices (bool update);
91
92  private:
93      /**
94      * \breif Get complex number from a string
95      * \param strCmplx a string store complex bumber i.e. 3+2i,
96      * \return a complex number of the string
97      */
98      std::complex<double> ParseComplex (std::string strCmplx);
99      /**
100     * \breif Load file which store small scale fading sigma vector
101     */
102     void LoadSmallScaleFading ();
103     /**
104     * \breif Load file which store antenna weights for enb
105     */
106     void LoadEnbAntenna ();
107     /**
108     * \breif Load file which store antenna weights for ue
109     */
110     void LoadUeAntenna ();
111     /**
112     * \breif Load file which store spatial signature matrix for  enb
113     */
114     void LoadEnbSpatialSignature ();
115     /**
116     * \breif Load file which store spatial signature matrix for  ue
117     */
118     void LoadUeSpatialSignature ();
119     /**
120     * \breif Calculate beamforming gain and fading distortion in frequency and time
121     * \param txPsd set of values vs frequency representing the
122     *              transmission power. See SpectrumChannel for details.
123     * \param a sender mobility
124     * \param b receiver mobility
125     * \return set of values vs frequency representing the received
126     *         power in the same units used for the txPsd parameter.
127     */
128     Ptr<SpectrumValue> DoCalcRxPowerSpectralDensity (Ptr<const SpectrumValue> txPsd,
129                                                      Ptr<const MobilityModel> a,
```

```
130                                                 Ptr<const MobilityModel> b) const;
131     /**
132     * \breif Set the beamforming vector of connected enbs and ues
133     * \param ueDevice a pointer to ueNetDevice
134     * \param enbDevice a pointer to enbNetDevice
135     */
136     void SetBeamformingVector (Ptr<NetDevice> ueDevice, Ptr<NetDevice> enbDevice);
137     /**
138     * \breif Store the channel matrix to channelMatrixMap
139     * \param ueDevice a pointer to ueNetDevice
140     * \param enbDevice a pointer to enbNetDevice
141     */
142     void SetChannelMatrix(Ptr<NetDevice> ueDevice, Ptr<NetDevice> enbDevice);
143     /**
144     * \breif Calculate the system bandwidth using
145     *        the user defined parameters
146     * \return value of the system abndwith
147     */
148     double GetSystemBandwidth () const;
149     /**
150     * \breif Calculate long term fading
151     * \param bfParas a pointer to beamforming vectors
152     * \return complex vector of gain
153     */
154     complexVector_t* GetLongTermFading (Ptr<BeamformingParams> bfParams) const;
155     complexVector_t* GetLongTermFadingOmni (Ptr<BeamformingParams> bfParams) const;
156     /**
157     * \breif calculate power spectrum density considering beamformign and fading
158     * \param bfParas a pointer to beamforming vectors
159     * \param Psd set of values vs frequency representing the
160     *              transmission power. See SpectrumChannel for details
161     * \param speed a double value to relative speed of tx and rx
162     * \return cset of values vs frequency representing the received
163     *         power in the same units used for the txPsd parameter.
164     */
165     Ptr<SpectrumValue> GetChannelGainVector (Ptr<const SpectrumValue> txPsd, Ptr<BeamformingParams> bfParams, double speed)
            const;
166     Ptr<SpectrumValue> GetChannelGainVectorOmni (Ptr<const SpectrumValue> txPsd, Ptr<BeamformingParams> bfParams, double
            speed) const;
167     Ptr<SpectrumValue> GetChannelGainVectorOmniDownlink (Ptr<const SpectrumValue> txPsd, Ptr<BeamformingParams> bfParams,
            double speed) const;
168     /**
169     * \a map to store channel matrix
170     * key pair<NetDevice,NetDevice> a pair of pointer to NetDevice present enb and ue for downlink
171     */
172     mutable std::map< key_t, Ptr<BeamformingParams> > m_channelMatrixMap;
173
174     uint32_t m_pathNum;
175     uint32_t m_enbAntennaSize;
176     uint32_t m_ueAntennaSize;
177 //  double m_subbandWidth;
178 //  double m_centreFrequency;
179 //  uint32_t m_numResourceBlocks;
180 //  uint32_t m_numSubbbandPerRB;
181
182 private:
183     Ptr<MmWavePhyMacCommon> m_phyMacConfig;
184     Time m_longTermUpdatePeriod;
185     bool m_smallScale;
186     bool m_fixSpeed;  // used for SINR sweep test
187     double m_ueSpeed;
188     bool m_update;
189     Ptr<UniformRandomVariable> m_uniformRV;
190
191     //Ptr<ExponentialRandomVariable> m_nextLongTermUpdate;  // next update of long term statistics in microseconds
192 };
193
194 }  //namespace ns3
195
196
197 #endif /* MMWAVE_BEAMFORMING_H_ */
```

Classe: mmwave-enb-phy.cc

```
1
2 /*
3  * mmwave-enb-phy.cc
```

```
 *
 *  Created on: Nov 5, 2014
 *      Author: sourjya
 */

#include <ns3/object-factory.h>
#include <ns3/log.h>
#include <cfloat>
#include <cmath>
#include <ns3/simulator.h>
#include <ns3/attribute-accessor-helper.h>
#include <ns3/double.h>

#include "mmwave-enb-phy.h"
#include "mmwave-ue-phy.h"
#include "mmwave-net-device.h"
#include "mmwave-ue-net-device.h"
#include "mmwave-spectrum-value-helper.h"
#include "mmwave-radio-bearer-tag.h"

#include <ns3/node-list.h>
#include <ns3/node.h>
#include <ns3/pointer.h>

namespace ns3{

NS_LOG_COMPONENT_DEFINE ("MmWaveEnbPhy");

NS_OBJECT_ENSURE_REGISTERED (MmWaveEnbPhy);

MmWaveEnbPhy::MmWaveEnbPhy ()
{
    NS_LOG_FUNCTION (this);
    NS_FATAL_ERROR ("This constructor should not be called");
}

MmWaveEnbPhy::MmWaveEnbPhy (Ptr<MmWaveSpectrumPhy> dlPhy, Ptr<MmWaveSpectrumPhy> ulPhy)
    :MmWavePhy (dlPhy, ulPhy),
    m_prevSlot (0),
  m_prevSlotDir (SlotAllocInfo::NA),
  m_currSymStart (0)
{
    m_enbCphySapProvider = new MemberLteEnbCphySapProvider<MmWaveEnbPhy> (this);
    Simulator::ScheduleNow (&MmWaveEnbPhy::StartSubFrame, this);
}

MmWaveEnbPhy::~MmWaveEnbPhy ()
{

}

TypeId
MmWaveEnbPhy::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::MmWaveEnbPhy")
    .SetParent<MmWavePhy> ()
    .AddConstructor<MmWaveEnbPhy> ()
    .AddAttribute ("TxPower",
                   "Transmission power in dBm",
                   DoubleValue (30.0),
                   MakeDoubleAccessor (&MmWaveEnbPhy::SetTxPower,
                                       &MmWaveEnbPhy::GetTxPower),
                   MakeDoubleChecker<double> ())
    .AddAttribute ("NoiseFigure",
                   "Loss (dB) in the Signal-to-Noise-Ratio due to non-idealities in the receiver."
                   " According to Wikipedia (http://en.wikipedia.org/wiki/Noise_figure), this is "
                   "\"the difference in decibels (dB) between"
                   " the noise output of the actual receiver to the noise output of an "
                   " ideal receiver with the same overall gain and bandwidth when the receivers "
                   " are connected to sources at the standard noise temperature T0.\" "
                   "In this model, we consider T0 = 290K.",
                   DoubleValue (5.0),
                   MakeDoubleAccessor (&MmWavePhy::SetNoiseFigure,
                                       &MmWavePhy::GetNoiseFigure),
                   MakeDoubleChecker<double> ())
     .AddAttribute ("DlSpectrumPhy",
                    "The downlink MmWaveSpectrumPhy associated to this MmWavePhy",
                    TypeId::ATTR_GET,
                    PointerValue (),
                    MakePointerAccessor (&MmWaveEnbPhy::GetDlSpectrumPhy),
                    MakePointerChecker <MmWaveSpectrumPhy> ())
```

```
85          .AddAttribute ("UlSpectrumPhy",
86                         "The uplink MmWaveSpectrumPhy associated to this MmWavePhy",
87                         TypeId::ATTR_GET,
88                         PointerValue (),
89                         MakePointerAccessor (&MmWaveEnbPhy::GetUlSpectrumPhy),
90                         MakePointerChecker <MmWaveSpectrumPhy> ())
91          .AddTraceSource ("UlSinrTrace",
92                           "UL SINR statistics.",
93                           MakeTraceSourceAccessor (&MmWaveEnbPhy::m_ulSinrTrace),
94                           "ns3::UlSinr::TracedCallback")
95
96        ;
97      return tid;
98
99    }
100
101   void
102   MmWaveEnbPhy::DoInitialize (void)
103   {
104       NS_LOG_FUNCTION (this);
105       Ptr<SpectrumValue> noisePsd = MmWaveSpectrumValueHelper::CreateNoisePowerSpectralDensity (m_phyMacConfig, m_noiseFigure)
              ;
106       m_downlinkSpectrumPhy->SetNoisePowerSpectralDensity (noisePsd);
107     //m_numRbg = m_phyMacConfig->GetNumRb() / m_phyMacConfig->GetNumRbPerRbg();
108       //m_ctrlPeriod = NanoSeconds (1000 * m_phyMacConfig->GetCtrlSymbols() * m_phyMacConfig->GetSymbolPeriod());
109       //m_dataPeriod = NanoSeconds (1000 * (m_phyMacConfig->GetSymbPerSlot() - m_phyMacConfig->GetCtrlSymbols()) *
              m_phyMacConfig->GetSymbolPeriod());
110
111       for (unsigned i = 0; i < m_phyMacConfig->GetL1L2CtrlLatency(); i++)
112       { // push elements onto queue for initial scheduling delay
113           m_controlMessageQueue.push_back (std::list<Ptr<MmWaveControlMessage> > ());
114       }
115       //m_sfAllocInfoUpdated = true;
116
117       for (unsigned i = 0; i < m_phyMacConfig->GetTotalNumChunk(); i++)
118       {
119           m_channelChunks.push_back(i);
120       }
121       SetSubChannels(m_channelChunks);
122
123       m_sfPeriod = NanoSeconds (1000.0 * m_phyMacConfig->GetSubframePeriod ());
124
125       for (unsigned i = 0; i < m_phyMacConfig->GetSubframesPerFrame(); i++)
126       {
127           m_sfAllocInfo.push_back (SfAllocInfo(SfnSf (m_frameNum, i, 0)));
128           SlotAllocInfo dlCtrlSlot;
129           dlCtrlSlot.m_slotType = SlotAllocInfo::CTRL;
130           dlCtrlSlot.m_numCtrlSym = 1;
131           dlCtrlSlot.m_tddMode = SlotAllocInfo::DL;
132           dlCtrlSlot.m_dci.m_numSym = 1;
133           dlCtrlSlot.m_dci.m_symStart = 0;
134           SlotAllocInfo ulCtrlSlot;
135           ulCtrlSlot.m_slotType = SlotAllocInfo::CTRL;
136           ulCtrlSlot.m_numCtrlSym = 1;
137           ulCtrlSlot.m_tddMode = SlotAllocInfo::UL;
138           ulCtrlSlot.m_slotIdx = 0xFF;
139           ulCtrlSlot.m_dci.m_numSym = 1;
140           ulCtrlSlot.m_dci.m_symStart = m_phyMacConfig->GetSymbolsPerSubframe()-1;
141           m_sfAllocInfo[i].m_slotAllocInfo.push_back (dlCtrlSlot);
142           m_sfAllocInfo[i].m_slotAllocInfo.push_back (ulCtrlSlot);
143       }
144
145       MmWavePhy::DoInitialize ();
146   }
147   void
148   MmWaveEnbPhy::DoDispose (void)
149   {
150
151   }
152
153   void
154   MmWaveEnbPhy::SetmmWaveEnbCphySapUser (LteEnbCphySapUser* s)
155   {
156     NS_LOG_FUNCTION (this);
157     m_enbCphySapUser = s;
158   }
159
160   LteEnbCphySapProvider*
161   MmWaveEnbPhy::GetmmWaveEnbCphySapProvider ()
162   {
163     NS_LOG_FUNCTION (this);
```

```
164     return m_enbCphySapProvider;
165   }
166
167   void
168   MmWaveEnbPhy::SetTxPower (double pow)
169   {
170       m_txPower = pow;
171   }
172   double
173   MmWaveEnbPhy::GetTxPower () const
174   {
175       return m_txPower;
176   }
177
178   void
179   MmWaveEnbPhy::SetNoiseFigure (double nf)
180   {
181       m_noiseFigure = nf;
182   }
183   double
184   MmWaveEnbPhy::GetNoiseFigure () const
185   {
186       return m_noiseFigure;
187   }
188
189   void
190   MmWaveEnbPhy::CalcChannelQualityForUe (std::vector <double> sinr, Ptr<MmWaveSpectrumPhy> ue)
191   {
192
193   }
194
195   Ptr<SpectrumValue>
196   MmWaveEnbPhy::CreateTxPowerSpectralDensity ()
197   {
198       Ptr<SpectrumValue> psd =
199               MmWaveSpectrumValueHelper::CreateTxPowerSpectralDensity (m_phyMacConfig, m_txPower, m_listOfSubchannels );
200       return psd;
201   }
202
203   void
204   MmWaveEnbPhy::DoSetSubChannels ()
205   {
206
207   }
208
209   void
210   MmWaveEnbPhy::SetSubChannels (std::vector<int> mask )
211   {
212       m_listOfSubchannels = mask;
213       Ptr<SpectrumValue> txPsd = CreateTxPowerSpectralDensity ();
214       NS_ASSERT (txPsd);
215       //std::cout << "eNB transmitindo \n";
216       m_downlinkSpectrumPhy->SetTxPowerSpectralDensity (txPsd);
217   }
218
219   Ptr<MmWaveSpectrumPhy>
220   MmWaveEnbPhy::GetDlSpectrumPhy () const
221   {
222     return m_downlinkSpectrumPhy;
223   }
224
225   Ptr<MmWaveSpectrumPhy>
226   MmWaveEnbPhy::GetUlSpectrumPhy () const
227   {
228     return m_uplinkSpectrumPhy;
229   }
230
231   void
232   MmWaveEnbPhy::StartSubFrame (void)
233   {
234       //std::cout<< "MmWaveEnbPhy::StartSubFrame \n";
235       NS_LOG_FUNCTION (this);
236
237       m_lastSfStart = Simulator::Now();
238
239       m_currSfAllocInfo = m_sfAllocInfo[m_sfNum];
240       //m_currSfNumSlots = m_currSfAllocInfo.m_dlSlotAllocInfo.size () + m_currSfAllocInfo.m_ulSlotAllocInfo.size ();
241       m_currSfNumSlots = m_currSfAllocInfo.m_slotAllocInfo.size ();
242
243       NS_ASSERT ((m_currSfAllocInfo.m_sfnSf.m_frameNum == m_frameNum) &&
244                  (m_currSfAllocInfo.m_sfnSf.m_sfNum == m_sfNum));
```

```
245
246     if (m_sfNum == 0)        // send MIB at the beginning of each frame
247     {
248         LteRrcSap::MasterInformationBlock mib;
249         mib.dlBandwidth = (uint8_t)4;
250         mib.systemFrameNumber = 1;
251         Ptr<MmWaveMibMessage> mibMsg = Create<MmWaveMibMessage> ();
252         mibMsg->SetMib(mib);
253         if (m_controlMessageQueue.empty())
254         {
255             std::list<Ptr<MmWaveControlMessage> > l;
256             m_controlMessageQueue.push_back (l);
257         }
258         m_controlMessageQueue.at (0).push_back (mibMsg);
259     }
260     else if (m_sfNum == 5)  // send SIB at beginning of second half-frame
261     {
262         Ptr<MmWaveSib1Message> msg = Create<MmWaveSib1Message> ();
263         msg->SetSib1 (m_sib1);
264         m_controlMessageQueue.at (0).push_back (msg);
265     }
266
267     StartSlot();
268 }
269
270 void
271 MmWaveEnbPhy::StartSlot (void)
272 {
273     //assume the control signal is omi
274     //std::cout << "MmWaveEnbPhy::StartSlot \n";
275     Ptr<AntennaArrayModel> antennaArray = DynamicCast<AntennaArrayModel> (GetDlSpectrumPhy ()->GetRxAntenna());
276     antennaArray->ChangeToOmniTx ();
277
278     NS_LOG_FUNCTION (this);
279
280     SlotAllocInfo currSlot;
281
282     /*uint8_t slotInd = 0;
283     if (m_slotNum >= m_currSfAllocInfo.m_dlSlotAllocInfo.size ())
284     {
285         if (m_currSfAllocInfo.m_ulSlotAllocInfo.size () > 0)
286         {
287             slotInd = m_slotNum - m_currSfAllocInfo.m_dlSlotAllocInfo.size ();
288             currSlot = m_currSfAllocInfo.m_ulSlotAllocInfo[slotInd];
289             m_currSymStart = currSlot.m_dci.m_symStart;
290         }
291     }
292     else
293     {
294         if (m_currSfAllocInfo.m_ulSlotAllocInfo.size () > 0)
295         {
296             slotInd = m_slotNum;
297             currSlot = m_currSfAllocInfo.m_dlSlotAllocInfo[slotInd];
298             m_currSymStart = currSlot.m_dci.m_symStart;
299         }
300     }*/
301
302     //slotInd = m_slotNum;
303     currSlot = m_currSfAllocInfo.m_slotAllocInfo[m_slotNum];
304     m_currSymStart = currSlot.m_dci.m_symStart;
305
306     SfnSf sfn = SfnSf (m_frameNum, m_sfNum, m_slotNum);
307   m_harqPhyModule->SubframeIndication (sfn);  // trigger HARQ module
308
309   std::list <Ptr<MmWaveControlMessage > > dciMsgList;
310
311     Time guardPeriod;
312     Time slotPeriod;
313
314     if(m_slotNum == 0) // DL control slot
315     {
316         //std::cout << "ENB-PHY DL m_slotNum == 0 \n";
317         // get control messages to be transmitted in DL-Control period
318         std::list <Ptr<MmWaveControlMessage > > ctrlMsgs = GetControlMessages ();
319         //std::list <Ptr<MmWaveControlMessage > >::iterator it = ctrlMsgs.begin ();
320         // find all DL/UL DCI elements and create DCI messages to be transmitted in DL control period
321         for (unsigned islot = 0; islot < m_currSfAllocInfo.m_slotAllocInfo.size (); islot++)
322         {
323             if (m_currSfAllocInfo.m_slotAllocInfo[islot].m_slotType != SlotAllocInfo::CTRL &&
324                     m_currSfAllocInfo.m_slotAllocInfo[islot].m_tddMode == SlotAllocInfo::DL)
325             {
```

```
326                 DciInfoElementTdma &dciElem = m_currSfAllocInfo.m_slotAllocInfo[islot].m_dci;
327                 NS_ASSERT (dciElem.m_format == DciInfoElementTdma::DL);
328                 if (dciElem.m_tbSize > 0)
329                 {
330                     Ptr<MmWaveTdmaDciMessage> dciMsg = Create<MmWaveTdmaDciMessage> ();
331                     dciMsg->SetDciInfoElement (dciElem);
332                     dciMsg->SetSfnSf (sfn);
333                     dciMsgList.push_back (dciMsg);
334                     ctrlMsgs.push_back (dciMsg);
335                 }
336             }
337         }
338 
339         unsigned ulSfNum = (m_sfNum + m_phyMacConfig->GetUlSchedDelay ()) % m_phyMacConfig->GetSubframesPerFrame ();
340         for (unsigned islot = 0; islot < m_sfAllocInfo[ulSfNum].m_slotAllocInfo.size (); islot++)
341         {
342             if (m_sfAllocInfo[ulSfNum].m_slotAllocInfo[islot].m_slotType != SlotAllocInfo::CTRL
343                     && m_sfAllocInfo[ulSfNum].m_slotAllocInfo[islot].m_tddMode == SlotAllocInfo::UL)
344             {
345                 DciInfoElementTdma &dciElem = m_sfAllocInfo[ulSfNum].m_slotAllocInfo[islot].m_dci;
346                 NS_ASSERT (dciElem.m_format == DciInfoElementTdma::UL);
347                 if (dciElem.m_tbSize > 0)
348                 {
349                     Ptr<MmWaveTdmaDciMessage> dciMsg = Create<MmWaveTdmaDciMessage> ();
350                     dciMsg->SetDciInfoElement (dciElem);
351                     dciMsg->SetSfnSf (sfn);
352                     //dciMsgList.push_back (dciMsg);
353                     ctrlMsgs.push_back (dciMsg);
354                 }
355             }
356         }
357 
358         // TX control period
359         slotPeriod = NanoSeconds (1000.0*m_phyMacConfig->GetSymbolPeriod ()*m_phyMacConfig->GetDlCtrlSymbols());
360         //Acrescentei essa chamada da fun   o AddExpectedTb
361         //**********
362         Ptr<PacketBurst> pktBurst = GetPacketBurst (SfnSf (m_frameNum, m_sfNum, currSlot.m_dci.m_symStart));
363         if(pktBurst && pktBurst->GetNPackets() > 0)
364         {
365             std::list< Ptr<Packet> > pkts = pktBurst->GetPackets ();
366             MmWaveMacPduTag macTag;
367             pkts.front ()->PeekPacketTag (macTag);
368             NS_ASSERT ((macTag.GetSfn().m_sfNum == m_sfNum) && (macTag.GetSfn().m_slotNum == currSlot.m_dci.m_symStart));
369         }
370         else
371         {
372             // sometimes the UE will be scheduled when no data is queued
373             // in this case, send an empty PDU
374             MmWaveMacPduTag tag (SfnSf(m_frameNum, m_sfNum, currSlot.m_dci.m_symStart));
375             Ptr<Packet> emptyPdu = Create <Packet> ();
376             MmWaveMacPduHeader header;
377             MacSubheader subheader (3, 0);  // lcid = 3, size = 0
378             header.AddSubheader (subheader);
379             emptyPdu->AddHeader (header);
380             emptyPdu->AddPacketTag (tag);
381             LteRadioBearerTag bearerTag (currSlot.m_dci.m_rnti, 3, 0);
382             emptyPdu->AddPacketTag (bearerTag);
383             pktBurst = CreateObject<PacketBurst> ();
384             pktBurst->AddPacket (emptyPdu);
385         }
386         //*****************
387         NS_LOG_DEBUG ("ENB TXing DL CTRL frame " << m_frameNum << " subframe " << (unsigned)m_sfNum << " symbols "
388                       << (unsigned)currSlot.m_dci.m_symStart << "-" << (unsigned)(currSlot.m_dci.m_symStart+currSlot.m_dci.
                              m_numSym-1)
389                       << "\t start " << Simulator::Now() << " end " << Simulator::Now() + slotPeriod-NanoSeconds(1.0));
390         SendCtrlChannels(pktBurst, ctrlMsgs, slotPeriod-NanoSeconds(1.0), currSlot); // -1 ns ensures control ends before
                data period
391     }
392     else if (m_slotNum == m_currSfNumSlots-1) // UL control slot
393     {
394         //std::cout << "ENB-PHY UL m_slotNum =! 0 \n";
395         slotPeriod = NanoSeconds (1000.0*m_phyMacConfig->GetSymbolPeriod ()*m_phyMacConfig->GetUlCtrlSymbols());
396         //Acrescentei essa chamada da fun   o AddExpectedTb
397         //************
398         m_downlinkSpectrumPhy->AddExpectedTb (currSlot.m_dci.m_rnti, currSlot.m_dci.m_ndi, currSlot.m_dci.m_tbSize, currSlot
                .m_dci.m_mcs,
399                                               m_channelChunks, currSlot.m_dci.m_harqProcess, currSlot.m_dci.m_rv, true,
400                                               currSlot.m_dci.m_symStart, currSlot.m_dci.m_numSym);
401         //**************
402         NS_LOG_DEBUG ("ENB RXing UL CTRL frame " << m_frameNum << " subframe " << (unsigned)m_sfNum << " symbols "
403                       << (unsigned)currSlot.m_dci.m_symStart << "-" << (unsigned)(currSlot.m_dci.m_symStart+currSlot.m_dci.
```

```
                                m_numSym-1)
404                                 << "\t start " << Simulator::Now() << " end " << Simulator::Now() + slotPeriod);
405       }
406       else if (currSlot.m_tddMode == SlotAllocInfo::DL)      // transmit DL slot
407       {
408           slotPeriod = NanoSeconds (1000.0 * m_phyMacConfig->GetSymbolPeriod() * currSlot.m_dci.m_numSym);
409           NS_ASSERT (currSlot.m_tddMode == SlotAllocInfo::DL);
410           //NS_LOG_DEBUG ("Slot " << m_slotNum << " scheduled for Downlink");
411           //          if (m_prevSlotDir == SlotAllocInfo::UL)  // if curr slot == DL and prev slot == UL
412           //          {
413           //              guardPeriod = NanoSeconds (1000.0 * m_phyMacConfig->GetGuardPeriod ());
414           //          }
415           Ptr<PacketBurst> pktBurst = GetPacketBurst (SfnSf (m_frameNum, m_sfNum, currSlot.m_dci.m_symStart));
416           if(pktBurst && pktBurst->GetNPackets() > 0)
417           {
418               std::list< Ptr<Packet> > pkts = pktBurst->GetPackets ();
419               MmWaveMacPduTag macTag;
420               pkts.front ()->PeekPacketTag (macTag);
421               NS_ASSERT ((macTag.GetSfn().m_sfNum == m_sfNum) && (macTag.GetSfn().m_slotNum == currSlot.m_dci.m_symStart));
422           }
423           else
424           {
425               // sometimes the UE will be scheduled when no data is queued
426               // in this case, send an empty PDU
427               MmWaveMacPduTag tag (SfnSf(m_frameNum, m_sfNum, currSlot.m_dci.m_symStart));
428               Ptr<Packet> emptyPdu = Create <Packet> ();
429               MmWaveMacPduHeader header;
430               MacSubheader subheader (3, 0);  // lcid = 3, size = 0
431               header.AddSubheader (subheader);
432               emptyPdu->AddHeader (header);
433               emptyPdu->AddPacketTag (tag);
434               LteRadioBearerTag bearerTag (currSlot.m_dci.m_rnti, 3, 0);
435               emptyPdu->AddPacketTag (bearerTag);
436               pktBurst = CreateObject<PacketBurst> ();
437               pktBurst->AddPacket (emptyPdu);
438           }
439           NS_LOG_DEBUG ("ENB TXing DL DATA frame " << m_frameNum << " subframe " << (unsigned)m_sfNum << " symbols "
440                         << (unsigned)currSlot.m_dci.m_symStart << "-" << (unsigned)(currSlot.m_dci.m_symStart+currSlot.m_dci.
                                  m_numSym-1)
441                         << "\t start " << Simulator::Now()+NanoSeconds(1.0) << " end " << Simulator::Now() + slotPeriod-
                                  NanoSeconds (2.0));
442           Simulator::Schedule (NanoSeconds(1.0), &MmWaveEnbPhy::SendDataChannels, this, pktBurst, slotPeriod-NanoSeconds (2.0)
                  , currSlot);
443       }
444       else if (currSlot.m_tddMode == SlotAllocInfo::UL)  // receive UL slot
445       {
446           //std::cout << "Entrou no if \n";
447           slotPeriod = NanoSeconds (1000.0 * m_phyMacConfig->GetSymbolPeriod() * currSlot.m_dci.m_numSym);
448           //NS_LOG_DEBUG ("Slot " << (uint8_t)m_slotNum << " scheduled for Uplink");
449           //std::cout << "AddExpectedTb - eNB \n";
450           m_downlinkSpectrumPhy->AddExpectedTb(currSlot.m_dci.m_rnti, currSlot.m_dci.m_ndi, currSlot.m_dci.m_tbSize,
451                                                currSlot.m_dci.m_mcs, m_channelChunks, currSlot.m_dci.m_harqProcess, currSlot.
                                                      m_dci.m_rv, false,
452                                                currSlot.m_dci.m_symStart, currSlot.m_dci.m_numSym);
453
454           for (uint8_t i = 0; i < m_deviceMap.size (); i++)
455           {
456               Ptr<MmWaveUeNetDevice> ueDev = DynamicCast<MmWaveUeNetDevice> (m_deviceMap.at (i));
457               uint64_t ueRnti = ueDev->GetPhy ()->GetRnti ();
458               //NS_LOG_UNCOND ("Scheduled rnti:"<<rnti <<" ue rnti:"<< ueRnti);
459               if (currSlot.m_rnti == ueRnti)
460               {
461                   //NS_LOG_UNCOND ("Change Beamforming Vector");
462                   Ptr<AntennaArrayModel> antennaArray = DynamicCast<AntennaArrayModel> (GetDlSpectrumPhy ()->GetRxAntenna());
463                   antennaArray->ChangeBeamformingVector (m_deviceMap.at (i));
464                   break;
465               }
466           }
467
468           NS_LOG_DEBUG ("ENB RXing UL DATA frame " << m_frameNum << " subframe " << (unsigned)m_sfNum << " symbols "
469                         << (unsigned)currSlot.m_dci.m_symStart << "-" << (unsigned)(currSlot.m_dci.m_symStart+currSlot.m_dci.
                                  m_numSym-1)
470                         << "\t start " << Simulator::Now() << " end " << Simulator::Now() + slotPeriod );
471       }
472
473     m_prevSlotDir = currSlot.m_tddMode;
474
475       m_phySapUser->SubframeIndication (SfnSf (m_frameNum, m_sfNum, m_slotNum));  // trigger MAC
476
477       Simulator::Schedule (slotPeriod, &MmWaveEnbPhy::EndSlot, this);
478   }
```

```
479
480 void
481 MmWaveEnbPhy::EndSlot (void)
482 {
483     NS_LOG_FUNCTION (this << Simulator::Now ().GetSeconds ());
484
485     Ptr<AntennaArrayModel> antennaArray = DynamicCast<AntennaArrayModel> (GetDlSpectrumPhy ()->GetRxAntenna());
486     antennaArray->ChangeToOmniTx ();
487
488     if (m_slotNum == m_currSfNumSlots-1)
489     {
490         m_slotNum = 0;
491         EndSubFrame ();
492     }
493     else
494     {
495         Time nextSlotStart;
496         //uint8_t slotInd = m_slotNum+1;
497         /*if (slotInd >= m_currSfAllocInfo.m_slotAllocInfo.size ())
498         {
499             if (m_currSfAllocInfo.m_slotAllocInfo.size () > 0)
500             {
501                 slotInd = slotInd - m_currSfAllocInfo.m_slotAllocInfo.size ();
502                 nextSlotStart = NanoSeconds (1000.0 * m_phyMacConfig->GetSymbolPeriod () *
503                                              m_currSfAllocInfo.m_ulSlotAllocInfo[slotInd].m_dci.m_symStart);
504             }
505         }
506         else
507         {
508             if (m_currSfAllocInfo.m_slotAllocInfo.size () > 0)
509             {
510                 nextSlotStart = NanoSeconds (1000.0 * m_phyMacConfig->GetSymbolPeriod () *
511                                              m_currSfAllocInfo.m_slotAllocInfo[slotInd].m_dci.m_symStart);
512             }
513         }*/
514         m_slotNum++;
515         nextSlotStart = NanoSeconds (1000.0 * m_phyMacConfig->GetSymbolPeriod () *
516                                                   m_currSfAllocInfo.m_slotAllocInfo[m_slotNum].m_dci.m_symStart);
517         Simulator::Schedule (nextSlotStart+m_lastSfStart-Simulator::Now(), &MmWaveEnbPhy::StartSlot, this);
518     }
519 }
520
521 void
522 MmWaveEnbPhy::EndSubFrame (void)
523 {
524     NS_LOG_FUNCTION (this << Simulator::Now ().GetSeconds ());
525
526     Time sfStart = m_lastSfStart + m_sfPeriod - Simulator::Now();
527     m_slotNum = 0;
528     if (m_sfNum == m_phyMacConfig->GetSubframesPerFrame ()-1)
529     {
530         m_sfNum = 0;
531 //      if (m_frameNum == 1023)
532 //      {
533 //          m_frameNum = 0;
534 //      }
535 //      else
536 //      {
537 //          m_frameNum++;
538 //      }
539         m_frameNum++;
540     }
541     else
542     {
543         m_sfNum++;
544     }
545
546     Simulator::Schedule (sfStart, &MmWaveEnbPhy::StartSubFrame, this);
547 }
548
549 void
550 MmWaveEnbPhy::SendDataChannels (Ptr<PacketBurst> pb, Time slotPrd, SlotAllocInfo& slotInfo)
551 {
552     if (slotInfo.m_isOmni)
553     {
554         Ptr<AntennaArrayModel> antennaArray = DynamicCast<AntennaArrayModel> (GetDlSpectrumPhy ()->GetRxAntenna());
555         antennaArray->ChangeToOmniTx ();
556     }
557     else
558     { // update beamforming vectors (currently supports 1 user only)
559         //std::map<uint16_t, std::vector<unsigned> >::iterator ueRbIt = slotInfo.m_ueRbMap.begin();
```

```
560             //uint16_t rnti = ueRbIt->first;
561             for (uint8_t i = 0; i < m_deviceMap.size (); i++)
562             {
563                 Ptr<MmWaveUeNetDevice> ueDev = DynamicCast<MmWaveUeNetDevice> (m_deviceMap.at (i));
564                 uint64_t ueRnti = ueDev->GetPhy ()->GetRnti ();
565                 //NS_LOG_UNCOND ("Scheduled rnti:"<<rnti <<" ue rnti:"<< ueRnti);
566                 if (slotInfo.m_dci.m_rnti == ueRnti)
567                 {
568                     //NS_LOG_UNCOND ("Change Beamforming Vector");
569                     Ptr<AntennaArrayModel> antennaArray = DynamicCast<AntennaArrayModel> (GetDlSpectrumPhy ()->GetRxAntenna());
570                     antennaArray->ChangeBeamformingVector (m_deviceMap.at (i));
571                     break;
572                 }
573
574             }
575         }
576
577         /*
578         if (!slotInfo.m_isOmni && !slotInfo.m_ueRbMap.empty ())
579         {
580             Ptr<AntennaArrayModel> antennaArray = DynamicCast<AntennaArrayModel> (GetDlSpectrumPhy ()->GetRxAntenna());
581              //set beamforming vector;
582              //for ENB, you can choose 64 antenna with 0-15 sectors, or 4 antenna with 0-3 sectors;
583              //input is (sector, antenna number)
584             antennaArray->SetSector (0,64);
585         }
586         */
587
588         std::list<Ptr<MmWaveControlMessage> > ctrlMsgs;
589         m_downlinkSpectrumPhy->StartTxDataFrames(pb, ctrlMsgs, slotPrd, slotInfo.m_slotIdx);
590     }
591
592     void
593     MmWaveEnbPhy::SendCtrlChannels(Ptr<PacketBurst> pb, std::list<Ptr<MmWaveControlMessage> > ctrlMsgs, Time slotPrd,
          SlotAllocInfo& slotInfo)
594     {
595         //std::cout << "MmWaveEnbPhy::SendCtrlChannels \n";
596         /* Send Ctrl messages*/
597         NS_LOG_FUNCTION (this<<"Send Ctrl");
598
599             //Mudando para omnidireconal nas msgs de RACH e RAR
600         Ptr<AntennaArrayModel> antennaArray = DynamicCast<AntennaArrayModel> (GetDlSpectrumPhy ()->GetRxAntenna());
601             antennaArray->ChangeToOmniTx ();
602         std::list<Ptr<MmWaveControlMessage> >::iterator it;
603         for (it = ctrlMsgs.begin (); it != ctrlMsgs.end (); it++)
604         {
605             Ptr<MmWaveControlMessage> msg = (*it);
606
607             //if (msg->GetMessageType () == MmWaveControlMessage::RAR ||
608         //  msg->GetMessageType() == MmWaveControlMessage::RACH_PREAMBLE)
609             //{
610                // std::cout<<"MUDOU PARA OMNIDIRECIONAL";
611                 antennaArray->ChangeToOmniTx ();
612             //}
613         }
614
615         m_downlinkSpectrumPhy->StartTxDlControlFrames (pb, ctrlMsgs, slotPrd, slotInfo.m_slotIdx);
616
617
618     }
619
620     bool
621     MmWaveEnbPhy::AddUePhy (uint64_t imsi, Ptr<NetDevice> ueDevice)
622     {
623         NS_LOG_FUNCTION (this<<imsi);
624         std::set <uint64_t>::iterator it;
625         it = m_ueAttached.find(imsi);
626
627         if (it == m_ueAttached.end ())
628         {
629             m_ueAttached.insert(imsi);
630             m_deviceMap.push_back(ueDevice);
631             return (true);
632         }
633         else
634         {
635             NS_LOG_ERROR ("Programming error...UE already attached");
636             return (false);
637         }
638     }
639
```

```
640 void
641 MmWaveEnbPhy::PhyDataPacketReceived (Ptr<Packet> p)
642 {
643     Simulator::ScheduleWithContext (m_netDevice->GetNode()->GetId(),
644                                     MicroSeconds(m_phyMacConfig->GetTbDecodeLatency()),
645                                     &MmWaveEnbPhySapUser::ReceivePhyPdu,
646                                     m_phySapUser,
647                                     p);
648 //      m_phySapUser->ReceivePhyPdu(p);
649 }
650
651 void
652 MmWaveEnbPhy::GenerateDataCqiReport (const SpectrumValue& sinr)
653 {
654   NS_LOG_FUNCTION (this << sinr);
655   //std::cout<< "GenerateDataCqiReport \t";
656   Values::const_iterator it;
657   MmWaveMacSchedSapProvider::SchedUlCqiInfoReqParameters ulcqi;
658   ulcqi.m_ulCqi.m_type = UlCqiInfo::PUSCH;
659   int i = 0;
660   double sinrdbmedia;
661   for (it = sinr.ConstValuesBegin (); it != sinr.ConstValuesEnd (); it++)
662   {
663
664     double sinrdb =(*it);
665
666     //       NS_LOG_DEBUG ("ULCQI RB " << i << " value " << sinrdb);
667     // convert from double to fixed point notaltion Sxxxxxxxxxx.xxx
668 //      int16_t sinrFp = LteFfConverter::double2fpS11dot3 (sinrdb);
669     ulcqi.m_ulCqi.m_sinr.push_back (*it);
670     sinrdbmedia = sinrdbmedia + sinrdb;
671     i++;
672
673   }
674  //std::cout<< "i= \t" << i << "\n";
675  //sinrdbmedia =  10 * std::log10 (sinrdbmedia/72);
676  //std::cout<< "enb-phy - sinr (db) UPLINK= \t" << sinrdbmedia << "\n";
677   // here we use the start symbol index of the slot in place of the slot index because the absolute UL slot index is
678   // not known to the scheduler when m_allocationMap gets populated
679   ulcqi.m_sfnSf = SfnSf (m_frameNum, m_sfNum, m_currSymStart);
680   SpectrumValue newSinr = sinr;
681     m_ulSinrTrace (0, newSinr, newSinr);
682   m_phySapUser->UlCqiReport (ulcqi);
683 }
684
685
686 void
687 MmWaveEnbPhy::PhyCtrlMessagesReceived (std::list<Ptr<MmWaveControlMessage> > msgList)
688 {
689     //std::cout << "ENTROU MmWaveEnbPhy::PhyCtrlMessagesReceived \n";
690     std::list<Ptr<MmWaveControlMessage> >::iterator ctrlIt = msgList.begin ();
691
692
693     while (ctrlIt != msgList.end ())
694     {
695         //std::cout << "ENTROU NO WHILE MmWaveEnbPhy::PhyCtrlMessagesReceived \n";
696         Ptr<MmWaveControlMessage> msg = (*ctrlIt);
697
698         if (msg->GetMessageType () == MmWaveControlMessage::DL_CQI)
699         {
700             NS_LOG_INFO ("received CQI");
701             m_phySapUser->ReceiveControlMessage (msg);
702         }
703         else if (msg->GetMessageType () == MmWaveControlMessage::BSR)
704         {
705             NS_LOG_INFO ("received BSR");
706             m_phySapUser->ReceiveControlMessage (msg);
707         }
708         else if (msg->GetMessageType() == MmWaveControlMessage::RACH_PREAMBLE)
709       {
710             NS_LOG_INFO ("received RACH_PREAMBLE");
711
712             NS_ASSERT (m_cellId > 0);
713             Ptr<MmWaveRachPreambleMessage> rachPreamble = DynamicCast<MmWaveRachPreambleMessage> (msg);
714             m_phySapUser->ReceiveRachPreamble (rachPreamble->GetRapId ());
715             std::cout << "received RACH_PREAMBLE with RapID \t" << rachPreamble->GetRapId () << "\n";
716         }
717         else if (msg->GetMessageType() == MmWaveControlMessage::DL_HARQ)
718         {
719             Ptr<MmWaveDlHarqFeedbackMessage> dlharqMsg = DynamicCast<MmWaveDlHarqFeedbackMessage> (msg);
720             DlHarqInfo dlharq = dlharqMsg->GetDlHarqFeedback ();
```

```
721                 // check whether the UE is connected
722                 if (m_ueAttached.find (dlharq.m_rnti) != m_ueAttached.end ())
723                 {
724                     m_phySapUser->ReceiveControlMessage (msg);
725                 }
726             }
727
728             ctrlIt++;
729         }
730
731     }
732
733     uint32_t
734     MmWaveEnbPhy::GetAbsoluteSubframeNo ()
735     {
736         return ((m_frameNum - 1)*(m_phyMacConfig->GetSubframesPerFrame()*m_phyMacConfig->GetSlotsPerSubframe()) + m_slotNum);
737     }
738
739     ////////////////////////////////////////////////////////
740     ////////                  sap                 /////////
741     ////////////////////////////////////////////////////////
742
743     void
744     MmWaveEnbPhy::DoSetBandwidth (uint8_t ulBandwidth, uint8_t dlBandwidth)
745     {
746       NS_LOG_FUNCTION (this << (uint32_t) ulBandwidth << (uint32_t) dlBandwidth);
747     }
748
749     void
750     MmWaveEnbPhy::DoSetEarfcn (uint16_t ulEarfcn, uint16_t dlEarfcn)
751     {
752       NS_LOG_FUNCTION (this << ulEarfcn << dlEarfcn);
753     }
754
755
756     void
757     MmWaveEnbPhy::DoAddUe (uint16_t rnti)
758     {
759         NS_LOG_FUNCTION (this << rnti);
760         bool success = AddUePhy (rnti);
761         NS_ASSERT_MSG (success, "AddUePhy() failed");
762
763     }
764
765     bool
766     MmWaveEnbPhy::AddUePhy (uint16_t rnti)
767     {
768       NS_LOG_FUNCTION (this << rnti);
769       std::set <uint16_t>::iterator it;
770       it = m_ueAttachedRnti.find (rnti);
771       if (it == m_ueAttachedRnti.end ())
772         {
773           m_ueAttachedRnti.insert (rnti);
774           return (true);
775         }
776       else
777         {
778           NS_LOG_ERROR ("UE already attached");
779           return (false);
780         }
781     }
782
783     void
784     MmWaveEnbPhy::DoRemoveUe (uint16_t rnti)
785     {
786       NS_LOG_FUNCTION (this << rnti);
787     }
788
789     void
790     MmWaveEnbPhy::DoSetPa (uint16_t rnti, double pa)
791     {
792       NS_LOG_FUNCTION (this << rnti);
793     }
794
795     void
796     MmWaveEnbPhy::DoSetTransmissionMode (uint16_t  rnti, uint8_t txMode)
797     {
798       NS_LOG_FUNCTION (this << rnti << (uint16_t)txMode);
799       // UL supports only SISO MODE
800     }
801
```

```
802 void
803 MmWaveEnbPhy::DoSetSrsConfigurationIndex (uint16_t  rnti, uint16_t srcCi)
804 {
805   NS_LOG_FUNCTION (this);
806 }
807
808
809 void
810 MmWaveEnbPhy::DoSetMasterInformationBlock (LteRrcSap::MasterInformationBlock mib)
811 {
812   NS_LOG_FUNCTION (this);
813   //m_mib = mib;
814 }
815
816
817 void
818 MmWaveEnbPhy::DoSetSystemInformationBlockType1 (LteRrcSap::SystemInformationBlockType1 sib1)
819 {
820   NS_LOG_FUNCTION (this);
821   m_sib1 = sib1;
822 }
823
824 int8_t
825 MmWaveEnbPhy::DoGetReferenceSignalPower () const
826 {
827   NS_LOG_FUNCTION (this);
828   return m_txPower;
829 }
830
831 void
832 MmWaveEnbPhy::SetPhySapUser (MmWaveEnbPhySapUser* ptr)
833 {
834     m_phySapUser = ptr;
835 }
836
837 void
838 MmWaveEnbPhy::SetHarqPhyModule (Ptr<MmWaveHarqPhy> harq)
839 {
840   m_harqPhyModule = harq;
841 }
842
843 void
844 MmWaveEnbPhy::ReceiveUlHarqFeedback (UlHarqInfo mes)
845 {
846   NS_LOG_FUNCTION (this);
847   // forward to scheduler
848   m_phySapUser->UlHarqFeedback (mes);
849 }
850
851 }
```

Classe: mmwave-ue-phy.cc

```
1
2 /*
3  * mmwave-ue-phy.cc
4  *
5  *  Created on: Nov 5, 2014
6  *      Author: sourjya
7  */
8
9 #include <ns3/object-factory.h>
10 #include <ns3/log.h>
11 #include <cfloat>
12 #include <cmath>
13 #include <ns3/simulator.h>
14 #include <ns3/double.h>
15 #include "mmwave-ue-phy.h"
16 #include "mmwave-ue-net-device.h"
17 #include "mmwave-spectrum-value-helper.h"
18 #include <ns3/pointer.h>
19 #include <ns3/node.h>
20
21 namespace ns3{
22
23 NS_LOG_COMPONENT_DEFINE ("MmWaveUePhy");
24
```

```
25  NS_OBJECT_ENSURE_REGISTERED (MmWaveUePhy);
26
27  MmWaveUePhy::MmWaveUePhy ()
28  {
29      NS_LOG_FUNCTION (this);
30      NS_FATAL_ERROR ("This constructor should not be called");
31  }
32
33  MmWaveUePhy::MmWaveUePhy (Ptr<MmWaveSpectrumPhy> dlPhy, Ptr<MmWaveSpectrumPhy> ulPhy)
34  : MmWavePhy(dlPhy, ulPhy),
35    m_prevSlot (0),
36    m_rnti (0)
37  {
38      NS_LOG_FUNCTION (this);
39      m_wbCqiLast = Simulator::Now ();
40      m_ueCphySapProvider = new MemberLteUeCphySapProvider<MmWaveUePhy> (this);
41      Simulator::ScheduleNow (&MmWaveUePhy::SubframeIndication, this, 0, 0);
42  }
43
44  MmWaveUePhy::~MmWaveUePhy ()
45  {
46      NS_LOG_FUNCTION (this);
47  }
48
49  TypeId
50  MmWaveUePhy::GetTypeId (void)
51  {
52      static TypeId tid = TypeId ("ns3::MmWaveUePhy")
53          .SetParent<MmWavePhy> ()
54          .AddConstructor<MmWaveUePhy> ()
55          .AddAttribute ("TxPower",
56                         "Transmission power in dBm",
57                         DoubleValue (30.0), //TBD zml
58                         MakeDoubleAccessor (&MmWaveUePhy::SetTxPower,
59                                             &MmWaveUePhy::GetTxPower),
60                         MakeDoubleChecker<double> ())
61          .AddAttribute ("DlSpectrumPhy",
62                          "The downlink MmWaveSpectrumPhy associated to this MmWavePhy",
63                          TypeId::ATTR_GET,
64                          PointerValue (),
65                          MakePointerAccessor (&MmWaveUePhy::GetDlSpectrumPhy),
66                          MakePointerChecker <MmWaveSpectrumPhy> ())
67          .AddAttribute ("UlSpectrumPhy",
68                         "The uplink MmWaveSpectrumPhy associated to this MmWavePhy",
69                         TypeId::ATTR_GET,
70                         PointerValue (),
71                         MakePointerAccessor (&MmWaveUePhy::GetUlSpectrumPhy),
72                         MakePointerChecker <MmWaveSpectrumPhy> ())
73          .AddTraceSource ("ReportCurrentCellRsrpSinr",
74                           "RSRP and SINR statistics.",
75                           MakeTraceSourceAccessor (&MmWaveUePhy::m_reportCurrentCellRsrpSinrTrace),
76                           "ns3::CurrentCellRsrpSinr::TracedCallback")
77          .AddTraceSource ("ReportUplinkTbSize",
78                           "Report allocated uplink TB size for trace.",
79                           MakeTraceSourceAccessor (&MmWaveUePhy::m_reportUlTbSize),
80                           "ns3::UlTbSize::TracedCallback")
81          .AddTraceSource ("ReportDownlinkTbSize",
82                           "Report allocated downlink TB size for trace.",
83                           MakeTraceSourceAccessor (&MmWaveUePhy::m_reportDlTbSize),
84                           "ns3::DlTbSize::TracedCallback")
85  ;
86
87      return tid;
88  }
89
90  void
91  MmWaveUePhy::DoInitialize (void)
92  {
93      NS_LOG_FUNCTION (this);
94      m_dlCtrlPeriod = NanoSeconds (1000 * m_phyMacConfig->GetDlCtrlSymbols() * m_phyMacConfig->GetSymbolPeriod());
95      m_ulCtrlPeriod = NanoSeconds (1000 * m_phyMacConfig->GetUlCtrlSymbols() * m_phyMacConfig->GetSymbolPeriod());
96
97      for (unsigned i = 0; i < m_phyMacConfig->GetSubframesPerFrame(); i++)
98      {
99          m_sfAllocInfo.push_back(SfAllocInfo(SfnSf (0, i, 0)));
100         SlotAllocInfo dlCtrlSlot;
101         dlCtrlSlot.m_slotType = SlotAllocInfo::CTRL;
102         dlCtrlSlot.m_numCtrlSym = 1;
103         dlCtrlSlot.m_tddMode = SlotAllocInfo::DL;
104         dlCtrlSlot.m_dci.m_numSym = 1;
105         dlCtrlSlot.m_dci.m_symStart = 0;
```

```
106             SlotAllocInfo ulCtrlSlot;
107             ulCtrlSlot.m_slotType = SlotAllocInfo::CTRL;
108             ulCtrlSlot.m_numCtrlSym = 1;
109             ulCtrlSlot.m_tddMode = SlotAllocInfo::UL;
110             ulCtrlSlot.m_slotIdx = 0xFF;
111             ulCtrlSlot.m_dci.m_numSym = 1;
112             ulCtrlSlot.m_dci.m_symStart = m_phyMacConfig->GetSymbolsPerSubframe()-1;
113             m_sfAllocInfo[i].m_slotAllocInfo.push_back (dlCtrlSlot);
114             m_sfAllocInfo[i].m_slotAllocInfo.push_back (ulCtrlSlot);
115         }
116
117         for (unsigned i = 0; i < m_phyMacConfig->GetTotalNumChunk(); i++)
118         {
119             m_channelChunks.push_back(i);
120         }
121
122         m_sfPeriod = NanoSeconds (1000.0 * m_phyMacConfig->GetSubframePeriod ());
123
124         MmWavePhy::DoInitialize ();
125     }
126
127     void
128     MmWaveUePhy::DoDispose (void)
129     {
130
131     }
132
133     void
134     MmWaveUePhy::SetUeCphySapUser (LteUeCphySapUser* s)
135     {
136       NS_LOG_FUNCTION (this);
137       m_ueCphySapUser = s;
138     }
139
140     LteUeCphySapProvider*
141     MmWaveUePhy::GetUeCphySapProvider ()
142     {
143       NS_LOG_FUNCTION (this);
144       return (m_ueCphySapProvider);
145     }
146
147     void
148     MmWaveUePhy::SetTxPower (double pow)
149     {
150         m_txPower = pow;
151     }
152     double
153     MmWaveUePhy::GetTxPower () const
154     {
155         return m_txPower;
156     }
157
158     void
159     MmWaveUePhy::SetNoiseFigure (double pf)
160     {
161
162     }
163
164     double
165     MmWaveUePhy::GetNoiseFigure () const
166     {
167         return m_noiseFigure;
168     }
169
170     Ptr<SpectrumValue>
171     MmWaveUePhy::CreateTxPowerSpectralDensity()
172     {
173         Ptr<SpectrumValue> psd =
174                 MmWaveSpectrumValueHelper::CreateTxPowerSpectralDensity (m_phyMacConfig, m_txPower, m_subChannelsForTx );
175         return psd;
176     }
177
178     void
179     MmWaveUePhy::DoSetSubChannels()
180     {
181
182     }
183
184     void
185     MmWaveUePhy::SetSubChannelsForReception(std::vector <int> mask)
186     {
```

```
187
188 }
189
190 std::vector <int>
191 MmWaveUePhy::GetSubChannelsForReception(void)
192 {
193     std::vector <int> vec;
194
195     return vec;
196 }
197
198 void
199 MmWaveUePhy::SetSubChannelsForTransmission(std::vector <int> mask)
200 {
201     m_subChannelsForTx = mask;
202     Ptr<SpectrumValue> txPsd = CreateTxPowerSpectralDensity ();
203     NS_ASSERT (txPsd);
204     //std::cout << "UE transmitindo \n";
205     m_downlinkSpectrumPhy->SetTxPowerSpectralDensity (txPsd);
206 }
207
208 std::vector <int>
209 MmWaveUePhy::GetSubChannelsForTransmission(void)
210 {
211     std::vector <int> vec;
212
213     return vec;
214 }
215
216 void
217 MmWaveUePhy::DoSendControlMessage (Ptr<MmWaveControlMessage> msg)
218 {
219   NS_LOG_FUNCTION (this << msg);
220   SetControlMessage (msg);
221 }
222
223
224 void
225 MmWaveUePhy::RegisterToEnb (uint16_t cellId, Ptr<MmWavePhyMacCommon> config)
226 {
227     m_cellId = cellId;
228     //TBD how to assign bandwitdh and earfcn
229     m_noiseFigure = 5.0;
230     m_phyMacConfig = config;
231
232     Ptr<SpectrumValue> noisePsd =
233             MmWaveSpectrumValueHelper::CreateNoisePowerSpectralDensity (m_phyMacConfig, m_noiseFigure);
234     m_downlinkSpectrumPhy->SetNoisePowerSpectralDensity (noisePsd);
235     m_downlinkSpectrumPhy->GetSpectrumChannel()->AddRx(m_downlinkSpectrumPhy);
236     m_downlinkSpectrumPhy->SetCellId(m_cellId);
237 }
238
239 Ptr<MmWaveSpectrumPhy>
240 MmWaveUePhy::GetDlSpectrumPhy () const
241 {
242   return m_downlinkSpectrumPhy;
243 }
244
245 Ptr<MmWaveSpectrumPhy>
246 MmWaveUePhy::GetUlSpectrumPhy () const
247 {
248   return m_uplinkSpectrumPhy;
249 }
250
251 void
252 MmWaveUePhy::ReceiveControlMessageList (std::list<Ptr<MmWaveControlMessage> > msgList)
253 {
254     NS_LOG_FUNCTION (this);
255
256     std::list<Ptr<MmWaveControlMessage> >::iterator it;
257     for (it = msgList.begin (); it != msgList.end (); it++)
258     {
259         Ptr<MmWaveControlMessage> msg = (*it);
260
261         if (msg->GetMessageType() == MmWaveControlMessage::DCI_TDMA)
262         {
263             NS_ASSERT_MSG (m_slotNum == 0, "UE" << m_rnti << " got DCI on slot != 0");
264             Ptr<MmWaveTdmaDciMessage> dciMsg = DynamicCast<MmWaveTdmaDciMessage> (msg);
265             DciInfoElementTdma dciInfoElem = dciMsg->GetDciInfoElement ();
266             SfnSf dciSfn = dciMsg->GetSfnSf ();
267
```

```
268             if(dciSfn.m_frameNum != m_frameNum || dciSfn.m_sfNum != m_sfNum)
269             {
270                 NS_FATAL_ERROR ("DCI intended for different subframe (dci= "
271                         << dciSfn.m_frameNum<<" "<<dciSfn.m_sfNum<<", actual= "<<m_frameNum<<" "<<m_sfNum);
272             }
273 
274 //          NS_LOG_DEBUG ("UE" << m_rnti << " DCI received for RNTI " << dciInfoElem.m_rnti << " in frame " << m_frameNum <<
        " subframe " << (unsigned)m_sfNum << " slot " << (unsigned)m_slotNum << " format " << (unsigned)dciInfoElem.m_format
        << " symStart " << (unsigned)dciInfoElem.m_symStart << " numSym " << (unsigned)dciInfoElem.m_numSym);
275 
276             if (dciInfoElem.m_rnti != m_rnti)
277             {
278                 continue; // DCI not for me
279             }
280 
281             if (dciInfoElem.m_format == DciInfoElementTdma::DL) // set downlink slot schedule for current slot
282             {
283                 NS_LOG_DEBUG ("UE" << m_rnti << " DL-DCI received for frame " << m_frameNum << " subframe " << (unsigned)
                        m_sfNum
284                                 << " symStart " << (unsigned)dciInfoElem.m_symStart << " numSym " << (unsigned)dciInfoElem.
                                        m_numSym  << " tbs " << dciInfoElem.m_tbSize
285                                 << " harqId " << (unsigned)dciInfoElem.m_harqProcess);
286 
287                 SlotAllocInfo slotInfo;
288                 slotInfo.m_tddMode = SlotAllocInfo::DL;
289                 slotInfo.m_dci = dciInfoElem;
290                 slotInfo.m_slotIdx = 0;
291                 std::deque <SlotAllocInfo>::iterator itSlot;
292                 for (itSlot = m_currSfAllocInfo.m_slotAllocInfo.begin ();
293                         itSlot != m_currSfAllocInfo.m_slotAllocInfo.end (); itSlot++)
294                 {
295                     if (itSlot->m_tddMode == SlotAllocInfo::UL)
296                     {
297                         break;
298                     }
299                     slotInfo.m_slotIdx++;
300                 }
301                 //m_currSfAllocInfo.m_slotAllocInfo.push_back (slotInfo);  // add SlotAllocInfo to current SfAllocInfo
302                 m_currSfAllocInfo.m_slotAllocInfo.insert (itSlot, slotInfo);
303             }
304             else if (dciInfoElem.m_format == DciInfoElementTdma::UL) // set downlink slot schedule for t+Tul_sched slot
305             {
306                 uint8_t ulSfIdx = (m_sfNum + m_phyMacConfig->GetUlSchedDelay()) % m_phyMacConfig->GetSubframesPerFrame ();
307                 uint16_t dciFrame = (ulSfIdx > m_sfNum) ? m_frameNum : m_frameNum+1;
308 
309                 NS_LOG_DEBUG ("UE" << m_rnti << " UL-DCI received for frame " << dciFrame << " subframe " << (unsigned)
                        ulSfIdx
310                                 << " symStart " << (unsigned)dciInfoElem.m_symStart << " numSym " << (unsigned)dciInfoElem.
                                        m_numSym << " tbs " << dciInfoElem.m_tbSize
311                                 << " harqId " << (unsigned)dciInfoElem.m_harqProcess);
312 
313                 SlotAllocInfo slotInfo;
314                 slotInfo.m_tddMode = SlotAllocInfo::UL;
315                 slotInfo.m_dci = dciInfoElem;
316                 SlotAllocInfo ulCtrlSlot = m_sfAllocInfo[ulSfIdx].m_slotAllocInfo.back ();
317                 m_sfAllocInfo[ulSfIdx].m_slotAllocInfo.pop_back ();
318                 //ulCtrlSlot.m_slotIdx++;
319                 slotInfo.m_slotIdx = m_sfAllocInfo[ulSfIdx].m_slotAllocInfo.size ();
320                 m_sfAllocInfo[ulSfIdx].m_slotAllocInfo.push_back (slotInfo);
321                 m_sfAllocInfo[ulSfIdx].m_slotAllocInfo.push_back (ulCtrlSlot);
322             }
323 
324             m_phySapUser->ReceiveControlMessage (msg);
325         }
326         else if (msg->GetMessageType () == MmWaveControlMessage::MIB)
327         {
328             NS_LOG_INFO ("received MIB");
329             NS_ASSERT (m_cellId > 0);
330             Ptr<MmWaveMibMessage> msg2 = DynamicCast<MmWaveMibMessage> (msg);
331             m_ueCphySapUser->RecvMasterInformationBlock (m_cellId, msg2->GetMib ());
332         }
333         else if (msg->GetMessageType () == MmWaveControlMessage::SIB1)
334         {
335             NS_ASSERT (m_cellId > 0);
336             Ptr<MmWaveSib1Message> msg2 = DynamicCast<MmWaveSib1Message> (msg);
337             m_ueCphySapUser->RecvSystemInformationBlockType1 (m_cellId, msg2->GetSib1 ());
338         }
339         else if (msg->GetMessageType () == MmWaveControlMessage::RAR)
340         {
341             NS_LOG_INFO ("received RAR");
342             std::cout << "received RAR \n";
```

```
343                 NS_ASSERT (m_cellId > 0);
344
345                 Ptr<MmWaveRarMessage> rarMsg = DynamicCast<MmWaveRarMessage> (msg);
346
347                 for (std::list<MmWaveRarMessage::Rar>::const_iterator it = rarMsg->RarListBegin ();
348                         it != rarMsg->RarListEnd ();
349                         ++it)
350                 {
351                     if (it->rapId == m_raPreambleId)
352                     {
353                         m_phySapUser->ReceiveControlMessage (rarMsg);
354                     }
355                 }
356             }
357             else
358             {
359                 NS_LOG_DEBUG ("Control message not handled. Type: "<< msg->GetMessageType());
360             }
361         }
362     }
363
364     void
365     MmWaveUePhy::QueueUlTbAlloc (TbAllocInfo m)
366     {
367       NS_LOG_FUNCTION (this);
368     //  NS_LOG_DEBUG ("UL TB Info Elem queue size == " << m_ulTbAllocQueue.size ());
369       m_ulTbAllocQueue.at (m_phyMacConfig->GetUlSchedDelay ()-1).push_back (m);
370     }
371
372     std::list<TbAllocInfo>
373     MmWaveUePhy::DequeueUlTbAlloc (void)
374     {
375         NS_LOG_FUNCTION (this);
376
377         if (m_ulTbAllocQueue.empty())
378         {
379             std::list<TbAllocInfo> emptylist;
380             return (emptylist);
381         }
382
383         if (m_ulTbAllocQueue.at (0).size () > 0)
384         {
385             std::list<TbAllocInfo> ret = m_ulTbAllocQueue.at (0);
386             m_ulTbAllocQueue.erase (m_ulTbAllocQueue.begin ());
387             std::list<TbAllocInfo> l;
388             m_ulTbAllocQueue.push_back (l);
389             return (ret);
390         }
391         else
392         {
393             m_ulTbAllocQueue.erase (m_ulTbAllocQueue.begin ());
394             std::list<TbAllocInfo> l;
395             m_ulTbAllocQueue.push_back (l);
396             std::list<TbAllocInfo> emptylist;
397             return (emptylist);
398         }
399     }
400
401     void
402     MmWaveUePhy::SubframeIndication (uint16_t frameNum, uint8_t sfNum)
403     {
404         m_frameNum = frameNum;
405         m_sfNum = sfNum;
406         m_lastSfStart = Simulator::Now();
407         m_currSfAllocInfo = m_sfAllocInfo[m_sfNum];
408         NS_ASSERT ((m_currSfAllocInfo.m_sfnSf.m_frameNum == m_frameNum) &&
409                    (m_currSfAllocInfo.m_sfnSf.m_sfNum == m_sfNum));
410         m_sfAllocInfo[m_sfNum] = SfAllocInfo (SfnSf (m_frameNum+1, m_sfNum, 0));
411         SlotAllocInfo dlCtrlSlot;
412         dlCtrlSlot.m_slotType = SlotAllocInfo::CTRL;
413         dlCtrlSlot.m_numCtrlSym = 1;
414         dlCtrlSlot.m_tddMode = SlotAllocInfo::DL;
415         dlCtrlSlot.m_dci.m_numSym = 1;
416         dlCtrlSlot.m_dci.m_symStart = 0;
417         SlotAllocInfo ulCtrlSlot;
418         ulCtrlSlot.m_slotType = SlotAllocInfo::CTRL;
419         ulCtrlSlot.m_numCtrlSym = 1;
420         ulCtrlSlot.m_tddMode = SlotAllocInfo::UL;
421         ulCtrlSlot.m_slotIdx = 0xFF;
422         ulCtrlSlot.m_dci.m_numSym = 1;
423         ulCtrlSlot.m_dci.m_symStart = m_phyMacConfig->GetSymbolsPerSubframe()-1;
```

```
424       m_sfAllocInfo[m_sfNum].m_slotAllocInfo.push_front (dlCtrlSlot);
425       m_sfAllocInfo[m_sfNum].m_slotAllocInfo.push_back (ulCtrlSlot);
426
427       StartSlot ();
428   }
429
430   void
431   MmWaveUePhy::StartSlot ()
432   {
433       //std::cout << "MmWaveUePhy::StartSlot \n";
434       //unsigned slotInd = 0;
435       SlotAllocInfo currSlot;
436       /*if (m_slotNum >= m_currSfAllocInfo.m_dlSlotAllocInfo.size ())
437       {
438           if (m_currSfAllocInfo.m_ulSlotAllocInfo.size () > 0)
439           {
440               slotInd = m_slotNum - m_currSfAllocInfo.m_dlSlotAllocInfo.size ();
441               currSlot = m_currSfAllocInfo.m_ulSlotAllocInfo[slotInd];
442           }
443       }
444       else
445       {
446           if (m_currSfAllocInfo.m_ulSlotAllocInfo.size () > 0)
447           {
448               slotInd = m_slotNum;
449               currSlot = m_currSfAllocInfo.m_dlSlotAllocInfo[slotInd];
450           }
451       }*/
452
453       currSlot = m_currSfAllocInfo.m_slotAllocInfo[m_slotNum];
454       m_currSlot = currSlot;
455
456       NS_LOG_INFO ("UE " << m_rnti << " frame " << m_frameNum << " subframe " << m_sfNum << " slot " << m_slotNum);
457
458       Time slotPeriod;
459
460       if (m_slotNum == 0)  // reserved DL control
461       {
462           //std::cout << "UE-PHY DL m_slotNum == 0 \n";
463           slotPeriod = NanoSeconds (1000.0 * m_phyMacConfig->GetSymbolPeriod () * m_phyMacConfig->GetDlCtrlSymbols ());
464           //Acrescentei essa chamada da fun  o AddExpectedTb
465           //**********************
466           m_downlinkSpectrumPhy->AddExpectedTb (currSlot.m_dci.m_rnti, currSlot.m_dci.m_ndi, currSlot.m_dci.m_tbSize, currSlot
                  .m_dci.m_mcs,
467                                                 m_channelChunks, currSlot.m_dci.m_harqProcess, currSlot.m_dci.m_rv, true,
468                                                 currSlot.m_dci.m_symStart, currSlot.m_dci.m_numSym);
469           m_reportDlTbSize (GetDevice ()->GetObject <MmWaveUeNetDevice> ()->GetImsi(), currSlot.m_dci.m_tbSize);
470           //****************
471
472           NS_LOG_DEBUG ("UE" << m_rnti << " RXing DL CTRL frame " << m_frameNum << " subframe " << (unsigned)m_sfNum << "
                  symbols "
473                         << (unsigned)currSlot.m_dci.m_symStart << "-" << (unsigned)(currSlot.m_dci.m_symStart+currSlot.m_dci.
                        m_numSym-1) <<
474                            "\t start " << Simulator::Now() << " end " << (Simulator::Now()+slotPeriod));
475       }
476       else if (m_slotNum == m_currSfAllocInfo.m_slotAllocInfo.size()-1) // reserved UL control
477       {
478           //std::cout << "Ue-phy UL m_slotNum =! 0 \n";
479           m_receptionEnabled = false;
480           SetSubChannelsForTransmission (m_channelChunks);
481           slotPeriod = NanoSeconds (1000.0 * m_phyMacConfig->GetSymbolPeriod () * m_phyMacConfig->GetUlCtrlSymbols ());
482           //Acrescentei essa chamada da fun  o AddExpectedTb
483           //*******************
484
485           Ptr<PacketBurst> pktBurst = GetPacketBurst (SfnSf(m_frameNum, m_sfNum, currSlot.m_dci.m_symStart));
486           if (pktBurst && pktBurst->GetNPackets () > 0)
487           {
488               std::list< Ptr<Packet> > pkts = pktBurst->GetPackets ();
489               MmWaveMacPduTag tag;
490               pkts.front ()->PeekPacketTag (tag);
491               NS_ASSERT ((tag.GetSfn().m_sfNum == m_sfNum) && (tag.GetSfn().m_slotNum == currSlot.m_dci.m_symStart));
492
493               LteRadioBearerTag bearerTag;
494               if(!pkts.front ()->PeekPacketTag (bearerTag))
495               {
496                   NS_FATAL_ERROR ("No radio bearer tag");
497               }
498           }
499           else
500           {
501               // sometimes the UE will be scheduled when no data is queued
```

```
502             // in this case, send an empty PDU
503             MmWaveMacPduTag tag (SfnSf(m_frameNum, m_sfNum, currSlot.m_dci.m_symStart));
504             Ptr<Packet> emptyPdu = Create <Packet> ();
505             MmWaveMacPduHeader header;
506             MacSubheader subheader (3, 0);  // lcid = 3, size = 0
507             header.AddSubheader (subheader);
508             emptyPdu->AddHeader (header);
509             emptyPdu->AddPacketTag (tag);
510             LteRadioBearerTag bearerTag (m_rnti, 3, 0);
511             emptyPdu->AddPacketTag (bearerTag);
512             pktBurst = CreateObject<PacketBurst> ();
513             pktBurst->AddPacket (emptyPdu);
514         }
515
516         m_reportUlTbSize (GetDevice ()->GetObject <MmWaveUeNetDevice> ()->GetImsi(), currSlot.m_dci.m_tbSize);
517
518         //****************************
519         std::list<Ptr<MmWaveControlMessage> > ctrlMsg = GetControlMessages ();
520         NS_LOG_DEBUG ("UE" << m_rnti << " TXing UL CTRL frame " << m_frameNum << " subframe " << (unsigned)m_sfNum << "
                symbols "
521                       << (unsigned)currSlot.m_dci.m_symStart << "-" << (unsigned)(currSlot.m_dci.m_symStart+currSlot.m_dci.
                            m_numSym-1) <<
522                           "\t start " << Simulator::Now() << " end " << (Simulator::Now()+slotPeriod-NanoSeconds(1.0)));
523         SendCtrlChannels (pktBurst, ctrlMsg, slotPeriod-NanoSeconds(1.0), m_slotNum);
524
525     }
526     else if (currSlot.m_dci.m_format == DciInfoElementTdma::DL)  // scheduled DL data slot
527     {
528         //std::cout << "Entrou no if \n";
529         m_receptionEnabled = true;
530         slotPeriod = NanoSeconds (1000.0 * m_phyMacConfig->GetSymbolPeriod () * currSlot.m_dci.m_numSym);
531         //std::cout << "AddExpectedTb - UE \n";
532         m_downlinkSpectrumPhy->AddExpectedTb (currSlot.m_dci.m_rnti, currSlot.m_dci.m_ndi, currSlot.m_dci.m_tbSize, currSlot
                .m_dci.m_mcs,
533                                               m_channelChunks, currSlot.m_dci.m_harqProcess, currSlot.m_dci.m_rv, true,
534                                               currSlot.m_dci.m_symStart, currSlot.m_dci.m_numSym);
535         m_reportDlTbSize (GetDevice ()->GetObject <MmWaveUeNetDevice> ()->GetImsi(), currSlot.m_dci.m_tbSize);
536         NS_LOG_DEBUG ("UE" << m_rnti << " RXing DL DATA frame " << m_frameNum << " subframe " << (unsigned)m_sfNum << "
                symbols "
537                       << (unsigned)currSlot.m_dci.m_symStart << "-" << (unsigned)(currSlot.m_dci.m_symStart+currSlot.m_dci.
                            m_numSym-1) <<
538                       "\t start " << Simulator::Now() << " end " << (Simulator::Now()+slotPeriod));
539     }
540     else if (currSlot.m_dci.m_format == DciInfoElementTdma::UL) // scheduled UL data slot
541     {
542         m_receptionEnabled = false;
543         SetSubChannelsForTransmission (m_channelChunks);
544         slotPeriod = NanoSeconds (1000.0 * m_phyMacConfig->GetSymbolPeriod () * currSlot.m_dci.m_numSym);
545         std::list<Ptr<MmWaveControlMessage> > ctrlMsg = GetControlMessages ();
546         Ptr<PacketBurst> pktBurst = GetPacketBurst (SfnSf(m_frameNum, m_sfNum, currSlot.m_dci.m_symStart));
547         if (pktBurst && pktBurst->GetNPackets () > 0)
548         {
549             std::list< Ptr<Packet> > pkts = pktBurst->GetPackets ();
550             MmWaveMacPduTag tag;
551             pkts.front ()->PeekPacketTag (tag);
552             NS_ASSERT ((tag.GetSfn().m_sfNum == m_sfNum) && (tag.GetSfn().m_slotNum == currSlot.m_dci.m_symStart));
553
554             LteRadioBearerTag bearerTag;
555             if(!pkts.front ()->PeekPacketTag (bearerTag))
556             {
557                 NS_FATAL_ERROR ("No radio bearer tag");
558             }
559         }
560         else
561         {
562             // sometimes the UE will be scheduled when no data is queued
563             // in this case, send an empty PDU
564             MmWaveMacPduTag tag (SfnSf(m_frameNum, m_sfNum, currSlot.m_dci.m_symStart));
565             Ptr<Packet> emptyPdu = Create <Packet> ();
566             MmWaveMacPduHeader header;
567             MacSubheader subheader (3, 0);  // lcid = 3, size = 0
568             header.AddSubheader (subheader);
569             emptyPdu->AddHeader (header);
570             emptyPdu->AddPacketTag (tag);
571             LteRadioBearerTag bearerTag (m_rnti, 3, 0);
572             emptyPdu->AddPacketTag (bearerTag);
573             pktBurst = CreateObject<PacketBurst> ();
574             pktBurst->AddPacket (emptyPdu);
575         }
576         m_reportUlTbSize (GetDevice ()->GetObject <MmWaveUeNetDevice> ()->GetImsi(), currSlot.m_dci.m_tbSize);
577         NS_LOG_DEBUG ("UE" << m_rnti << " TXing UL DATA frame " << m_frameNum << " subframe " << (unsigned)m_sfNum << "
```

```
                symbols "
578                     << (unsigned)currSlot.m_dci.m_symStart << "-" << (unsigned)(currSlot.m_dci.m_symStart+currSlot.m_dci.
                            m_numSym-1)
579                     << "\t start " << Simulator::Now() << " end " << (Simulator::Now()+slotPeriod));
580           Simulator::Schedule (NanoSeconds(1.0), &MmWaveUePhy::SendDataChannels, this, pktBurst, ctrlMsg, slotPeriod-
                  NanoSeconds(2.0), m_slotNum);
581       }
582
583       m_prevSlotDir = currSlot.m_tddMode;
584
585       m_phySapUser->SubframeIndication (SfnSf(m_frameNum, m_sfNum, m_slotNum));   // trigger mac
586
587       //NS_LOG_DEBUG ("MmWaveUePhy: Scheduling slot end for " << slotPeriod);
588       Simulator::Schedule (slotPeriod, &MmWaveUePhy::EndSlot, this);
589   }
590
591
592   void
593   MmWaveUePhy::EndSlot ()
594   {
595       if (m_slotNum == m_currSfAllocInfo.m_slotAllocInfo.size()-1)
596       {   // end of subframe
597           uint16_t frameNum;
598           uint8_t sfNum;
599           if (m_sfNum == m_phyMacConfig->GetSubframesPerFrame ()-1)
600           {
601               sfNum = 0;
602               frameNum = m_frameNum + 1;
603           }
604           else
605           {
606               frameNum = m_frameNum;
607               sfNum = m_sfNum + 1;
608           }
609           m_slotNum = 0;
610           //NS_LOG_DEBUG ("MmWaveUePhy: Next subframe scheduled for " << m_lastSfStart + m_sfPeriod - Simulator::Now());
611           Simulator::Schedule (m_lastSfStart + m_sfPeriod - Simulator::Now(), &MmWaveUePhy::SubframeIndication, this, frameNum
                  , sfNum);
612       }
613       else
614       {
615           Time nextSlotStart;
616           /*uint8_t slotInd = m_slotNum+1;
617           if (slotInd >= m_currSfAllocInfo.m_dlSlotAllocInfo.size ())
618           {
619               if (m_currSfAllocInfo.m_ulSlotAllocInfo.size () > 0)
620               {
621                   slotInd = slotInd - m_currSfAllocInfo.m_dlSlotAllocInfo.size ();
622                   nextSlotStart = NanoSeconds (1000.0 * m_phyMacConfig->GetSymbolPeriod () *
623                                                m_currSfAllocInfo.m_ulSlotAllocInfo[slotInd].m_dci.m_symStart);
624               }
625           }
626           else
627           {
628               if (m_currSfAllocInfo.m_ulSlotAllocInfo.size () > 0)
629               {
630                   nextSlotStart = NanoSeconds (1000.0 * m_phyMacConfig->GetSymbolPeriod () *
631                                                m_currSfAllocInfo.m_dlSlotAllocInfo[slotInd].m_dci.m_symStart);
632               }
633           }*/
634           m_slotNum++;
635           nextSlotStart = NanoSeconds (1000.0 * m_phyMacConfig->GetSymbolPeriod () *
636                                                        m_currSfAllocInfo.m_slotAllocInfo[m_slotNum].m_dci.m_symStart);
637           Simulator::Schedule (nextSlotStart+m_lastSfStart-Simulator::Now(), &MmWaveUePhy::StartSlot, this);
638       }
639
640       if (m_receptionEnabled)
641       {
642           m_receptionEnabled = false;
643       }
644   }
645
646
647   uint32_t
648   MmWaveUePhy::GetSubframeNumber (void)
649   {
650       return m_slotNum;
651   }
652
653   void
654   MmWaveUePhy::PhyDataPacketReceived (Ptr<Packet> p)
```

```
655 {
656   Simulator::ScheduleWithContext (m_netDevice->GetNode()->GetId(),
657                                   MicroSeconds(m_phyMacConfig->GetTbDecodeLatency()),
658                                   &MmWaveUePhySapUser::ReceivePhyPdu,
659                                   m_phySapUser,
660                                   p);
661 //  m_phySapUser->ReceivePhyPdu (p);
662 }
663
664 void
665 MmWaveUePhy::SendDataChannels (Ptr<PacketBurst> pb, std::list<Ptr<MmWaveControlMessage> > ctrlMsg, Time duration, uint8_t
        slotInd)
666 {
667
668     //Ptr<AntennaArrayModel> antennaArray = DynamicCast<AntennaArrayModel> (GetDlSpectrumPhy ()->GetRxAntenna());
669     /* set beamforming vector;
670      * for UE, you can choose 16 antenna with 0-7 sectors, or 4 antenna with 0-3 sectors
671      * input is (sector, antenna number)
672      *
673      * */
674     //antennaArray->SetSector (3,16);
675
676     if (pb->GetNPackets() > 0)
677     {
678         LteRadioBearerTag tag;
679         if(!pb->GetPackets().front()->PeekPacketTag (tag))
680         {
681             NS_FATAL_ERROR ("No radio bearer tag");
682         }
683     }
684
685     m_downlinkSpectrumPhy->StartTxDataFrames (pb, ctrlMsg, duration, slotInd);
686 }
687
688 void
689 MmWaveUePhy::SendCtrlChannels (Ptr<PacketBurst> pb, std::list<Ptr<MmWaveControlMessage> > ctrlMsg, Time prd, uint8_t slotInd
        )
690 {
691     //std::cout << "MmWaveUePhy::SendCtrlChannels \n";
692     //Mudando para omnidireconal nas msgs de RACH
693     Ptr<AntennaArrayModel> antennaArray = DynamicCast<AntennaArrayModel> (GetDlSpectrumPhy ()->GetRxAntenna());
694         antennaArray->ChangeToOmniTx ();
695     std::list<Ptr<MmWaveControlMessage> >::iterator it;
696     for (it = ctrlMsg.begin (); it != ctrlMsg.end (); it++)
697     {
698         Ptr<MmWaveControlMessage> msg = (*it);
699
700     //if (msg->GetMessageType () == MmWaveControlMessage::RAR ||
701         //msg->GetMessageType() == MmWaveControlMessage::RACH_PREAMBLE)
702
703         //{
704             //std::cout<<"MUDOU PARA OMNIDIRECIONAL \n";
705             antennaArray->ChangeToOmniTx ();
706         //}
707     }
708
709     m_downlinkSpectrumPhy->StartTxDlControlFrames(pb,ctrlMsg,prd,slotInd);
710     NS_LOG_DEBUG("Entrei na mmwave-ue-phy e na fun  o SendCtrlChannels");
711 }
712
713
714 uint32_t
715 MmWaveUePhy::GetAbsoluteSubframeNo ()
716 {
717     return ((m_frameNum-1)*8 + m_slotNum);
718 }
719
720 Ptr<MmWaveDlCqiMessage>
721 MmWaveUePhy::CreateDlCqiFeedbackMessage (const SpectrumValue& sinr)
722 {
723     if (!m_amc)
724     {
725         m_amc = CreateObject <MmWaveAmc> (m_phyMacConfig);
726     }
727     NS_LOG_FUNCTION (this);
728     SpectrumValue newSinr = sinr;
729
730     //std::cout << "SINR DOWNLINK" << sinr << "\n";
731     // CREATE DlCqiLteControlMessage
732     Ptr<MmWaveDlCqiMessage> msg = Create<MmWaveDlCqiMessage> ();
733     DlCqiInfo dlcqi;
```

```
734
735         dlcqi.m_rnti = m_rnti;
736         dlcqi.m_cqiType = DlCqiInfo::WB;
737
738         std::vector<int> cqi;
739
740         //uint8_t dlBandwidth = m_phyMacConfig->GetNumChunkPerRb () * m_phyMacConfig->GetNumRb ();
741         NS_ASSERT (m_currSlot.m_dci.m_format==0);
742         int mcs;
743         dlcqi.m_wbCqi = m_amc->CreateCqiFeedbackWbTdma (newSinr, m_currSlot.m_dci.m_numSym, m_currSlot.m_dci.m_tbSize, mcs);
744
745 //      int activeSubChannels = newSinr.GetSpectrumModel()->GetNumBands ();
746         /*cqi = m_amc->CreateCqiFeedbacksTdma (newSinr, m_currNumSym);
747         int nbSubChannels = cqi.size ();
748         double cqiSum = 0.0;
749         // average the CQIs of the different RBs
750         for (int i = 0; i < nbSubChannels; i++)
751         {
752             if (cqi.at (i) != -1)
753             {
754                 cqiSum += cqi.at (i);
755                 activeSubChannels++;
756             }
757 //          NS_LOG_DEBUG (this << " subch " << i << " cqi " <<  cqi.at (i));
758         }*/
759 //      if (activeSubChannels > 0)
760 //      {
761 //          dlcqi.m_wbCqi = ((uint16_t) cqiSum / activeSubChannels);
762 //      }
763 //      else
764 //      {
765 //          // approximate with the worst case -> CQI = 1
766 //          dlcqi.m_wbCqi = 1;
767 //      }
768         msg->SetDlCqi (dlcqi);
769         return msg;
770     }
771
772     void
773     MmWaveUePhy::GenerateDlCqiReport (const SpectrumValue& sinr)
774     {
775         if(m_ulConfigured && (m_rnti > 0) && m_receptionEnabled)
776         {
777             if (Simulator::Now () > m_wbCqiLast + m_wbCqiPeriod)
778             {
779                 SpectrumValue newSinr = sinr;
780                 Ptr<MmWaveDlCqiMessage> msg = CreateDlCqiFeedbackMessage (newSinr);
781
782                 //std::cout << "SINR DOWNLINK" << sinr << "\n";
783                 //---------Acrescentei
784                 Values::const_iterator it;
785                 double sinrdbmedia;
786                 for (it = sinr.ConstValuesBegin (); it != sinr.ConstValuesEnd (); it++)
787                 {
788                         double sinrdb = 10 * std::log10 ((*it));
789                         sinrdbmedia = sinrdbmedia + sinrdb;
790
791                 }
792                 sinrdbmedia = sinrdbmedia/72;
793                 //std::cout<< "sinr (db) DOWNLINK= \t" << sinrdbmedia << "\n";
794                 //-----------------------------
795
796                 if (msg)
797                 {
798                     DoSendControlMessage (msg);
799                 }
800                 Ptr<MmWaveUeNetDevice> UeRx = DynamicCast<MmWaveUeNetDevice> (GetDevice());
801                 m_reportCurrentCellRsrpSinrTrace (UeRx->GetImsi(), newSinr, newSinr);
802             }
803         }
804     }
805
806     void
807     MmWaveUePhy::ReceiveLteDlHarqFeedback (DlHarqInfo m)
808     {
809       NS_LOG_FUNCTION (this);
810       // generate feedback to eNB and send it through ideal PUCCH
811       Ptr<MmWaveDlHarqFeedbackMessage> msg = Create<MmWaveDlHarqFeedbackMessage> ();
812       msg->SetDlHarqFeedback (m);
813       Simulator::Schedule (MicroSeconds(m_phyMacConfig->GetTbDecodeLatency()), &MmWaveUePhy::DoSendControlMessage, this, msg);
814 //    if (m.m_harqStatus == DlHarqInfo::NACK)  // Notify MAC/RLC
```

```
815   //   {
816   //       m_phySapUser->NotifyHarqDeliveryFailure (m.m_harqProcessId);
817   //   }
818   }
819
820   bool
821   MmWaveUePhy::IsReceptionEnabled ()
822   {
823       return m_receptionEnabled;
824   }
825
826   void
827   MmWaveUePhy::ResetReception()
828   {
829       m_receptionEnabled = false;
830   }
831
832   uint16_t
833   MmWaveUePhy::GetRnti ()
834   {
835       return m_rnti;
836   }
837
838
839   void
840   MmWaveUePhy::DoReset ()
841   {
842       NS_LOG_FUNCTION (this);
843   }
844
845   void
846   MmWaveUePhy::DoStartCellSearch (uint16_t dlEarfcn)
847   {
848       NS_LOG_FUNCTION (this << dlEarfcn);
849   }
850
851   void
852   MmWaveUePhy::DoSynchronizeWithEnb (uint16_t cellId, uint16_t dlEarfcn)
853   {
854       NS_LOG_FUNCTION (this << cellId << dlEarfcn);
855       DoSynchronizeWithEnb (cellId);
856   }
857
858   void
859   MmWaveUePhy::DoSetPa (double pa)
860   {
861     NS_LOG_FUNCTION (this << pa);
862   }
863
864
865   void
866   MmWaveUePhy::DoSynchronizeWithEnb (uint16_t cellId)
867   {
868       NS_LOG_FUNCTION (this << cellId);
869       if (cellId == 0)
870       {
871           NS_FATAL_ERROR ("Cell ID shall not be zero");
872       }
873   }
874
875   void
876   MmWaveUePhy::DoSetDlBandwidth (uint8_t dlBandwidth)
877   {
878       NS_LOG_FUNCTION (this << (uint32_t) dlBandwidth);
879   }
880
881
882   void
883   MmWaveUePhy::DoConfigureUplink (uint16_t ulEarfcn, uint8_t ulBandwidth)
884   {
885       NS_LOG_FUNCTION (this << ulEarfcn << ulBandwidth);
886     m_ulConfigured = true;
887   }
888
889   void
890   MmWaveUePhy::DoConfigureReferenceSignalPower (int8_t referenceSignalPower)
891   {
892       NS_LOG_FUNCTION (this << referenceSignalPower);
893   }
894
895   void
```

```
896 MmWaveUePhy::DoSetRnti (uint16_t rnti)
897 {
898     NS_LOG_FUNCTION (this << rnti);
899     m_rnti = rnti;
900 }
901
902 void
903 MmWaveUePhy::DoSetTransmissionMode (uint8_t txMode)
904 {
905     NS_LOG_FUNCTION (this << (uint16_t)txMode);
906 }
907
908 void
909 MmWaveUePhy::DoSetSrsConfigurationIndex (uint16_t srcCi)
910 {
911     NS_LOG_FUNCTION (this << srcCi);
912 }
913
914 void
915 MmWaveUePhy::SetPhySapUser (MmWaveUePhySapUser* ptr)
916 {
917     m_phySapUser = ptr;
918 }
919
920 void
921 MmWaveUePhy::SetHarqPhyModule (Ptr<MmWaveHarqPhy> harq)
922 {
923   m_harqPhyModule = harq;
924 }
925
926 }
```

Classe: mmwave-spectrum-phy.cc

```
1
2  /*
3   * mmwave-spectrum-phy.cc
4   *
5   *  Created on: Nov 5, 2014
6   *      Author: sourjya
7   */
8
9  #include <ns3/object-factory.h>
10 #include <ns3/log.h>
11 #include <ns3/ptr.h>
12 #include <ns3/boolean.h>
13 #include <cmath>
14 #include <ns3/simulator.h>
15 #include <ns3/trace-source-accessor.h>
16 #include <ns3/antenna-model.h>
17 #include "mmwave-spectrum-phy.h"
18 #include "mmwave-phy-mac-common.h"
19 #include <ns3/mmwave-enb-net-device.h>
20 #include <ns3/mmwave-ue-net-device.h>
21 #include <ns3/mmwave-ue-phy.h>
22 #include "mmwave-radio-bearer-tag.h"
23 #include <stdio.h>
24 #include <ns3/double.h>
25 #include <ns3/mmwave-mi-error-model.h>
26 #include "mmwave-mac-pdu-tag.h"
27
28 namespace ns3 {
29
30 NS_LOG_COMPONENT_DEFINE ("MmWaveSpectrumPhy");
31
32 NS_OBJECT_ENSURE_REGISTERED (MmWaveSpectrumPhy);
33
34 static const double bler_max=0.5;
35
36 static const double EffectiveCodingRate[29] = {
37   0.08,
38   0.1,
39   0.11,
40   0.15,
41   0.19,
42   0.24,
43   0.3,
```

```
44      0.37,
45      0.44,
46      0.51,
47      0.3,
48      0.33,
49      0.37,
50      0.42,
51      0.48,
52      0.54,
53      0.6,
54      0.43,
55      0.45,
56      0.5,
57      0.55,
58      0.6,
59      0.65,
60      0.7,
61      0.75,
62      0.8,
63      0.85,
64      0.89,
65      0.92
66  };
67
68  MmWaveSpectrumPhy::MmWaveSpectrumPhy()
69      :m_state(IDLE)
70  {
71      m_interferenceData = CreateObject<mmWaveInterference> ();
72      m_random = CreateObject<UniformRandomVariable> ();
73      m_random->SetAttribute ("Min", DoubleValue (0.0));
74      m_random->SetAttribute ("Max", DoubleValue (1.0));
75  }
76  MmWaveSpectrumPhy::~MmWaveSpectrumPhy()
77  {
78
79  }
80
81  TypeId
82  MmWaveSpectrumPhy::GetTypeId(void)
83  {
84      static TypeId
85          tid =
86          TypeId ("ns3::MmWaveSpectrumPhy")
87          .SetParent<NetDevice> ()
88          .AddTraceSource ("RxPacketTraceEnb",
89                           "The no. of packets received and transmitted by the Base Station",
90                           MakeTraceSourceAccessor (&MmWaveSpectrumPhy::m_rxPacketTraceEnb),
91                           "ns3::EnbTxRxPacketCount::TracedCallback")
92          .AddTraceSource ("RxPacketTraceUe",
93                           "The no. of packets received and transmitted by the User Device",
94                           MakeTraceSourceAccessor (&MmWaveSpectrumPhy::m_rxPacketTraceUe),
95                           "ns3::UeTxRxPacketCount::TracedCallback")
96          .AddAttribute ("DataErrorModelEnabled",
97                                      "Activate/Deactivate the error model of data (TBs of PDSCH and PUSCH) [by default is
                                            active].",
98                           BooleanValue (true),
99                           MakeBooleanAccessor (&MmWaveSpectrumPhy::m_dataErrorModelEnabled),
100                          MakeBooleanChecker ())
101         ;
102
103     return tid;
104 }
105 void
106 MmWaveSpectrumPhy::DoDispose()
107 {
108
109 }
110
111 void
112 MmWaveSpectrumPhy::SetDevice(Ptr<NetDevice> d)
113 {
114     m_device = d;
115
116     Ptr<MmWaveEnbNetDevice> enbNetDev =
117             DynamicCast<MmWaveEnbNetDevice> (GetDevice ());
118
119     if (enbNetDev != 0)
120     {
121         m_isEnb = true;
122         //std::cout << "m_isEnb=" << m_isEnb << "\n";
123     }
```

```
124       else
125       {
126           //std::cout << "m_isEnb=" << m_isEnb << "\n";
127           m_isEnb = false;
128       }
129   }
130
131   Ptr<NetDevice>
132   MmWaveSpectrumPhy::GetDevice() const
133   {
134       return m_device;
135       //std::cout << "device" << m_device << "\n";
136   }
137
138   void
139   MmWaveSpectrumPhy::SetMobility (Ptr<MobilityModel> m)
140   {
141       m_mobility = m;
142   }
143
144   Ptr<MobilityModel>
145   MmWaveSpectrumPhy::GetMobility ()
146   {
147       return m_mobility;
148   }
149
150   void
151   MmWaveSpectrumPhy::SetChannel (Ptr<SpectrumChannel> c)
152   {
153       m_channel = c;
154   }
155
156   Ptr<const SpectrumModel>
157   MmWaveSpectrumPhy::GetRxSpectrumModel () const
158   {
159       return m_rxSpectrumModel;
160   }
161
162   Ptr<AntennaModel>
163   MmWaveSpectrumPhy::GetRxAntenna ()
164   {
165       return m_antenna;
166   }
167
168   void
169   MmWaveSpectrumPhy::SetAntenna (Ptr<AntennaModel> a)
170   {
171       m_antenna = a;
172   }
173
174   void
175   MmWaveSpectrumPhy::SetState (State newState)
176   {
177     ChangeState (newState);
178   }
179
180   void
181   MmWaveSpectrumPhy::ChangeState (State newState)
182   {
183     NS_LOG_LOGIC (this << " state: " << m_state << " -> " << newState);
184     m_state = newState;
185   }
186
187
188   void
189   MmWaveSpectrumPhy::SetNoisePowerSpectralDensity(Ptr<const SpectrumValue> noisePsd)
190   {
191       NS_LOG_FUNCTION (this << noisePsd);
192       NS_ASSERT (noisePsd);
193       m_rxSpectrumModel = noisePsd->GetSpectrumModel ();
194       m_interferenceData->SetNoisePowerSpectralDensity (noisePsd);
195
196   }
197
198   void
199   MmWaveSpectrumPhy::SetTxPowerSpectralDensity (Ptr<SpectrumValue> TxPsd)
200   {
201       m_txPsd = TxPsd;
202   }
203
204   void
```

```
205 MmWaveSpectrumPhy::SetPhyRxDataEndOkCallback (MmWavePhyRxDataEndOkCallback c)
206 {
207     m_phyRxDataEndOkCallback = c;
208 }
209
210
211 void
212 MmWaveSpectrumPhy::SetPhyRxCtrlEndOkCallback (MmWavePhyRxCtrlEndOkCallback c)
213 {
214   m_phyRxCtrlEndOkCallback = c;
215 }
216
217 void
218 MmWaveSpectrumPhy::AddExpectedTb (uint16_t rnti, uint8_t ndi, uint16_t size, uint8_t mcs,
219                                   std::vector<int> chunkMap, uint8_t harqId, uint8_t rv, bool downlink,
220                                   uint8_t symStart, uint8_t numSym)
221 {
222     //layer = layer;
223     ExpectedTbMap_t::iterator it;
224     it = m_expectedTbs.find (rnti);
225     if (it != m_expectedTbs.end ())
226     {
227         m_expectedTbs.erase (it);
228     }
229     // insert new entry
230     //ExpectedTbInfo_t tbInfo = {ndi, size, mcs, chunkMap, harqId, rv, 0.0, downlink, false, false, 0};
231     ExpectedTbInfo_t tbInfo = {ndi, size, mcs, chunkMap, harqId, rv, 0.0, downlink, false, false, 0, symStart, numSym};
232     m_expectedTbs.insert (std::pair<uint16_t, ExpectedTbInfo_t> (rnti,tbInfo));
233 }
234
235 /*
236 void
237 MmWaveSpectrumPhy::AddExpectedTb (uint16_t rnti, uint16_t size, uint8_t mcs, std::vector<int> chunkMap, bool downlink)
238 {
239     //layer = layer;
240     ExpectedTbMap_t::iterator it;
241     it = m_expectedTbs.find (rnti);
242     if (it != m_expectedTbs.end ())
243     {
244         m_expectedTbs.erase (it);
245     }
246     // insert new entry
247     ExpectedTbInfo_t tbInfo = {1, size, mcs, chunkMap, 0, 0, 0.0, downlink, false, false};
248     m_expectedTbs.insert (std::pair<uint16_t, ExpectedTbInfo_t> (rnti,tbInfo));
249 }
250 */
251
252 void
253 MmWaveSpectrumPhy::SetPhyDlHarqFeedbackCallback (MmWavePhyDlHarqFeedbackCallback c)
254 {
255   NS_LOG_FUNCTION (this);
256   m_phyDlHarqFeedbackCallback = c;
257 }
258
259 void
260 MmWaveSpectrumPhy::SetPhyUlHarqFeedbackCallback (MmWavePhyUlHarqFeedbackCallback c)
261 {
262   NS_LOG_FUNCTION (this);
263   m_phyUlHarqFeedbackCallback = c;
264 }
265
266 void
267 MmWaveSpectrumPhy::StartRx (Ptr<SpectrumSignalParameters> params)
268 {
269    //std::cout << "m_state do StartRX == " << m_state << "\n";
270
271     NS_LOG_FUNCTION(this);
272
273     Ptr<MmWaveEnbNetDevice> EnbTx =
274             DynamicCast<MmWaveEnbNetDevice> (params->txPhy->GetDevice ());
275     Ptr<MmWaveEnbNetDevice> enbRx =
276             DynamicCast<MmWaveEnbNetDevice> (GetDevice ());
277     if((EnbTx != 0 && enbRx != 0) || (EnbTx == 0 && enbRx == 0))
278     {
279         NS_LOG_INFO ("BS to BS or UE to UE transmission neglected.");
280         return;
281     }
282
283     Ptr<MmwaveSpectrumSignalParametersDataFrame> mmwaveDataRxParams =
284             DynamicCast<MmwaveSpectrumSignalParametersDataFrame> (params);
285
```

```
286       Ptr<MmWaveSpectrumSignalParametersDlCtrlFrame> DlCtrlRxParams =
287               DynamicCast<MmWaveSpectrumSignalParametersDlCtrlFrame> (params);
288
289       if (mmwaveDataRxParams!=0)
290       {
291           //std::cout<< "mmwaveDataRxParams!=0 \n";
292           bool isAllocated = true;
293           Ptr<MmWaveUeNetDevice> ueRx = 0;
294           ueRx = DynamicCast<MmWaveUeNetDevice> (GetDevice ());
295
296           if ((ueRx!=0) && (ueRx->GetPhy ()->IsReceptionEnabled () == false))
297           {
298               isAllocated = false;
299           }
300
301           if (isAllocated)
302           {
303               m_interferenceData->AddSignal (mmwaveDataRxParams->psd, mmwaveDataRxParams->duration);
304               //std::cout<< "m_interferenceData->AddSignal \n";
305               if(mmwaveDataRxParams->cellId == m_cellId)
306               {
307                   //m_interferenceData->AddSignal (mmwaveDataRxParams->psd, mmwaveDataRxParams->duration);
308                   StartRxData (mmwaveDataRxParams);
309               }
310               /*
311               else
312               {
313                   if (ueRx != 0)
314                   {
315                       m_interferenceData->AddSignal (mmwaveDataRxParams->psd, mmwaveDataRxParams->duration);
316                   }
317               }
318               */
319           }
320       }
321       else
322       {
323           //std::cout<< "mmwaveDataRxParams=0 \n";
324
325           if (DlCtrlRxParams!=0)
326           {
327               //std::cout<< "DlCtrlRxParams->duration" << DlCtrlRxParams->duration << "\n";
328               //std::cout<< "m_interferenceData->AddSignal \n";
329               m_interferenceData->AddSignal (DlCtrlRxParams->psd, DlCtrlRxParams->duration);// Acrescentei essa linha
330               //std::cout<< "m_interferenceData->AddSignal \n";
331               //  std::cout<< "PSD: \t" << (*DlCtrlRxParams->psd) << "\n";
332               if (DlCtrlRxParams->cellId == m_cellId)
333               {
334                   //StartRxCtrl (params);
335                   //std::cout << "StartRxCtrl (DlCtrlRxParams) \n";
336                   ChangeState (IDLE); //Acrescentei essa linha
337                   StartRxCtrl (DlCtrlRxParams); //Alterei aqui
338               }
339               else
340               {
341                   // Do nothing
342               }
343           }
344       }
345   }
346
347   void
348   MmWaveSpectrumPhy::StartRxData (Ptr<MmwaveSpectrumSignalParametersDataFrame> params)
349   {
350       m_interferenceData->StartRx (params->psd);
351
352       NS_LOG_FUNCTION(this);
353
354       Ptr<MmWaveEnbNetDevice> enbRx =
355                   DynamicCast<MmWaveEnbNetDevice> (GetDevice ());
356       Ptr<MmWaveUeNetDevice> ueRx =
357                   DynamicCast<MmWaveUeNetDevice> (GetDevice ());
358       switch(m_state)
359       {
360       case TX:
361           NS_FATAL_ERROR("Cannot receive while transmitting");
362           break;
363       case RX_CTRL:
364               NS_FATAL_ERROR("Cannot receive control in data period");
365               break;
366       case RX_DATA:
```

```
367       case IDLE:
368       {
369           if (params->cellId == m_cellId)
370           {
371               if (m_rxPacketBurstList.empty())
372               {
373           NS_ASSERT (m_state == IDLE);
374                   // first transmission, i.e., we re IDLE and we start RX
375                   m_firstRxStart = Simulator::Now ();
376                   m_firstRxDuration = params->duration;
377                   NS_LOG_LOGIC (this << " scheduling EndRx with delay " << params->duration.GetSeconds () << "s");
378
379                   Simulator::Schedule (params->duration, &MmWaveSpectrumPhy::EndRxData, this);
380               }
381               else
382               {
383                   NS_ASSERT (m_state == RX_DATA);
384                   // sanity check: if there are multiple RX events, they
385                   // should occur at the same time and have the same
386                   // duration, otherwise the interference calculation
387                   // won t be correct
388                   NS_ASSERT ((m_firstRxStart == Simulator::Now ()) && (m_firstRxDuration == params->duration));
389               }
390
391               ChangeState (RX_DATA);
392               if (params->packetBurst && !params->packetBurst->GetPackets ().empty ())
393               {
394                   m_rxPacketBurstList.push_back (params->packetBurst);
395               }
396               //NS_LOG_DEBUG (this << " insert msgs " << params->ctrlMsgList.size ());
397               m_rxControlMessageList.insert (m_rxControlMessageList.end (), params->ctrlMsgList.begin (), params->ctrlMsgList.
                      end ());
398
399               NS_LOG_LOGIC (this << " numSimultaneousRxEvents = " << m_rxPacketBurstList.size ());
400           }
401           else
402           {
403               NS_LOG_LOGIC (this << " not in sync with this signal (cellId="
404                             << params->cellId  << ", m_cellId=" << m_cellId << ")");
405           }
406       }
407       break;
408       default:
409           NS_FATAL_ERROR("Programming Error: Unknown State");
410       }
411   }
412   //void
413   //MmWaveSpectrumPhy::StartRxCtrl (Ptr<SpectrumSignalParameters> params)
414   void
415   MmWaveSpectrumPhy::StartRxCtrl (Ptr<MmWaveSpectrumSignalParametersDlCtrlFrame> params) //Alterei aqui o argumento da
          fun   o
416   {
417       //std::cout << "m_state == " << m_state << "\n";
418       //std::cout << "MmWaveSpectrumPhy::StartRxCtrl \n";
419       NS_LOG_FUNCTION (this);
420       // RDF: method currently supports Downlink control only!
421       switch (m_state)
422       {
423           case TX:
424               NS_FATAL_ERROR ("Cannot RX while TX: according to FDD channel access, the physical layer for transmission cannot
                      be used for reception");
425               break;
426           case RX_DATA:
427               NS_FATAL_ERROR ("Cannot RX data while receiving control");
428               break;
429           case RX_CTRL:
430           case IDLE:
431           {
432               // the behavior is similar when we re IDLE or RX because we can receive more signals
433               // simultaneously (e.g., at the eNB).
434               Ptr<MmWaveSpectrumSignalParametersDlCtrlFrame> dlCtrlRxParams = \
435                       DynamicCast<MmWaveSpectrumSignalParametersDlCtrlFrame> (params);
436
437
438               //std::cout << "sinr = " << m_sinrPerceived << "\n";
439               // To check if we re synchronized to this signal, we check for the CellId
440               uint16_t cellId = 0;
441               if (dlCtrlRxParams != 0)
442               {
443                   cellId = dlCtrlRxParams->cellId;
444               }
```

```
445             else
446             {
447                 NS_LOG_ERROR ("SpectrumSignalParameters type not supported");
448             }
449             // check presence of PSS for UE measuerements
450             /*if (dlCtrlRxParams->pss == true)
451             {
452                 SpectrumValue pssPsd = *params->psd;
453                 if (!m_phyRxPssCallback.IsNull ())
454                 {
455                     m_phyRxPssCallback (cellId, params->psd);
456                 }
457             }*/
458             if (cellId  == m_cellId)
459             {
460
461                 //**** Acrescentei o IF abaixo:
462                 /*if (m_rxPacketBurstList.empty())
463             {
464                 NS_ASSERT (m_state == IDLE);
465                 // first transmission, i.e., we re IDLE and we start RX
466                 m_firstRxStart = Simulator::Now ();
467                 m_firstRxDuration = params->duration;
468                 NS_LOG_LOGIC (this << " scheduling EndRx with delay " << params->duration.GetSeconds () << "s");
469
470                 Simulator::Schedule (params->duration, &MmWaveSpectrumPhy::EndRxCtrl, this);
471             }*/
472             //*********
473
474                 if(m_state == RX_CTRL)
475                 {
476                     //std::cout << "Entrou no IF m_state == RX_CTRL \n";
477                     Ptr<MmWaveUeNetDevice> ueRx =
478                                     DynamicCast<MmWaveUeNetDevice> (GetDevice ());
479                     if (ueRx)
480                     {
481                         NS_FATAL_ERROR ("UE already receiving control data from serving cell");
482                     }
483                     NS_ASSERT ((m_firstRxStart == Simulator::Now ())
484                                && (m_firstRxDuration == params->duration));
485                 }
486                 NS_LOG_LOGIC (this << " synchronized with this signal (cellId=" << cellId << ")");
487
488                 if (m_state == IDLE)
489                 {
490                     //std::cout << "Entrou no IF m_state == IDLE \n";
491                     // first transmission, i.e., we re IDLE and we start RX
492                     //NS_ASSERT (m_rxControlMessageList.empty ());//Comentei aqui, estava dando erro
493                     m_firstRxStart = Simulator::Now ();
494                     m_firstRxDuration = params->duration;
495                     NS_LOG_LOGIC (this << " scheduling EndRx with delay " << params->duration);
496                     // store the DCIs
497                     m_rxControlMessageList = dlCtrlRxParams->ctrlMsgList;
498                     Simulator::Schedule (params->duration, &MmWaveSpectrumPhy::EndRxCtrl, this);
499                     ChangeState (RX_CTRL); //Alterei aqui comentando essa linha
500                     //std::cout << " m_interferenceData->StartRx \n ";
501                     m_interferenceData->StartRx (dlCtrlRxParams->psd); //Acrescentei essa fun   o aqui
502                     //std::cout<< "PSD: \t" << (*dlCtrlRxParams->psd);
503
504                     //********** Acrescentei o IF abaixo
505                     if (params->packetBurst && !params->packetBurst->GetPackets ().empty ())
506                     {
507                         //std::cout << " entrei no if do packetBurst \n";
508                         m_rxPacketBurstList.push_back (params->packetBurst);
509                     }
510                     //**********
511                 }
512
513
514                 else
515                 {
516                     //std::cout << "Entrou no ELSE m_rxControlMessageList.insert \n";
517                     m_rxControlMessageList.insert (m_rxControlMessageList.end (), dlCtrlRxParams->ctrlMsgList.begin (),
                                                    dlCtrlRxParams->ctrlMsgList.end ());
518                 }
519
520             }
521             break;
522         }
523         default:
524         {
```

```
525             NS_FATAL_ERROR ("unknown state");
526             break;
527         }
528         }
529     }
530
531     void
532     MmWaveSpectrumPhy::EndRxData ()
533     {
534         //std::cout << "entrou no MmWaveSpectrumPhy::EndRxData";
535         m_interferenceData->EndRx();
536
537         double sinrAvg = Sum(m_sinrPerceived)/(m_sinrPerceived.GetSpectrumModel()->GetNumBands());
538         double sinrMin = 99999999999;
539         for (Values::const_iterator it = m_sinrPerceived.ConstValuesBegin (); it != m_sinrPerceived.ConstValuesEnd (); it++)
540         {
541             if (*it < sinrMin)
542             {
543                 sinrMin = *it;
544             }
545         }
546
547         Ptr<MmWaveEnbNetDevice> enbRx = DynamicCast<MmWaveEnbNetDevice> (GetDevice ());
548         Ptr<MmWaveUeNetDevice> ueRx = DynamicCast<MmWaveUeNetDevice> (GetDevice ());
549
550         NS_ASSERT(m_state = RX_DATA);
551         ExpectedTbMap_t::iterator itTb = m_expectedTbs.begin ();
552         while (itTb != m_expectedTbs.end ())
553         {
554             //std::cout << "entrou no MmWaveSpectrumPhy::EndRxData";
555             if ((m_dataErrorModelEnabled)&&(m_rxPacketBurstList.size ()>0))
556             {
557                 HarqProcessInfoList_t harqInfoList;
558                 uint8_t rv = 0;
559                 if (itTb->second.ndi == 0)
560                 {
561                     // TB retxed: retrieve HARQ history
562                     if (itTb->second.downlink)
563                     {
564                         harqInfoList = m_harqPhyModule->GetHarqProcessInfoDl (itTb->first, itTb->second.harqProcessId);
565                     }
566                     else
567                     {
568                         harqInfoList = m_harqPhyModule->GetHarqProcessInfoUl (itTb->first, itTb->second.harqProcessId);
569                     }
570                     if (harqInfoList.size () > 0)
571                     {
572                         rv = harqInfoList.back ().m_rv;
573                     }
574                 }
575                 //std::cout << "MmWaveSpectrumPhy::EndRxData () \n";
576                 TbStats_t tbStats = MmWaveMiErrorModel::GetTbDecodificationStats (m_sinrPerceived,
577                         itTb->second.rbBitmap, itTb->second.size, itTb->second.mcs, harqInfoList);
578                 itTb->second.tbler = tbStats.tbler;
579                 itTb->second.mi = tbStats.miTotal;
580                 itTb->second.corrupt = m_random->GetValue () > tbStats.tbler ? false : true;
581                 if (itTb->second.corrupt)
582                 {
583                     NS_LOG_INFO (this << " RNTI " << itTb->first << " size " << itTb->second.size << " mcs " << (uint32_t)itTb->
                                 second.mcs << " bitmap " << itTb->second.rbBitmap.size () << " rv " << rv << " TBLER " << tbStats.
                                 tbler << " corrupted " << itTb->second.corrupt);
584                 }
585             }
586             itTb++;
587         }
588
589         std::map <uint16_t, DlHarqInfo> harqDlInfoMap;
590         for (std::list<Ptr<PacketBurst> >::const_iterator i = m_rxPacketBurstList.begin ();
591                 i != m_rxPacketBurstList.end (); ++i)
592         {
593             for (std::list<Ptr<Packet> >::const_iterator j = (*i)->Begin (); j != (*i)->End (); ++j)
594             {
595                 if ((*j)->GetSize () == 0)
596                 {
597                     continue;
598                 }
599
600                 LteRadioBearerTag bearerTag;
601                 if((*j)->PeekPacketTag (bearerTag) == false)
602                 {
603                     NS_FATAL_ERROR ("No radio bearer tag found");
```

```
604             }
605             uint16_t rnti = bearerTag.GetRnti ();
606
607             itTb = m_expectedTbs.find (rnti);
608             if(itTb != m_expectedTbs.end ())
609             {
610                 if (!itTb->second.corrupt)
611                 {
612                     m_phyRxDataEndOkCallback (*j);
613                 }
614                 else
615                 {
616                     NS_LOG_INFO ("TB failed");
617                 }
618
619                 MmWaveMacPduTag pduTag;
620                 if((*j)->PeekPacketTag (pduTag) == false)
621                 {
622                     NS_FATAL_ERROR ("No radio bearer tag found");
623                 }
624
625                 RxPacketTraceParams traceParams;
626                 traceParams.m_tbSize = itTb->second.size;
627                 traceParams.m_cellId = 0;
628                 traceParams.m_frameNum = pduTag.GetSfn ().m_frameNum;
629                 traceParams.m_sfNum = pduTag.GetSfn ().m_sfNum;
630                 traceParams.m_slotNum = pduTag.GetSfn ().m_slotNum;
631                 traceParams.m_rnti = rnti;
632                 traceParams.m_mcs = itTb->second.mcs;
633                 traceParams.m_rv = itTb->second.rv;
634                 traceParams.m_sinr = sinrAvg;
635                 traceParams.m_sinrMin = itTb->second.mi;//sinrMin;
636                 traceParams.m_tbler = itTb->second.tbler;
637                 traceParams.m_corrupt = itTb->second.corrupt;
638                 traceParams.m_symStart = itTb->second.symStart;
639                 traceParams.m_numSym = itTb->second.numSym;
640
641                 if (enbRx)
642                 {
643                     m_rxPacketTraceEnb (traceParams);
644
645                 }
646                 else if (ueRx)
647                 {
648                     m_rxPacketTraceUe (traceParams);
649
650                 }
651
652                 // send HARQ feedback (if not already done for this TB)
653                 if (!itTb->second.harqFeedbackSent)
654                 {
655                     itTb->second.harqFeedbackSent = true;
656                     if (!itTb->second.downlink)  // UPLINK TB
657                     {
658                         //double sinrdb = 10 * std::log10 ((sinrAvg));
659                        // std::cout<< "DATA sinr (db) UPLINK= \t" << sinrdb << "\n";
660                         UlHarqInfo harqUlInfo;
661                         harqUlInfo.m_rnti = rnti;
662                         harqUlInfo.m_tpc = 0;
663                         harqUlInfo.m_harqProcessId = itTb->second.harqProcessId;
664                         harqUlInfo.m_numRetx = itTb->second.rv;
665                         if (itTb->second.corrupt)
666                         {
667                             harqUlInfo.m_receptionStatus = UlHarqInfo::NotOk;
668                             NS_LOG_DEBUG ("UE" << rnti << " send UL-HARQ-NACK" << " harqId " << (unsigned)itTb->second.
                                    harqProcessId <<
669                                                 " size " << itTb->second.size << " mcs " << (unsigned)itTb->second.
                                                     mcs <<
670                                                 " mi " << itTb->second.mi << " tbler " << itTb->second.tbler << "
                                                     SINRavg " << sinrAvg);
671                             m_harqPhyModule->UpdateUlHarqProcessStatus (rnti, itTb->second.harqProcessId, itTb->second.mi,
                                    itTb->second.size, itTb->second.size / EffectiveCodingRate [itTb->second.mcs]);
672                         }
673                         else
674                         {
675                             harqUlInfo.m_receptionStatus = UlHarqInfo::Ok;
676 //                            NS_LOG_DEBUG ("UE" << rnti << " send UL-HARQ-ACK" << " harqId " << (unsigned)itTb->second.
        harqProcessId <<
677 //                                                " size " << itTb->second.size << " mcs " << (unsigned)itTb->second.
        mcs <<
678 //                                                " mi " << itTb->second.mi << " tbler " << itTb->second.tbler << "
```

```
        SINRavg " << sinrAvg);
679                             m_harqPhyModule->ResetUlHarqProcessStatus (rnti, itTb->second.harqProcessId);
680                         }
681                         if (!m_phyUlHarqFeedbackCallback.IsNull ())
682                         {
683                             m_phyUlHarqFeedbackCallback (harqUlInfo);
684                         }
685                     }
686                     else
687                     {
688                         //double sinrdb = 10 * std::log10 ((sinrAvg));
689                         //std::cout<< "DATA sinr (db) DOWNLINK= \t" << sinrdb << "\n";
690                         std::map <uint16_t, DlHarqInfo>::iterator itHarq = harqDlInfoMap.find (rnti);
691                         if (itHarq==harqDlInfoMap.end ())
692                         {
693                             DlHarqInfo harqDlInfo;
694                             harqDlInfo.m_harqStatus = DlHarqInfo::NACK;
695                             harqDlInfo.m_rnti = rnti;
696                             harqDlInfo.m_harqProcessId = itTb->second.harqProcessId;
697                             harqDlInfo.m_numRetx = itTb->second.rv;
698                             if (itTb->second.corrupt)
699                             {
700                                 harqDlInfo.m_harqStatus = DlHarqInfo::NACK;
701                                 NS_LOG_DEBUG ("UE" << rnti << " send DL-HARQ-NACK" << " harqId " << (unsigned)itTb->second.
                                        harqProcessId <<
702                                                                                 " size " << itTb->second.size << " mcs " << (unsigned)itTb->
                                                                                     second.mcs <<
703                                                                                 " mi " << itTb->second.mi << " tbler " << itTb->second.tbler <<
                                                                                      " SINRavg " << sinrAvg);
704                                 m_harqPhyModule->UpdateDlHarqProcessStatus (rnti, itTb->second.harqProcessId, itTb->second.
                                        mi, itTb->second.size, itTb->second.size / EffectiveCodingRate [itTb->second.mcs]);
705                             }
706                             else
707                             {
708                                 harqDlInfo.m_harqStatus = DlHarqInfo::ACK;
709 //                                NS_LOG_DEBUG ("UE" << rnti << " send DL-HARQ-ACK" << " harqId " << (unsigned)itTb->second.
    harqProcessId <<
710 //                                                                                " size " << itTb->second.size << " mcs " << (unsigned)itTb->
    second.mcs <<
711 //                                                                                " mi " << itTb->second.mi << " tbler " << itTb->second.tbler <<
    " SINRavg " << sinrAvg);
712                                 m_harqPhyModule->ResetDlHarqProcessStatus (rnti, itTb->second.harqProcessId);
713                             }
714                             harqDlInfoMap.insert (std::pair <uint16_t, DlHarqInfo> (rnti, harqDlInfo));
715                         }
716                         else
717                         {
718                             if (itTb->second.corrupt)
719                             {
720                                 (*itHarq).second.m_harqStatus = DlHarqInfo::NACK;
721                                 NS_LOG_DEBUG ("UE" << rnti << " send DL-HARQ-NACK" << " harqId " << (unsigned)itTb->second.
                                        harqProcessId <<
722                                                                                 " size " << itTb->second.size << " mcs " << (unsigned)itTb->
                                                                                     second.mcs <<
723                                                                                 " mi " << itTb->second.mi << " tbler " << itTb->second.tbler <<
                                                                                      " SINRavg " << sinrAvg);
724                                 m_harqPhyModule->UpdateDlHarqProcessStatus (rnti, itTb->second.harqProcessId, itTb->second.
                                        mi, itTb->second.size, itTb->second.size / EffectiveCodingRate [itTb->second.mcs]);
725                             }
726                             else
727                             {
728                                 (*itHarq).second.m_harqStatus = DlHarqInfo::ACK;
729 //                                NS_LOG_DEBUG ("UE" << rnti << " send DL-HARQ-ACK" << " harqId " << (unsigned)itTb->second.
    harqProcessId <<
730 //                                                                                " size " << itTb->second.size << " mcs " << (unsigned)itTb->second.mcs <<
731 //                                                                                " mi " << itTb->second.mi << " tbler " << itTb->second.tbler << " SINRavg " <<
     sinrAvg);
732                                 m_harqPhyModule->ResetDlHarqProcessStatus (rnti, itTb->second.harqProcessId);
733                             }
734                         }
735                     } // end if (itTb->second.downlink) HARQ
736                 } // end if (!itTb->second.harqFeedbackSent)
737             }
738             else
739             {
740                 //              NS_FATAL_ERROR ("End of the tbMap");
741                 // Packet is for other device
742             }
743
744
745         }
```

```
746         }
747
748         // send DL HARQ feedback to LtePhy
749         std::map <uint16_t, DlHarqInfo>::iterator itHarq;
750         for (itHarq = harqDlInfoMap.begin (); itHarq != harqDlInfoMap.end (); itHarq++)
751         {
752             if (!m_phyDlHarqFeedbackCallback.IsNull ())
753             {
754                 m_phyDlHarqFeedbackCallback ((*itHarq).second);
755             }
756         }
757         // forward control messages of this frame to MmWavePhy
758
759         if (!m_rxControlMessageList.empty () && !m_phyRxCtrlEndOkCallback.IsNull ())
760         {
761             m_phyRxCtrlEndOkCallback (m_rxControlMessageList);
762         }
763
764         m_state = IDLE;
765         m_rxPacketBurstList.clear ();
766         m_expectedTbs.clear ();
767         m_rxControlMessageList.clear ();
768     }
769
770     void
771     MmWaveSpectrumPhy::EndRxCtrl ()
772     {
773         //Alterei essa fun   o praticamente toda
774         //std::cout << "MmWaveSpectrumPhy::EndRxCtrl () \n";
775         m_interferenceData->EndRx();
776         bool error = false;
777
778         /*double sinrAvg = Sum(m_sinrPerceived)/(72);
779         double sinrdB = (10 * std::log10 (sinrAvg));
780         std::cout << "sinrdB =" << sinrdB << "\n";*/
781
782         Ptr<MmWaveEnbNetDevice> enbRx =
783                 DynamicCast<MmWaveEnbNetDevice> (GetDevice ());
784         Ptr<MmWaveUeNetDevice> ueRx =
785                 DynamicCast<MmWaveUeNetDevice> (GetDevice ());
786
787         //std::cout << "enbRx" << enbRx << "\n";
788         //std::cout << "ueRx" << ueRx << "\n";
789
790         /*
791         if (m_isEnb == 1)
792         {
793         //error = false;
794         //std::cout << "m_isEnb do EndRxCtrl=" << m_isEnb << "\n";
795
796         double sinrAvg = Sum(m_sinrPerceived)/(72);
797         double sinrdB = (10 * std::log10 (sinrAvg));
798         //std::cout << " m_sinrPerceived RACH=" << m_sinrPerceived<< "\n";
799         std::cout << "sinrdB RACH =" << sinrdB << "\n";
800
801         }
802
803         else
804         {
805
806             double sinrAvg = Sum(m_sinrPerceived)/(72);
807             double sinrdB = (10 * std::log10 (sinrAvg));
808             //std::cout << " m_sinrPerceived RAR=" << m_sinrPerceived<< "\n";
809             std::cout << "sinrdB RAR =" << sinrdB << "\n";
810             //std::cout << "m_isEnb do EndRxCtrl=" << m_isEnb << "\n";
811         }*/
812           //-----------------------------------------------------------------------
813         ExpectedTbMap_t::iterator itTb = m_expectedTbs.begin ();
814         while (itTb != m_expectedTbs.end ())
815         {
816
817             //std::cout << "entrou no MmWaveSpectrumPhy::EndRxData";
818             if ((m_dataErrorModelEnabled)&&(m_rxPacketBurstList.size ()>0))
819             {
820                 HarqProcessInfoList_t harqInfoList;
821
822                 if (itTb->second.ndi == 0)
823                 {
824                     // TB retxed: retrieve HARQ history
825                     if (itTb->second.downlink)
826                     {
```

```
827                     harqInfoList = m_harqPhyModule->GetHarqProcessInfoDl (itTb->first, itTb->second.harqProcessId);
828                 }
829                 else
830                 {
831                     harqInfoList = m_harqPhyModule->GetHarqProcessInfoUl (itTb->first, itTb->second.harqProcessId);
832                 }
833 
834             }
835             //std::cout << "MmWaveSpectrumPhy::EndRxData () \n";
836             TbStats_t tbStats = MmWaveMiErrorModel::GetTbDecodificationStats (m_sinrPerceived,
837                     itTb->second.rbBitmap, itTb->second.size, itTb->second.mcs, harqInfoList);
838             itTb->second.tbler = tbStats.tbler;
839             //std::cout << "itTb-> second.tbler ="<< itTb->second.tbler << "\n" ;
840             itTb->second.mi = tbStats.miTotal;
841             itTb->second.corrupt = m_random->GetValue () > tbStats.tbler ? false : true;
842             if (itTb->second.corrupt)
843             {
844                 error = true;
845                 //std::cout << "error = true \n" ;
846 
847             }
848             else
849             {
850                 //std::cout << "error = false \n" ;
851             }
852         }
853         itTb++;
854     }
855 
856       //-----------------------------------------------------------------------------
857 
858         /*if (sinrdB < (-14.0))
859             //if (itTb->second.corrupt)
860             {
861 
862                 error = true;
863               //  std::cout << "error = true \n" ;
864             }
865         else
866             {
867 
868                 error = false;
869              //   std::cout << "error = false \n" ;
870             }*/
871 
872 
873 
874     if (!error)
875     {
876     //std::cout << "Encaminha mensagens de CTRL \n";
877         if (!m_rxControlMessageList.empty ())
878         {
879             //std::cout << "entrou no IF \n";
880             if (!m_phyRxCtrlEndOkCallback.IsNull ())
881                 {
882                 m_phyRxCtrlEndOkCallback (m_rxControlMessageList); //Encaminha msgs para a camada PHY
883 
884                 }
885         }
886     }
887     else
888     {
889         //std::cout << "N o encaminha mensagens de CTRL \n";
890     }
891 
892 
893   //NS_ASSERT(m_state = RX_CTRL);
894 
895 
896  // std::cout << "m_state = IDLE \n";
897   m_state = IDLE;
898   m_rxControlMessageList.clear ();
899   m_rxPacketBurstList.clear ();
900   m_expectedTbs.clear ();
901 
902 }
903 
904 bool
905 MmWaveSpectrumPhy::StartTxDataFrames (Ptr<PacketBurst> pb, std::list<Ptr<MmWaveControlMessage> > ctrlMsgList, Time duration,
        uint8_t slotInd)
906 {
```

```
907       switch (m_state)
908       {
909       case RX_DATA:
910     case RX_CTRL:
911           NS_FATAL_ERROR ("cannot TX while RX: Cannot transmit while receiving");
912           break;
913       case TX:
914           NS_FATAL_ERROR ("cannot TX while already Tx: Cannot transmit while a transmission is still on");
915           break;
916       case IDLE:
917       {
918           NS_ASSERT(m_txPsd);
919
920           m_state = TX;
921           Ptr<MmwaveSpectrumSignalParametersDataFrame> txParams = new MmwaveSpectrumSignalParametersDataFrame ();
922           txParams->duration = duration;
923           txParams->txPhy = this->GetObject<SpectrumPhy> ();
924           txParams->psd = m_txPsd;
925           txParams->packetBurst = pb;
926           txParams->cellId = m_cellId;
927           txParams->ctrlMsgList = ctrlMsgList;
928           txParams->slotInd = slotInd;
929           txParams->txAntenna = m_antenna;
930
931           //NS_LOG_DEBUG ("ctrlMsgList.size () == " << txParams->ctrlMsgList.size ());
932
933           /* This section is used for trace */
934           Ptr<MmWaveEnbNetDevice> enbTx =
935                       DynamicCast<MmWaveEnbNetDevice> (GetDevice ());
936           Ptr<MmWaveUeNetDevice> ueTx =
937                       DynamicCast<MmWaveUeNetDevice> (GetDevice ());
938   //      if (enbTx)
939   //      {
940   //          EnbPhyPacketCountParameter traceParam;
941   //          traceParam.m_noBytes = (txParams->packetBurst)?txParams->packetBurst->GetSize ():0;
942   //          traceParam.m_cellId = txParams->cellId;
943   //          traceParam.m_isTx = true;
944   //          traceParam.m_subframeno = enbTx->GetPhy ()->GetAbsoluteSubframeNo ();
945   //          m_reportEnbPacketCount (traceParam);
946   //      }
947   //      else if (ueTx)
948   //      {
949   //          UePhyPacketCountParameter traceParam;
950   //          traceParam.m_noBytes = (txParams->packetBurst)?txParams->packetBurst->GetSize ():0;
951   //          traceParam.m_imsi = ueTx->GetImsi ();
952   //          traceParam.m_isTx = true;
953   //          traceParam.m_subframeno = ueTx->GetPhy ()->GetAbsoluteSubframeNo ();
954   //          m_reportUePacketCount (traceParam);
955   //      }
956
957           m_channel->StartTx (txParams);
958
959           Simulator::Schedule (duration, &MmWaveSpectrumPhy::EndTx, this);
960       }
961       break;
962       default:
963           NS_LOG_FUNCTION (this<<"Programming Error. Code should not reach this point");
964       }
965       return true;
966   }
967
968   bool
969   MmWaveSpectrumPhy::StartTxDlControlFrames (Ptr<PacketBurst> pb, std::list<Ptr<MmWaveControlMessage> > ctrlMsgList, Time
          duration, uint8_t slotInd)
970   {
971       //std::cout << "MmWaveSpectrumPhy::StartTxDlControlFrames \n";
972       //std::cout << "m_state = " << m_state << "\n" ;
973       NS_LOG_LOGIC (this << " state: " << m_state);
974
975       switch (m_state)
976       {
977     case RX_DATA:
978       case RX_CTRL:
979           NS_FATAL_ERROR ("cannot TX while RX: Cannot transmit while receiving");
980           break;
981       case TX:
982           NS_FATAL_ERROR ("cannot TX while already Tx: Cannot transmit while a transmission is still on");
983           break;
984       case IDLE:
985       {
986           NS_ASSERT(m_txPsd);
```

```
987
988             m_state = TX;
989
990             Ptr<MmWaveSpectrumSignalParametersDlCtrlFrame> txParams = Create<MmWaveSpectrumSignalParametersDlCtrlFrame> ();
991             txParams->duration = duration;
992             txParams->txPhy = GetObject<SpectrumPhy> ();
993             txParams->psd = m_txPsd;
994             txParams->cellId = m_cellId;
995             txParams->pss = true;
996             txParams->ctrlMsgList = ctrlMsgList;
997             txParams->txAntenna = m_antenna;
998             //*** Acrescentei esses dois par metros aqui abaixo
999             txParams->packetBurst = pb;
1000            txParams->slotInd = slotInd;
1001            //***
1002
1003                /* This section is used for trace */
1004            Ptr<MmWaveEnbNetDevice> enbTx =
1005                        DynamicCast<MmWaveEnbNetDevice> (GetDevice ());
1006            Ptr<MmWaveUeNetDevice> ueTx =
1007                        DynamicCast<MmWaveUeNetDevice> (GetDevice ());
1008
1009            m_channel->StartTx (txParams);
1010            Simulator::Schedule (duration, &MmWaveSpectrumPhy::EndTx, this);
1011
1012
1013        }
1014        }
1015        //std::cout << "m_state 2 = " << m_state << "\n" ;
1016        return false;
1017    }
1018
1019    void
1020    MmWaveSpectrumPhy::EndTx()
1021    {
1022        NS_ASSERT (m_state == TX);
1023        //std::cout << "ENDTX() \n";
1024        m_state = IDLE;
1025    }
1026
1027    Ptr<SpectrumChannel>
1028    MmWaveSpectrumPhy::GetSpectrumChannel ()
1029    {
1030        return m_channel;
1031    }
1032
1033    void
1034    MmWaveSpectrumPhy::SetCellId (uint16_t cellId)
1035    {
1036        m_cellId = cellId;
1037    }
1038
1039
1040    void
1041    MmWaveSpectrumPhy::AddDataPowerChunkProcessor (Ptr<mmWaveChunkProcessor> p)
1042    {
1043        m_interferenceData->AddPowerChunkProcessor (p);
1044    }
1045
1046    void
1047    MmWaveSpectrumPhy::AddDataSinrChunkProcessor (Ptr<mmWaveChunkProcessor> p)
1048    {
1049        m_interferenceData->AddSinrChunkProcessor (p);
1050        //std::cout << "MmWaveSpectrumPhy::AddDataSinrChunkProcessor \n";
1051    }
1052
1053    void
1054    MmWaveSpectrumPhy::UpdateSinrPerceived (const SpectrumValue& sinr)
1055    {
1056        NS_LOG_FUNCTION (this << sinr);
1057        //std::cout << "MmWaveSpectrumPhy::UpdateSinrPerceived \n";
1058        m_sinrPerceived = sinr;
1059    }
1060
1061    void
1062    MmWaveSpectrumPhy::SetHarqPhyModule (Ptr<MmWaveHarqPhy> harq)
1063    {
1064      m_harqPhyModule = harq;
1065    }
1066
1067
```

```
1068     }
```