Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Departamento de Ingeniería Telemática

MODELADO DE LA NORMA 802.11n EN NS-3

Autor: David Bravo Almazán
Tutor: Juan M. Vozmediano Torres

Proyecto Fin de Carrera

Ingeniería de Telecomunicación

Sevilla, 29 de Junio de 2014

Tanto la memoria de este trabajo como el software desarrollado se distribuyen bajo la licencia GNU GPL v3.

La Licencia Pública General GNU (GNU GPL) es una licencia libre, sin derechos para software y otro tipo de trabajos.

Las licencias para la mayoría del software y otros trabajos prácticos están destinadas a suprimir la libertad de compartir y modificar esos trabajos. Por el contrario, la Licencia Pública General GNU persigue garantizar su libertad para compartir y modificar todas las versiones de un programa–y asegurar que permanecerá como software libre para todos sus usuarios. Nosotros, La Fundación de Software Libre, usamos la Licencia Pública General GNU para la mayoría de nuestro software; y también se aplica a cualquier trabajo realizado de la misma forma por sus autores. Usted también puede aplicarla a sus programas.

Cuando hablamos de software libre, nos referimos a libertad, no a precio. Nuestras Licencias Públicas Generales están destinadas a garantizar la libertad de distribuir copias de software libre (y cobrar por ello si quiere), a recibir el código fuente o poder conseguirlo si así lo desea, a modificar el software o usar parte del mismo en nuevos programas libres, y a saber que puede hacer estas cosas.

Más información sobre las licencias y sus términos:

http://www.gnu.org/licenses/gpl.html (Licencia original en inglés)

http://www.viti.es/gnu/licenses/gpl.html (Traducción de la licencia al castellano)

Antes de comenzar a hablar de conceptos técnicos me gustaría agradecer el apoyo que mis familiares y amigos me han dado durante mi paso por la universidad. Suena a tópico, pero tengo que reseñar que sin ellos llegar hasta aquí hubiera sido imposible para mí. La confianza que han depositado siempre en mi ha sido el motor que me ha traído hasta aquí.

Gracias de verdad

"The nice thing about standards is that there are so many to choose from.

And if you really don't like all the standards you just have to wait another year until the one arises you are looking for."

"Lo bueno de las normas es que hay muchas entre las que elegir.

Y si realmente ninguna norma le gusta solo tiene que esperar otro año hasta que aparezca la norma que busca."

– A. Tanenbaum, "Introduction to Computer Networks"

Índice

Figuras

Tablas

Resumen

En la actualidad son valoradas las herramientas que permiten simular el comportamiento de dispositivos reales. Este tipo de herramientas son típicamente de pago, aunque también existen alternativas distribuidas bajo licencia libre. La calidad del software técnico distribuido bajo licencia libre ha estado siempre ligada al ámbito académico.

Network Simulator 3 es un simulador de redes basado en eventos discretos y distribuido bajo licencia libre. El proyecto *ns-3* nació con la idea de dar la posibilidad de simular redes que se comporten bajo las nuevas normas tecnológicas. Hoy en día las nuevas normas tecnológicas aportan una cantidad tan grande de novedades tecnológicas que no pueden ser modeladas a tiempo por la comunidad de desarrolladores de *ns-3*. Es el caso de la norma 802.11n, esta norma de la IEEE define las modificaciones de sus normas predecesoras que hacen que las redes inalámbricas de área local mejoren significativamente su rendimiento.

El modelado que hace *ns-3* de la norma 802.11n cuenta con fallos que hacen imposible la creación de escenarios en los que dispositivos 802.11n se comuniquen usando las nuevas tasas de transmisión aportadas por la norma. Por otra parte, *ns-3* cuenta con carencias en el modelado de la sensibilidad de recepción de señales; y en la gestión de los casos de conexión y desconexión de un cliente.

Este proyecto solventa los fallos comentados y aporta bancos de pruebas que verifican su validez. El avance en la calidad del modelado de la norma 802.11n es tan significativo que abre la puerta a investigaciones de la norma 802.11n que se quieran realizar utilizando Network Simulator 3 como herramienta.

Por último, este documento aporta una sólida base de conocimiento con la que iniciar el desarrollo de un algoritmo de selección de tasa. El diseño de un algoritmo de estas características es la línea de continuación ofrecida. Ahora, gracias al modelado introducido en este proyecto, es posible simular de manera fiable el comportamiento del algoritmo diseñado.

1. Introducción

A finales de Julio de 2013 recibí un correo electrónico de AOIFE Solutions en el que decían buscar alumnos en prácticas con la posibilidad de realizar el proyecto fin de carrera con ellos. Mi perfil les llegó a través de Juan Manuel Vozmediano al que le estaré siempre agradecido por ello. Tras una entrevista y alguna reunión con Jose Ayub González me embarqué en las prácticas y en la realización de este proyecto fin de carrera en el marco de las redes inalámbricas.

La red inalámbrica más popular es la denominada red de área local inalámbrica, también conocida como WLAN (del inglés wireless local area network). La tecnología de este tipo de redes está definida por la IEEE y queda recogida en la norma 802.11.

1.1. Objetivo del proyecto

El objetivo del presente proyecto fin de carrera es detectar y corregir las debilidades del modelado del módulo Wi-Fi de *ns-3* que hacen que el comportamiento de la simulaciones sea erróneo bajo la norma 802.11n. Se han abordado las debilidades que aún no han sido detectadas por la comunidad de desarrolladores de *ns-3*. Las debilidades objetivo han sido corregidas bajo el prisma del estudio del comportamiento de los algoritmos de selección de tasa de datos bajo la norma 802.11n.

Tres han sido las principales debilidades a corregir: dispositivos operando con los esquemas de modulación y codificación (MCS) de la norma 802.11n, detección de las señales según el modo de transmisión con la que son emitidas y gestión de asociaciones entre APs y STAs en el caso de una de-asociación.

Además del modelado, el proyecto también aporta todos los escenarios en los que éstos han sido probados. En la construcción de estos escenarios se ha empleado parte del esfuerzo de este proyecto para poder tener un marco fiable de recogida de estadísticas, además de un correcto uso de los modelos de *ns-3*.

1.2. Planificación

El estudio y desarrollo del modelado de una tecnología requiere de unos profundos conocimientos tanto de la tecnología en cuestión como de sus diferentes variantes o normas. Varios son los ámbitos que han de ser dominados por toda persona que se embarque por primera vez en un proyecto de estas características.

La preparación comienza con el estudio de las redes de área local inalámbrica. Los conocimientos adquiridos durante la carrera deben ser complementados con los conocimientos específicos de esta tecnología en las capas del modelo OSI que se van a tratar. Este conocimiento está recogido en las diferentes normas creadas por la IEEE.

Por otra parte se debe dedicar un tiempo considerable a la familiarización con la herramienta de trabajo. En este proyecto, la herramienta es un simulador de eventos discretos de redes. La familiarización con este software engloba la instalación, la lectura de tutoriales, la preparación del entorno de trabajo y la especialización en el bloque de interés.

Una vez adquirido el conocimiento necesario, se pasa a hacer un análisis exhaustivo del comportamiento actual del modelado. Durante ese análisis se manifiestan diferentes debilidades o errores de modelado que hacen que el modelo no se comporte acorde a lo establecido por la norma. De esos errores se seleccionan los más relevantes, y una vez seleccionados, se pasa a descubrir cuál es el origen de los mismos.

Conocido el origen del problema se pasa a diseñar una corrección que tenga un pequeño impacto en el esqueleto del modelado actual, pero que solucione de manera efectiva el comportamiento erróneo. El siguiente paso por tanto será codificar dicha corrección y comprobar que el nuevo modelado se comporta acorde al diseño. En caso de que el comportamiento no sea el deseado, se replanteará el diseño de la corrección o la codificación de la misma, según sea el caso. El diseño de la corrección y la codificación de la misma deben ser verificadas de tal manera que se eliminen comportamientos transitorios. También se eliminará la dependencia con las variables aleatorias del modelado a través de la realización de baterías de simulaciones con diferentes semillas.

El coste de este proyecto viene dado por todo aquello que supondría una inversión si el proyecto se hubiera realizado fuera del ámbito académico. Es decir: 900 horas de trabajo de un ingeniero junior más el coste de la adquisición de un ordenador de gama media-alta. El gasto en software es cero puesto que se ha trabajado con software de libre distribución: *ns-3* y el sistema operativo *Fedora 20*.

2. Antecedentes

Este capítulo recoge el conocimiento necesario para una correcta comprensión de la materia que se trata en este documento. Para empezar se hace una revisión histórica de las normas que regulan el funcionamiento de las redes de área local inalámbrica. Después, se hace un análisis más exhaustivo de la norma 802.11n, por ser la norma con la que se trabaja en este proyecto.

Parte importante es también el simulador de red utilizado, *ns-3*. En la sección correspondiente de este capítulo se encuentra tanto la información general del software como los detalles de cómo está organizado. Tras la visión general del simulador, se estudia el módulo que modela el funcionamiento de los dispositivos que operan en redes inalámbricas bajo la norma 802.11.

2.1. Wi-Fi 802.11

[1]En 1999 el grupo de empresas formado por 3Com, Airones, Intersil, Lucent Technologies, Nokia y Symbol Technologies detectaron la necesidad de establecer un mecanismo de conexión inalámbrica que fuese compatible entre distintos dispositivos. De esta manera, estas empresas conformaron la Wireless Ethernet Compatibility Alliance, o WECA, actualmente llamada Wi-Fi Alliance. Tenían como objetivo crear una marca que fomentara el sencillo uso de la tecnología inalámbrica asegurando a su vez la compatibilidad entre equipos mediante la definición de los dos niveles inferiores de la arquitectura OSI.

El término Wi-Fi no responde a ninguna abreviatura, es una simple invención de una agencia que a petición de la WECA creó este término para sustituir al del nombre de la tecnología: IEEE 802.11b de Secuencia Directa.

Como concepto general, la norma IEE 802.11 define los dos niveles inferiores de la arquitectura OSI (el físico y de enlace) de las redes inalámbricas. Es decir, define cómo se utilizará el canal físico (aire) a través del cual enviamos y recibimos nuestra información, y los protocolos utilizados para ello.

2.1.1. Historia/Evolución

Legacy

La norma original IEEE 802.11 de 1997 (conocido como "legacy"), especificaba dos tasas de transmisión de 1 y 2 Mbit/s sobre infrarrojos (IR) o sobre radiofrecuencia en la banda ISM de 2,4 GHz. Aunque la transmisión por infrarrojos sigue incluida en la norma

no hay en el mercado productos que la utilicen. Permite usar codificación DSSS (Direct Sequence Spread Spectrum) o FHSS (Frequency Hopping Spread Spectrum).

802.11a

La revisión 802.11a a la norma original fue ratificada en 1999 y funciona en la banda de 5GHz utilizando 52 subportadoras OFDM (Orthogonal Frequency-Division Multiplexing). 802.11a tiene una velocidad teórica máxima de 54 Mbit/s, con velocidades reales de aproximadamente 20 Mbit/s. La velocidad de datos se reduce a 48, 36, 24, 18, 12, 9 o 6 Mbit/s en caso necesario. 802.11a tiene 12 canales no solapados, 8 para red inalámbrica y 4 para conexiones punto a punto. Los equipos 802.11a y 802.11b no pueden operar entre ellos. Los primeros productos con soporte para la norma 802.11a aparecieron en el mercado en 2001.

802.11b

La revisión 802.11b de la norma original fue ratificada en 1999 y funciona en la banda de 2.4GHz. Fue la primera revisión que tuvo una amplia aceptación en el mercado. 802.11b tiene una velocidad teórica máxima de transmisión de 11 Mbit/s, pero debido al espacio ocupado por la codificación del protocolo CSMA/CA, en la práctica la velocidad máxima de transmisión es de aproximadamente 5.9 Mbit/s para TCP y 7.1 Mbit/s para UDP.

Los dispositivos 802.11b deben mantener la compatibilidad con la norma original IEEE 802.11 (utilizando DSSS). Aunque también utiliza una técnica de ensanchado de espectro basada en DSSS, en realidad la extensión 802.11b introduce CCK (Complementary Code Keying) para llegar a velocidades de 5,5 y 11 Mbit/s (tasa de bit en capa física). La norma también admite el uso de PBCC (Packet Binary Convolutional Coding) como opcional.

802.11g

La revisión 802.11g es la evolución de la norma 802.11b y fue ratificada en Junio de 2003. Es compatible con la norma b y utiliza las mismas frecuencias (2.4GHz) aunque con una velocidad teórica máxima de transmisión de 54 Mbit/s. La velocidad real de transferencia es de aproximadamente 22 Mbit/s, similar a la de la norma 802.11a.

En redes bajo la norma g la presencia de nodos bajo la norma b reduce significativamente la velocidad de transmisión. Los equipos que trabajan bajo la norma 802.11g llegaron al mercado muy rápidamente, incluso antes de su ratificación oficial. Esto se debió en parte a que para construir equipos bajo esta nueva norma se podían adaptar los ya diseñados para la norma b. A partir de 2005, la mayoría de los productos que se comercializan siguen la revisión 802.11g con compatibilidad hacia 802.11b.

802.11n

En enero de 2004, el IEEE anunció la formación de un grupo de trabajo 802.11 para desarrollar una nueva revisión la norma 802.11. La velocidad real de transmisión podría llegar a los 300 Mbit/s (lo que significa que las velocidades teóricas de transmisión serían aún mayores), y debería ser hasta 10 veces más rápida que una red bajo las normas 802.11a y 802.11g, y unas 40 veces más rápida que una red bajo la norma 802.11b. También se espera que el alcance de operación de las redes sea mayor con esta nueva norma gracias a la tecnología MIMO Multiple Input Multiple Output, que permite utilizar varios canales a la vez para enviar y recibir datos gracias a la incorporación de varias antenas. Existen también otras propuestas alternativas que podrán ser consideradas. La norma ya está redactada, y se viene implantando desde 2008. A principios de 2007 se aprobó el segundo boceto de la norma. Ha sufrido una serie de retrasos y el último lo lleva hasta noviembre de 2009.

Protocolo 802.11	Lanzamiento	Frecuencia (GHz)	Ancho de banda (MHz)	Tasa de datos por flujo (Mbit/s)	Flujos MIMO permitidos	Modulación
legacy	Junio 1997	2.4	20	1, 2	1	DSSS, FHSSS
a	Septiembre 2009	5	20	6, 9, 12, 18, 24, 36, 48, 54	1	OFDM
b	Septiembre 2009	2.4	20	1, 2, 5.5, 11	1	DSSS
g	Junio 2003	2.4	20	6, 9, 12, 18, 24, 36, 48, 54	1	OFDM, DSSS
n	Octubre 2009	2.4/5	20	7.2, 14.4, 21.7, 28.9, 43.3, 57.8, 65, 72.2	4	OFDM
			40	15, 30, 45, 60, 90, 120, 135, 150		

Tabla 2.1: Comparativa normas 802.11

A diferencia de las otras versiones de Wi-Fi, 802.11n puede trabajar en dos bandas de frecuencias: 2,4 GHz (la que emplean 802.11b y 802.11g) y 5 GHz (la que usa 802.11a). Gracias a ello, 802.11n es compatible con dispositivos basados en todas las ediciones anteriores de Wi-Fi. Además, es útil que trabaje en la banda de 5 GHz, ya que está menos congestionada y en 802.11n permite alcanzar un mayor rendimiento.

802.11n se basa en las normas anteriores, pero hace uso de tecnología MIMO (Multiple Input, Multiple Output) para emplear más de un canal a la vez. De este modo el equipo receptor puede, por ejemplo, percibir las señales que llegan reflejadas (y por tanto con un

poco de retardo). Además, hace uso de channel bonding, una característica que incrementa el ancho de banda de cada canal a 40MHz. Y, como ya sabemos, a mayor ancho de banda, mayor capacidad de transmisión de datos. Se ampliarán los detalles técnicos de esta norma en el siguiente apartado debido a que es la norma usada para este proyecto. Se conoce que la futura norma sustituta de 802.11n será 802.11ac con tasas de transferencia superiores a 1 Gbit/s.

802.11ac

Norma de conexión Wi-Fi en desarrollo, con notables mejorías respecto a 802.11n, se calcula que sea de uso común en 2016. Se utiliza parte de las normas 802.11a y n. Puede suministrar una velocidad de transmisión de más de 1 Gbit/s en la banda de 5 GHz. Se utiliza canales extendidos de 80 o 160 MHz, el doble o el cuádruple que en n. Utiliza MIMO multi-usuario con 5 a 8 secuencias espaciales, con la modulación 256-QAM, el cuádruple que en n.

2.1.2. 802.11n

Como ya se ha comentado, dentro de las **normas más extendidas** en el mercado, esta revisión es la más avanzada tecnológicamente. Es por esto que a continuación se describirán los mecanismos tecnológicos que le diferencian de sus normas predecesoras.

En la última publicación de la norma 802.11n [2] se recogen los detalles tecnológicos de la misma, detallando cuales son necesarios y optativos según la capa del modelo OSI. De todos estos avances que incluye la norma 802.11n nos detendremos en los que mayor peso han tenido en este proyecto fin de carrera.

Intervalo de guarda (Guard Interval, GI)

El intervalo de guarda es un período de tiempo entre símbolos OFDM que se utiliza para preparar al sistema ante la llegada tardía de símbolos a través de caminos largos. En escenarios multitrayecto los símbolos viajan por diferentes caminos. El hecho de que las señales recorran distancias diferentes hace que dos simbolos emitidos en distintos instantes de tiempo puedan interferir entre sí al llegar al receptor. Este efecto se conoce como interferencia entre símbolos (ISI).

Aunque no es una regla en el sentido legal, una buena regla usada por los diseñadores OFDM es que el intervalo de guarda debe ser igual a 4 veces el máximo retraso de expansión multitrayecto (multipath delay spread), siendo este la diferencia de tiempo entre varios caminos de una misma señal. Cuando 802.11a fue diseñado, los diseñadores usaron un valor conservador de 200 ns para el retraso de expansión, y siguiendo la regla anterior fijaron el GI a 800 ns. Después la experiencia ha demostrado que en la mayoría de entornos interiores casi nunca se llega a los 100 ns de retraso de expansión y a menudo suele estar comprendido entre los 50 y 75 ns. Por eso para conseguir un rendimiento mayor del enlace radio, 802.11n incluye una opción para intervalo de guarda corto, el cual se reduce a 400 ns. El éxito del GI corto depende de la expansión multitrayecto. En general, la interferencia multitrayecto es peor cuando hay importantes reflexiones debido a metales.

El resto de parámetros OFDM se quedan igual. La velocidad global se incrementa porque la parte del tiempo de símbolo dedicada a la transmisión de datos es todavía de 3,2 μs, pero cada símbolo es más corto. La longitud total de cada símbolo se reduce de 4 μs con el GI largo a 3,6 (3,2 μs + 0,4 μs) con el GI corto, reduciendo de esta forma la sobrecarga de OFDM en un 10 %.

Esquema de modulación y codificación

En 802.11n las velocidades de datos están ahora definidas por un esquema de modulación y codificación (Modulation and Coding Scheme, MCS). En 802.11a/g, las velocidades se definían mediante la modulación y estaban comprendidas entre los 6 y 54 Mbit/s. Sin embargo, en 802.11n las velocidades de datos dependen de varios factores: la modulación, el número de flujos espaciales, la anchura del canal, el intervalo de guarda y la tasa de codificación. Cada MCS puede suponer una variación del número de flujos espaciales, la modulación y la tasa de codificación. Dentro de cada MCS hay distintas velocidades en función del ancho de banda del canal y el intervalo de guarda. 802.11n define 77 MCS diferentes. Los 32 primeros con modulación igual y el resto con modulación distinta. Actualmente, la mayoría de los productos solamente soportan modulación igual. La modulación distinta es útil cuando un flujo espacial está más dañado que los demás. Las modulaciones usadas son BPSK, QPSK, 16-QAM y 64-QAM. En la tablas 1.2 se muestran los primeros 8 MCS con sus diferentes parámetros y las velocidades de datos permitidas para cada MCS, para modulación igual.

La operación con un flujo (MCS 0-7) es obligatoria para todas las estaciones. La operación con 2 flujos (MCS 8-15) es obligatoria para puntos de acceso. La operación con 3 y 4 flujos (MCS 16-31) es opcional para todos los dispositivos. Los AP con 3 flujos dominan sobre los de 4. Al MCS 32 se le conoce como High Throughput Duplicate Mode. Este usa un canal de 40 MHz y un único flujo espacial a una tasa de 6 Mbit/s. Es un modo muy conservador para conseguir una alta fiabilidad. Normalmente se utiliza en redes de gran escala. Los MCS del 33 al 76 se usan para modulación distinta.

Índice MCS	Flujos espaciales	Tipo de modulac.	Tasa de codific.	Tasa de datos (Mbit/s)			
				Canal de 20 MHz		Canal de 40 MHz	
				800ns GI	400ns GI	800ns GI	400ns GI
0	1	BPSK	1/2	6.50	7.20	13.50	15.00
1	1	QPSK	1/2	13.00	14.40	27.00	30.00
2	1	QPSK	3/4	19.50	21.70	40.50	45.00
3	1	16-QAM	1/2	26.00	28.90	54.00	60.00
4	1	16-QAM	3/4	39.00	43.30	81.00	90.00
5	1	64-QAM	2/3	52.00	57.80	108.00	120.00
6	1	64-QAM	3/4	58.50	65.00	121.50	135.00
7	1	64-QAM	5/6	65.00	72.20	135.00	150.00

Tabla 2.2: Índices de esquemas de modulación y codificación

Espacio intertrama reducido (RIFS - Reduced Interframe Space)

Las funciones de acceso al canal básicas de 802.11 introducen espacios entre transmisiones como un método para establecer el acceso al medio. La idea principal del método de mediación de espacio entre tramas, es que las transmisiones de alta prioridad, como confirmaciones, tengan un período de espera menor. En las normas previas a la 802.11n el espacio entre tramas utilizado era el corto (short interframe space, SIFS).Así, una confirmación puede ser transmitida después de esperar sólo un SIFS. De la misma manera, una trama CTS transmitida inmediatamente siguiendo a una trama RTS necesita esperar solo un SIFS antes de acceder al medio.

Banda	SIFS	RIFS
2.4 GHz	10 μs	2 μs
5 GHz	16 μs	2 μs

Tabla 2.3: Espacio intertrama

802.11n define un nuevo espacio entre tramas, denominado reducido (Reduced Interframe Space, RIFS) y su función es equivalente a la del SIFS. Sin embargo, es más corto, como se muestra en la tabla 1.3. No define un nuevo nivel de prioridad. Su único propósito es ser usado en lugar de SIFS para aumenta la eficiencia. No esta disponible en dispositivos 802.11a/b/g por lo que no puede ser usado por un dispositivo 802.11a/b/g.

La mayoría de los dispositivos no transmiten usando RIFS porque la eficiencia que se consigue es relativamente pequeña. Sin embargo, todos los dispositivos con la certificación Wi-Fi n deben tener la habilidad de recibir tramas transmitidas después de un RIFS.

Modos PLCP (PHY Layer Convergence Protocol)

Como en las anteriores capas físicas de 802.11, la especificación 802.11n define una trama de capa física usando PLCP. En 802.11n PLCP soporta 3 modos diferentes:

Modo No-HT

Todos los dispositivos tienen que soportar este modo, el cual permite la compatibilidad con dispositivos 802.11a/b/g. Ninguna funcionalidad de 802.11n está disponible en este modo. El formato de trama para este modo es exactamente igual que el de 802.11a o 802.11g. Alguna documentación se refiere a él como modo legado (legacy mode), porque opera acorde con las mismas reglas que las especificaciones anteriores.

Modo HT mixto (HT-MM)

Este modo también debe ser soportado por todos los dispositivos 802.11n. Este sólo tiene compatibilidad con 802.11a/g, pero la parte de High Throuhput no puede ser decodificada por un dispositivo 802.11a/g.

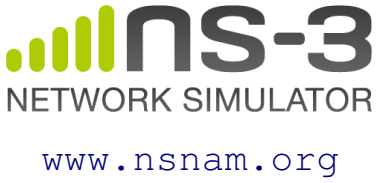
Modo greenfield (HT-GF)

Este último modo no es compatible con ninguna otra versión de 802.11, por ello es conveniente utilizarlo solamente en áreas donde todos los dispositivos son compatibles

con 802.11n. El modo HT greenfield consigue un pequeño aumento en el rendimiento con respecto al modo mixto, ya que la cabecera PLPC es 8 μs más corta. No es obligatorio y no es muy frecuente su implementación en dispositivos 802.11n.

2.2. Network Simulator 3

Más conocido como *ns-3*, Network Simulator 3 es un simulador de redes basado en eventos discretos. *ns-3* es software libre, acogido a la versión 2 de la GNU General Public License igual que su antecesor, el Network Simulator 2 (ns-2). ns-2 fue desarrollado en C++ y provee una interfaz de simulación a través de OTcl, una variante orientada a objetos de Tcl. En ns-2 el usuario describe una topología de red por medio de scripts OTcl, y luego el programa principal simula dicha topología utilizando los parámetros definidos.



En el año 2006, ns-3 nace como variante de ns-2, se decidió implementar una nueva versión desde cero utilizando C++ como lenguaje único para los modelos y la interfaz de simulación. La base del desarrollo fue el paquete Yans (Yet Another Network Simulator[10]). Su uso está destinado a las comunidades educativas e investigadoras, el proyecto se esfuerza por mantener un entorno abierto y accesible a los investigadores para compartir y/o aportar su software. *ns-3* no es compatible con su versión anterior (ns-2), es un nuevo simulador. Ambos simuladores están escritos en C++ pero *ns-3* no admite las APIs de ns-2. Algunos modelos de ns-2 han sido portados a *ns-3*. Mientras *ns-3* esté en construcción, ns-2 será mantenido buscando a su vez mecanismos de integración entre ambos.
ns-3 proporciona modelos para ver como trabajan y rinden las redes de paquetes de datos. También cuenta con un motor de simulación para realizar diferentes simulaciones de experimentos. Una de las razones para utilizar *ns-3* es la posibilidad de realizar estudios de experimentos imposibles en un sistema real o de experimentos difíciles de ejecutar en sistemas reales. Es útil también para hacer estudios del comportamiento de un sistema bajo un entorno controlado y reproducible. Por supuesto útil también para aprender como funcionan las redes, es una herramienta educativa muy recomendable.

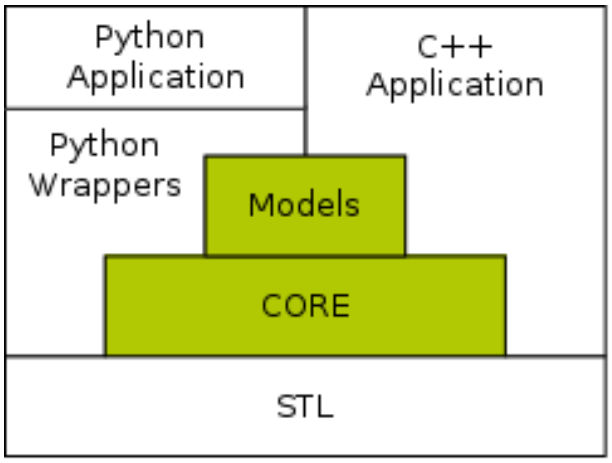
Existen muchas herramientas para realizar estudios de simulación de redes, a continuación se presentan algunas de las características que diferencia a *ns-3* de otras herramientas:

- *ns-3* está diseñado como un conjunto de bibliotecas que pueden ser usadas conjuntamente entre ellas, incluso con bibliotecas software externas. Algunas plataformas de simulación proporcionan al usuario una única interfaz gráfica de usuario integrada en la que se ejecutan todas las tareas, *ns-3* es más modular en ese sentido. Muchas son las herramientas externas de análisis de datos, visualización y animación que pueden ser usadas con *ns-3*. En cualquier caso, los usuarios deben estar preparados para trabajar en un terminal y con las herramientas para desarrollo de software en C++ y/o Python.

- *ns-3* se utiliza principalmente en sistemas Linux, aunque existe soporte para otras plataformas como FreeBSD y Cygwin (para Windows), el soporte nativo para Windows Visual Studio se encuentra en proceso de desarrollo.
- *ns-3* no es un producto software oficialmente apoyado por ninguna empresa. El soporte de *ns-3* se realiza a través de la lista de correo de usuarios de *ns-3* y del grupo de usuarios creado en Google Groups.

En *ns-3* el simulador está escrito enteramente en C++ con enlaces Python opcionales. Por tanto, los scripts de simulación pueden ser escritos en C++ o Python. Existen nuevos visualizadores y animadores disponibles y bajo desarrollo. También se pueden utilizar utilidades de análisis de trazas puesto que *ns-3* es capaz de generar archivos pcap con las trazas de los paquetes. ns-2 cuenta con un conjunto mas variado de módulos aportados debido a su larga historia. Sin embargo, *ns-3* cuenta con modelos más detallados en muchas de las áreas más populares en investigación actualmente (incluye módulos detallados para LTE y Wi-Fi). Otra potente utilidad es que toda la pila de red de Linux puede ser encapsulada en un nodo de *ns-3*, se consigue a través de un framework llamado Direct Code Execution (DCE). *ns-3* incluye también un planificador en tiempo real que facilita las simulaciones en bucle para interactuar con sistemas reales.

ns-3 es una biblioteca de C++ que proporciona un conjunto de modelos de simulación de red implementados como objetos C++ y envueltos con Python. Normalmente los usuarios usan estas bibliotecas escribiendo un aplicación en C++ o Python que crea ejemplares de un conjunto de modelos de simulación para construir el escenario de simulación deseado, ejecutar el hilo principal y salir de él cuando se complete la simulación.



STL - Standard Template Library
(Biblioteca de plantillas normalizadas)

Figura 2.1: Arquitectura *ns-3*

La biblioteca *ns-3* está envuelta en Python gracias a la biblioteca pybindgen que delega el análisis de las cabeceras *ns-3* en C++ a gccxml y pygccxml para generar automáticamente el correspondiente enlazado C++. Estos archivos C++ generados automáticamente son compilados dentro del modulo python de *ns-3* para permitir a los usuarios interactuar con el core y con los modelos de *ns-3* en C++ a través de scripts en python.

Los eventos de simulación en *ns-3* son simples llamadas a funciones que están programadas para ejecutarse en un tiempo de simulación determinado. Cualquier función se puede convertir en un evento programado haciendo uso de las funciones callback o retrollamada. Estas funciones se usan mucho en el simulador para reducir las dependencias en tiempo de compilación entre los objetos de simulación.

ns-3 cuenta con una potente API de bajo nivel que permite a los usuarios expertos hacer distintas configuraciones de los objetos con una mayor flexibilidad. En lo alto de todo esto existe un conjunto de capas de "helpers" que ofrecen funciones sencillas con un comportamiento razonable por defecto. Los usuarios pueden mezclar el uso de las APIs sencillas de los helpers con las APIs más complejas de la capas inferiores.

Los nodos de *ns-3* siguen el modelo de la arquitectura de red de Linux, las interfaces y objetos clave (sockets, dispositivos de red...) también están modelados a semejanza de un equipo Linux. Esto facilita la reultilización de código y mejora el realismo de los modelos, también hace que el control de flujo del simulador sea más sencillo al poderlo comparar con los sistemas reales.

El código fuente de *ns-3* está organizado en su mayoría en el directorio *src* y puede ser descrito por el esquema de organización del software de *ns-3* de la figura 2.2. El trabajo se realiza de capas inferiores a superiores, en general los módulos solo tienen dependencias en módulos de capas inferiores.

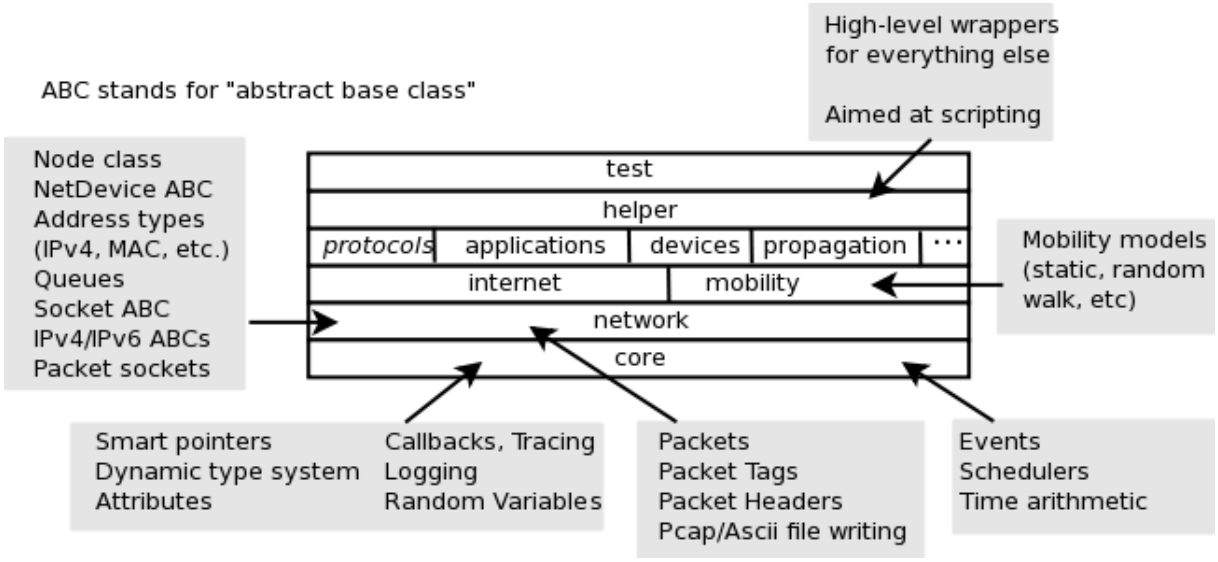


Figura 2.2: Organización del software en *ns-3*

En el core del simulador se encuentran aquellos componentes que son comunes para todos los modelos de protocolo, hardware y entorno. El core del simulador está implementado en *src/core*, otros objetos fundamentales en un simulador como los paquetes están implementados en *src/network*. Estos dos módulos están destinados a conformar el núcleo de un simulador genérico que puede ser usado por diferentes tipos de redes y no solo redes basadas en internet. Además del núcleo de *ns-3*, existen dos módulos que complementan el núcleo con APIs basadas en C++. Los programas de *ns-3* pueden acceder directamente a todas las APIs o pueden hacer uso de las APIs llamadas "helpers" que proporcionan un encapsulado de llamadas a APIs de más bajo nivel. El hecho de que los programas

de *ns-3* puedan escribirse usando dos APIs (por separado o de manera conjunta) es un aspecto fundamental del simulador.

Los nodos de *ns-3* pueden contener una colección de objetos *NetDevie*, al igual que un equipo real tiene separadas las tarjetas de las interfaces Ethernet, Wi-Fi, Bluetooth, etc. Añadiendo objetos WifiNetDevice a los nodos, uno puede crear modelos infraestructura basados en 802.11 y redes ad-hoc. Los objetos WifiNetDevice modelan un controlador de una interfaz de red inalámbrica basado en la norma 802.11 de la IEEE[2]. *ns-3* proporciona modelos para las siguientes funcionalidades de 802.11:

- Distributed Coordination Function (DCF) básica de 802.11 para los modos infraestructura y ad-hoc.
- Las capas físicas de las versiones de la norma 802.11a, 802.11b y 802.11g.
- Enhanced Distributed Channel Access (EDCA) basado en Quality of Service (QoS) y extensiones de encolado de 802.11e.
- La capacidad de utilizar diferentes modelos de pérdidas de propagación y retraso por propagación.
- Diversos algoritmos de control de la tasa entre los que se incluyen Aarf[11], Arf[12], Cara[13], Onoe, Rraa[14], ConstantRate y Minstrel.
- Redes mesh, norma 802.11s.

El conjunto de modelos 802.11 proporcionados en *ns-3* pretende una implementación fiel de la capa MAC especificada en 802.11 y pretende también proporcionar un modelo de la capa PHY de la norma 802.11a. En *ns-3*, los nodos pueden tener diferentes WifiNetDevice en canales separados, pudiendo coexistir con otro tipo de dispositivos, eliminando así una limitación de la arquitectura de ns-2. El código fuente del WifiNetDevice se encuentra en el directorio *src/wifi*.

La implementación es modular y ofrece cuatro niveles de modelos:

- Modelos de la capa PHY.
- Los llamados modelos MacLow que implementan DCF y EDCAF.
- Los llamados modelos "MAC High" que implementan la generación de radiobalizas, sondeo y maquinas de estados de asociaciones del nivel MAC.
- Un conjunto de algoritmos de control de tasa usados por los modelos MacLow.

A continuación veremos alguna de estas capas más en detalle.

Modelos "MAC High"

Hay tres grandes modelos de MAC que proveen los tres elementos de una posible topología Wi-Fi; el punto de acceso o AP (ApWifiMac), la estación no-AP o STA (StaWifiMac) y un STA en un conjunto básico independiente de servicios (IBSS, comúnmente conocido como red ad-hoc), (AdhocWifiMac).
El modelo más sencillo es el *AdhocWifiMac*, esta clase modela la capa MAC de un dispositivo que no genera radiobalizas, ni realiza sondeos o asociaciones. La clase *StaWifiMac*

implementa sondeo activo y una máquina de estados de asociación que se encarga de hacer una reasociación cada vez que muchas de las radiobalizas se pierden. Por último, *ApWifiMac* modela un punto de acceso que periódicamente genera radiobalizas y que acepta todos los intentos de asociación por parte de los STAs.
Estos tres modelos "MAC High" comparten una misma clase padre, llamada *RegularWifiMac*. A través de esta clase se puede configurar entre otras cosas el atributo *QosSupported* que permite la configuración al estilo QoS 802.11e/WMM y el atributo *HtSupported* que permite el soporte de 802.11n High Throughput.

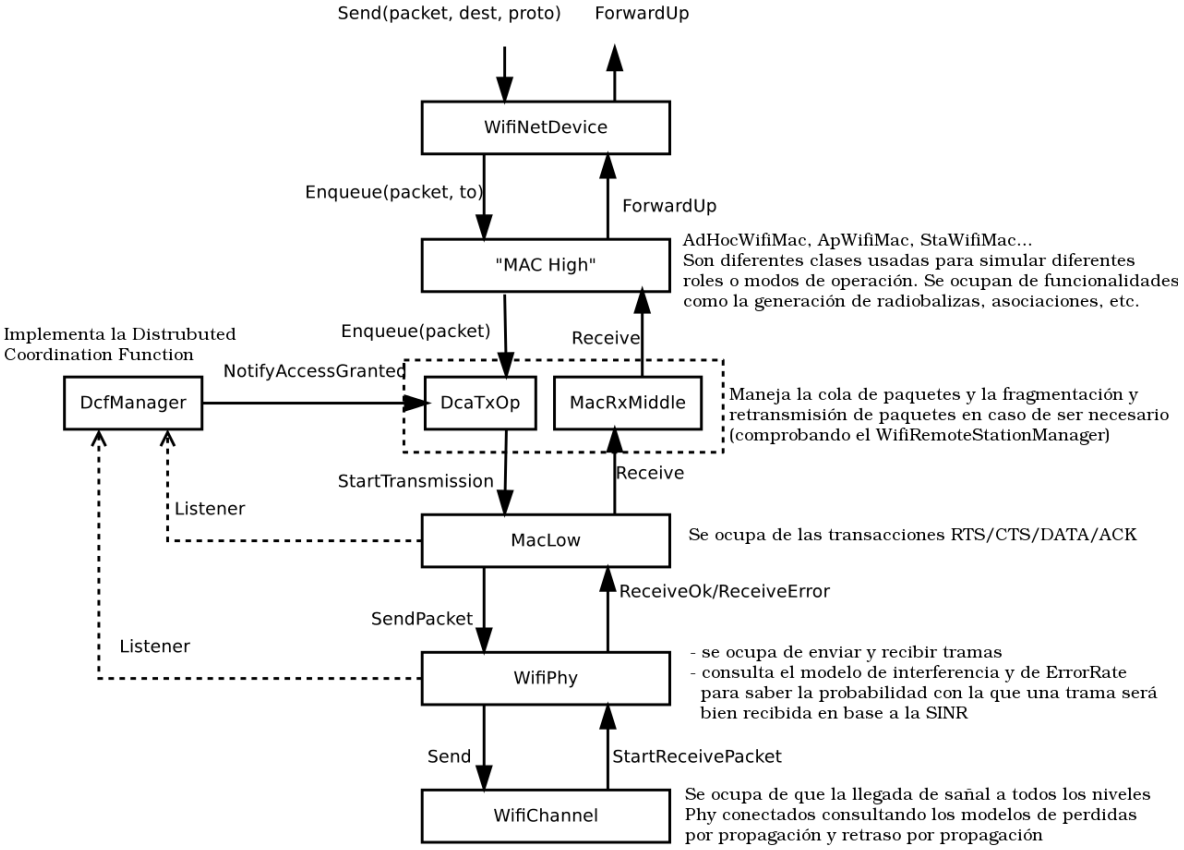


Figura 2.3: Arquitectura del módulo Wi-Fi

Capa MAC low

La capa MacLow está dividida en tres componentes:

- *MacLow* que se encarga de las transacciones de mensajes RTS/CTS/DATA/ACK.
- *DcfManager* y *DcfState* que implementan las funciones DCF y EDCAF.
- *DcaTxop* y *EdcaTxopN* que manejan la cola, la fragmentación y retransmisión de paquetes en caso de ser necesario. El objeto *DcaTxop* es utilizado en capas MAC altas que no tienen el QoS activo, también puede ser utilizado para transmitir ciertos tipos de tramas. Por ejemplo, la norma dice que las tramas de gestión deben ser enviadas accediendo al medio usando DCF. *EdcaTxopN* es utilizado en capas MAC

altas que tienen el QoS activo, además de realizar operaciones de QoS como la agregación MSDU de 802.11n.

La modularidad proporcionada por la implementación hace que la configuración de bajo nivel del WifiNetDevice sea potente aunque compleja. Por esta razón, *ns-3* ofrece algunas clases de ayuda para realizar operaciones comunes de manera sencilla y por otra parte aprovechar el sistema de atributos de *ns-3* para permitir a los usuarios controlar la parametrización de los modelos subyacentes.

Conocidas las bases del modelado en *ns-3*, en el siguiente capítulo se hace un análisis de sus debilidades. Como ya se ha comentado, se ha puesto el foco en el modelado del módulo Wi-Fi y mas en concreto en la implementación de la norma 802.11n.

3. Análisis de debilidades

Con la ayuda del grupo de usuarios alojado en Google, la lista de distribución de correo de desarrolladores de *ns-3* y el sistema de seguimiento de errores se realiza un estudio en profundidad de las debilidades reconocidas por la comunidad de usuarios y desarrolladores de *ns-3*. Son numerosas las debilidades, pero las que afectan de manera directa al propósito de este proyecto están enumeradas a continuación:

- La utilidad Direct Code Execution está solo disponible para dispositivos cableados, no existe implementación para dispositivos inalámbricos.
- No existe una implementación sólida de la tecnología MIMO, la propagación multitrayecto es el fenómeno físico del que se vale esta tecnología para sacar provecho de la recepción de diferentes versiones de una misma señal. *ns-3* carece de un modelo de propagación multitrayecto, lo que imposibilita que en una antena receptora lleguen varias versiones de la misma señal.
- No hay ningún modelo que compute las interferencias en el espectro en el que operan las redes inalámbricas. No se contempla que dos puntos de acceso cercanos operen en canales adyacentes, lo que derivaría en interferencia mutua, peor relación señal-ruido y por tanto un peor rendimiento.
- Los algoritmos existentes de adaptación de la tasa han sido diseñados para las normas previas a 802.11n. Estos algoritmos han sido probados para 802.11a/b/g pero no para 802.11, donde algunos de ellos producen incluso fallos de memoria.

Actualmente existe un plan de trabajo dentro de la comunidad de desarrolladores de *ns-3* para corregir estas debilidades. Por otra parte se han detectado deficiencias no registradas, se trata de deficiencias en el modelado de la norma 802.11n así como en el funcionamiento básico de una conexión entre un cliente y un punto de acceso.

Debido a que las debilidades anteriormente enumeradas son conocidas, se decide corregir las debilidades no registradas y cuyo origen es aún desconocido por los grupos de trabajo de *ns-3*. En las secciones siguientes de este capítulo se encuentra el análisis de estas debilidades.

3.1. Dispositivos que operan en modos 802.11n

La norma 802.11n ha sido la última en ser modelada en *ns-3* y su funcionamiento aún no ha sido probado en profundidad. El principal problema encontrado es la imposibilidad de configurar un dispositivo para que trabaje con los modos de operación correspondientes a la norma 802.11n. Además el número de bytes transmitidos correctamente es muy bajo en el caso de que un dispositivo configurado para la norma 802.11n quiera transmitir utilizando modos compatibles con normas anteriores.

ns-3 permite instalar en cada nodo una o varias interfaces de red, esta debilidad se manifiesta en un escenario compuesto por dos nodos; uno de ellos con la interfaz de red correspondiente a un punto de acceso y el otro con la interfaz correspondiente a una estación o STA. Los detalles específicos para la construcción del escenario bajo la norma 802.11n se pueden consultar en el anexo A.1.

En las ejecuciones del escenario descrito se observa como el número de bytes transmitidos es de un orden de magnitud no esperado. En simulaciones de 5 segundos el número de bytes recibidos satisfactoriamente en la capa MAC del STA es de un 1 kB. Esto pone de manifiesto que algo no funciona correctamente.

Un exhaustivo análisis del problema descubre que los modos de transmisión DSSS (Direct-Sequence Spread Spectrum) no son compatibles con la configuración HT de la capa MAC. Para confirmar ésto se ha forzado el modo de transmisión a un valor constante para toda la simulación. A través de la clase `ConstantRateWifiManager` se puede mantener constante el modo de transmisión al que se transmiten los datos:

```
WifiHelper wifi = WifiHelper::Default ();
wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
  "DataMode",StringValue ("OfdmRate6_5MbpsBW20MHz"));
```

El modo de transmisión elegido es el correspondiente al índice MCS 0 para intervalo de guarda largo y ancho de canal de 20 MHz de la tabla 2.2.

Tras la simulación con la nueva configuración los valores correspondientes a los bytes recibidos superan los 6000 kB. Se hacen varias ejecuciones con los distintos modos de transmisión que soporta *ns-3* y que no son DSSS. Los resultados confirman que los modos de transmisión DSSS no son soportados bajo la configuración HT de la capa MAC.

La raíz del problema es el error del simulador al trabajar con los modos de transmisión que no son OFDM con la capa MAC configurada para HT.

3.2. Sensibilidad de recepción de señal para modulaciones OFDM de 802.11n

El modelado de *ns-3* cuenta con un umbral mínimo para la detección de señales en el receptor. Se trata del atributo privado `m_edThresholdW` de la clase `YansWifiPhy`. Este atributo solo se emplea en un sitio, dentro del método `StartReceivePacket`, para tirar los paquetes cuya potencia de señal en el receptor sea menor que el valor del umbral guardado en `m_edThresholdW`. Por defecto este umbral vale -96 dBm.

Se detecta que éste es el único umbral existente para discernir si la potencia de la señal recibida es suficiente para recibir correctamente el paquete o no. En este caso no se trata de un modelado erróneo, sino de una carencia del mismo. El modelado si contempla cual debe ser la potencia mínima de una señal para ser detectada en el receptor, pero no implementa cual debe ser el valor mínimo de la potencia de dicha señal para que sea demodulada correctamente.

3.3. Gestión de asociaciones entre APs y STAs

El modelado que hace *ns-3* de la pérdida de conexión entre un punto de acceso y una estación está incompleto. El modelado no cuenta con ningún mecanismo que contemple la pérdida de conexión por el excesivo aumento de la distancia que separa a un cliente del punto de acceso al que está conectado. Esta deficiencia del modelado se detecta en el contexto del trabajo con la distancia límite a la que puede estar el cliente para poder demodular las señales correspondientes a los índices MCS.

Con el objetivo de sobrepasar cada una de estas distancias límites, se configura en el cliente un patrón de movimiento por el cual, partiendo de la posición del punto de acceso al que está conectado, se desplaza en línea recta a una velocidad constante (1 m/s). A medida que el cliente se va alejando del punto de acceso, los índices MCS utilizados van siendo menores hasta alcanzar la distancia límite por la que ni siquiera el índice MCS 0 puede ser recibido correctamente. Éste es el momento en el que el cliente pierde conexión con el punto de acceso, cuando el cliente deja de recibir los paquetes de gestión correspondientes a las radiobalizas.

Es en este caso en el que se manifiesta el error, se observa como el punto de acceso a pesar de que el cliente ha perdido conexión, continua enviándole paquetes correspondientes al tráfico configurado en el escenario. Los paquetes enviados al cliente por el punto de acceso tras la pérdida de conexión son recopilados en las estadísticas como paquetes tirados por señal insuficiente como para ser recibidos.

El error se considera de importancia porque afecta directamente al estudio de los algoritmos de adaptación de la tasa. Dos son los principales transtornos que produce en los escenarios de estudio de estos algoritmos:

- El punto de acceso tiene una carga de trabajo inútil que puede provocar retrasos a la hora de servir paquetes a otros clientes conectados al mismo.
- Quedan falseadas las estadísticas de paquetes tirados. En el estudio de un algoritmo de selección de tasa lo que interesa es conocer los paquetes tirados durante los cambios de tasa de transmisión y no ver ese apartado engordado por el hecho de que un cliente haya perdido conexión con el punto de acceso.

4. Modelado del módulo Wi-Fi

Para cumplir con los objetivos descritos, se ha de actuar sobre el módulo Wi-Fi de *ns-3* para que el modelado de los dispositivos 802.11n sea el correcto. En este capítulo se desgrana como se ha llevado a cabo este modelado a través de los apartados "Análisis de la solución adoptada", "Modelado de la solución" y "Simulaciones de verificación". La descripción y verificación del modelado aportado se divide en tres secciones: en la primera se demuestra como se ha conseguido que los dispositivos 802.11n se comuniquen utilizando modos de transmisión específicos de la norma; en la segunda sección se explica la introducción de un umbral de sensibilidad de recepción de señal para modulaciones OFDM de 802.11n; y en la última sección de este capítulo se puede ver como se ha modelado la gestión de asociaciones entre APs y STAs para unos casos concretos.

4.1. Dispositivos que operan en modos 802.11n

El modelado actual que hace *ns-3* de dispositivos configurados para trabajar bajo la norma 802.11n no permite que dos dispositivos configurados para 802.11n se comuniquen con modos de operación específicos de dicha norma. El objetivo en este apartado es corregir este modelado con el fin de establecer un intercambio de paquetes entre dos nodos. Estos dos nodos deberán trabajar con cualquiera de los modos de operación de la norma 802.11n recogidos en la tabla 2.2.

Como se ha comentado anteriormente, no se consigue comunicación entre dispositivos 802.11n, la raíz del problema es el error del simulador al trabajar con los modos de transmisión que no son OFDM con la capa MAC configurada para HT. Puesto que el objetivo que se busca es conseguir operar con los índices MCS y con las novedades tecnológicas que ofrece la norma 802.11n, se descarta abordar la corrección de este error. El enfoque de la solución está puesto en conseguir comunicación utilizando modos de transmisión específicos de la norma 802.11n y evitar que los dispositivos operen bajo modos de transmisión no OFDM.

4.1.1. Análisis de la solución adoptada

La solución que se ha escogido es limitar el uso de los modos de transmisión no OFDM a los dispositivos configurados para trabajar bajo la norma 802.11n. La elección de esta solución está motivada por diferentes artículos que explican como los modos de transmisión no OFDM, obligatorios por compatibilidad con las primeras normas 802.11, hacen que el rendimiento de las redes inalámbricas caiga por el ineficiente uso del espectro radioeléctrico. Como representante de los artículos que hablan de este hecho, se cita una

presentación de Cisco [15] en la que se enumeran los motivos por los que explican a la IEEE las desventajas actuales de usar modos de transmisión que no son OFDM.

A modo resumen se enumeran los argumentos más extendidos para evitar el uso de modos de transmisión no OFDM:

- Las tasas de transmisión DSSS de 1 y 2 Mbit/s son mucho más lentas que las tasas de transmisión más bajas de los modos OFDM. Esto deriva en un mayor tiempo de ocupación del canal lo que hace que el porcentaje de éxito de acceso al canal por parte de otros dispositivos sea menor.
- Estos modos de transmisión introducen cabeceras de mayor tamaño además de tiempos de preámbulo y ranura más elevados. Esto provoca un menor rendimiento de la red debido a que son menores los datos efectivos que viajan en cada trama.
- Para que puedan ser usados simultáneamente modos de transmisión OFDM con modos no OFDM, se necesitan mecanismos adicionales de protección de colisiones. Estos mecanismos contribuyen también a que la cantidad de datos efectivos enviados por unidad de tiempo sea menor.
- El número de dispositivos en el mundo que solo soportan modos de transmisión no OFDM es despreciable a día de hoy. Esto hace que el desactivar estos modos de transmisión no haga imposible la conexión para la gran mayoría de dispositivos.

Estos argumentos han llegado a los grupos de trabajo de la IEEE, que se están planteando el pasar los modos de transmisión no OFDM de obligatorios a opcionales para las normas más recientes.

4.1.2. Modelado de la solución

La corrección del modelado está basada en el conocimiento que se ha adquirido de la gestión de los modos de transmisión en *ns-3*, este conocimiento queda recogido en el anexo A.2.

Para respetar el desarrollo del proyecto *ns-3* se han condicionado las modificaciones del modelado a la activación del modo Greenfield de la norma 802.11n. Este modo no es compatible con ninguna otra versión de 802.11 y su uso está restringido a redes donde todos los dispositivos son compatibles con la norma 802.11n. Modela perfectamente el contexto deseado, una red donde no existen dispositivos con soporte único de modos de transmisión no OFMD. Para activar este modo en la capa PHY, lo único que hay que hacer en el escenario es activar el parámetro correspondiente de la clase `YansWifiPhyHelper`:

```
YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ();
wifiPhy.Set ("GreenfieldEnabled", BooleanValue(true));
```

La activación de este modo no afecta de ningún manera al modelado, dado que únicamente está creada la bandera. Esta bandera no se utiliza para modelar las características del modo Greenfield, puesto que ésta es una característica sin desarrollar en *ns-3*. A continuación se explican los cambios realizados en el modelado de *ns-3* para llevar a cabo esta corrección.

Se modifica el valor que se le da al modo de transmisión por defecto. Se ha configurado el modo por defecto para que tome el valor del primer índice MCS del conjunto de modos de transmisión devuelto por el método `GetMembershipSelectorModes` de la clase `WifiRemoteStationManager`.

En esta misma clase se encuentra el método `GetSupported`, éste es el método que utilizan las clases que modelan los algoritmos de selección de la tasa de transmisión en *ns-3* para saber que modo de transmisión le corresponde al índice que se le pasa como parámetro. Este índice lo utilizan estos algoritmos para recorrer los modos de transmisión disponibles, los modos de transmisión están siempre ordenados de menor a mayor tasa de transmisión. Esta lista de modos de transmisión debe ser cambiante según estén configurados los parámetros de la capa PHY. Los parámetros configurables para la capa PHY en *ns-3* son el intervalo de guarda y el ancho del canal, estos parámetros se configuran en el escenario a través de las siguientes sentencias:

```
wifiPhy.Set ("ShortGuardEnabled", BooleanValue(false));
wifiPhy.Set ("ChannelBonding", BooleanValue(false));
```

Si `ShortGuardEnabled` está a `false`, se configura el intervalo de guarda como largo (800 ns) mientras que si está a `true` el intervalo de guarda será corto (400 ns). El ancho de banda del canal es de 20 MHz si el parámetro `ChannelBonding` se configura a `false` y de 40 MHz si se configura a `true`. Estas dos variables junto con el índice MCS elegido por el algoritmo de selección de tasa, determinan la tasa a la que se transmiten los datos. Recordar que estas tasas pueden ser consultadas en todo momento en la tabla 2.2.

`GetSupported` no tiene en cuenta estos parámetros a la hora de devolver el modo de transmisión al que se va a transmitir. Para corregir el modelado lo que se ha hecho es utilizar un método ya existente pero infrautilizado, se trata del método `McsToWifiMode` de la clase `YansWifiPhy`. Este método recibe como parámetro el índice MCS y devuelve el modo de transmisión correspondiente teniendo en cuenta la configuración de los parámetros `ShortGuardEnabled` y `ChannelBonding`.

Estas correcciones realizadas en el archivo `wifi-remote-station-manager.cc` quedan reflejadas en el anexo C, parche C.1.

Otra parte fundamental del modelado a corregir es la negociación previa al establecimiento de una asociación entre un punto de acceso y una estación. Los puntos de acceso difunden periódicamente radiobalizas, que son tramas de gestión en las que viaja información relativa a los parámetros y capacidades del punto de acceso. Las estaciones por su parte, difunden una petición de sondeo para descubrir las redes 802.11 cercanas. Este es el punto de partida del proceso de asociación de una estación a un punto de acceso, que resumimos a continuación (se han omitido los detalles del proceso de autenticación por no ser utilizado en el modelado que nos ocupa):

1. En la petición de sondeo viajan las tasas de datos y las capacidades que la estación soporta. Esta petición es enviada en difusión y por tanto todos los puntos de acceso que la reciban responderán.
2. Cuando un punto de acceso recibe una petición de sondeo, comprueba que al menos una de las tasas de datos de la estación sea soportada por dicho punto de acceso. Si ambos tienen tasas de datos compatibles, el punto de acceso mandará una respuesta

de sondeo donde viaja el SSID (nombre de la red inalámbrica), las tasas de datos que soporta, los tipos de encriptación (si es necesario) y otras capacidades del punto de acceso.

3. De todas las respuestas de sondeo recibidas la estación elegirá las redes compatibles. Una vez que las redes compatibles han sido descubiertas, la estación intercambiará mensajes de autenticación con el punto de acceso de una de esas redes.
4. Una vez que el proceso de autenticación a finalizado con éxito, la estación envía una petición de asociación al punto de acceso. Esta petición contiene de nuevo la misma información que la petición de sondeo además de los identificadores acordados en el proceso de autenticación.
5. Si la información que llega en la petición de asociación concuerda con las capacidades del punto de acceso, éste contestará con una respuesta de asociación con un mensaje de acceso satisfactorio para dar acceso a la red a la estación.

Todo este proceso está modelado en *ns-3*, exceptuando el proceso de autenticación. De este proceso se corrige la parte que se encarga de difundir y comprobar las tasas de datos soportadas por los dispositivos que viajan en las tramas del proceso de asociación. Esto se modela en los archivos `ap-wifi-mac.cc` y `sta-wifi-mac.cc`, en ellos se modela la capa MAC de un punto de acceso y una estación respectivamente.

En el archivo `ap-wifi-mac.cc` se encuentra la clase `ApWifiMac` y sus métodos. El método `GetSupportedRates` debe ser modificado para que, en el caso de que el modo Greenfield esté activado, la lista de modos de transmisión devuelta sea la que devuelve el método `GetMembershipSelectorModes` de la clase `YansWifiPhy`. Con ésto, cada vez que se invoque al método `GetSupportedRates` en el proceso de asociación, serán devueltas las tasas de transmisión correspondientes a los índices MCS de 0 a 7 para intervalo de guarda largo y ancho de canal de 20 MHz. Por último en este archivo, se debe modificar también la parte que gestiona la recepción de un mensaje de petición de asociación. Se tiene que comprobar si las tasas de datos soportadas por la estación son soportadas también por el punto de acceso, para ello se recorre la lista de modos de transmisión devuelta por el ya conocido método `GetMembershipSelectorModes`.

Estas correcciones realizadas en el archivo `ap-wifi-mac.cc` quedan reflejadas en el anexo C, parche C.2.

En el archivo `sta-wifi-mac.cc` se encuentra la clase `StaWifiMac` y sus métodos. La modificación del método `GetSupportedRates` es idéntica a la explicada en el párrafo anterior puesto que el uso que se le da en esta clase es exactamente el mismo. También existe un paralelismo con la modificación de la gestión de la recepción de un mensaje de petición de asociación del párrafo anterior, la modificación vuelve a ser la misma solo que en este caso enmarcada en la recepción de un mensaje de respuesta de asociación. Estas correcciones realizadas en el archivo `sta-wifi-mac.cc` quedan reflejadas en el anexo C, parche C.3.

Éstas han sido las modificaciones para hacer funcionar a los dispositivos en los modos de transmisión OFDM de la norma 802.11n. Para demostrar que son efectivas, se necesita equipar al escenario de funciones que recopilen estadísticas que demuestren que las modificaciones consiguen su objetivo. En el modelado ademas de código para modelar sistemas,

también se incluye código para la recopilación de estadísticas que ayuden a valorar si el modelado es correcto o no. Es el caso de tres métodos de la clase `WifiPhy`:

- `NotifyRxBegin` indica que un paquete esta siendo recibido.
- `NotifyRxEnd` indica que el paquete ha sido recibido correctamente.
- `NotifyRxDrop` indica que el paquete ha sido tirado debido a la tasa de error de paquete correspondiente a la modulación utilizada.

Estos métodos te dan accesibilidad desde el escenario al paquete que se encuentra en proceso de recepción. Para realizar la comprobaciones necesarias en esta corrección, es necesario tener también el modo de transmisión con el que se esta recibiendo dicho paquete para ver que las modificaciones logran que las transmisiones se hacen con los modos de transmisión deseados. Partiendo de estos tres métodos se han creado otros tres métodos idénticos (`NotifyRxBeginMode`, `NotifyRxEndMode y NotifyRxDropMode`), a estos tres métodos nuevos se les ha añadido la información del modo de transmisión con el que se esta recibiendo el paquete. La creación de estos métodos implica la modificación de los archivos `wifi-phy.cc` y `wifi-phy.h`, estas modificaciones se pueden ver en el anexo C, parches C.4 y C.5.

Con los nuevos métodos de notificación creados, queda colocarlos apropiadamente y darles uso en el escenario. La localización elegida ha sido la misma que los métodos originales. La inclusión de estos nuevos métodos queda recogida en el anexo C, parche C.6. El escenario creado para verificar que las modificaciones que se han hecho cumplen su objetivo se puede consultar en el anexo B, escenario B.2. Destacar que para hacer uso en el escenario de los métodos de notificación creados se deben incluir las siguientes sentencias:

```
Config::Connect("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/↩
    PhyRxBeginMode", MakeCallback(&PhyRxBeginMode));
Config::Connect("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/↩
    PhyRxBeginMode", MakeCallback(&PhyRxEndMode));
Config::Connect("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/↩
    PhyRxBeginMode", MakeCallback(&PhyRxDropMode));
```

4.1.3. Simulaciones de verificación

Para verificar el comportamiento deseado tras las modificaciones del modelado se han realizado baterías de simulaciones con cuatro configuraciones diferentes del escenario. Estas cuatro configuraciones difieren en los valores que toman las banderas de tipo booleano de los parámetros `ShortGuardEnabled` y `ChannelBonding`:

```
wifiPhy.Set ("ShortGuardEnabled", BooleanValue(sge));
wifiPhy.Set ("ChannelBonding", BooleanValue(ch));
```

A través de un script se realizan baterías de simulaciones en las que las variables `sge` y `ch` toman valores `true` y `false` hasta contemplar las cuatro combinaciones posibles. Para cada una de estas cuatro configuraciones se han realizado 2 baterías de 20 simulaciones cada una. La primera batería para un tiempo de simulación corto (5 segundos) y la segunda para un tiempo de simulación largo (300 segundos). De cada batería de 20 simulaciones

se ha computado la media y la desviación estándar de cada valor a juzgar para tener una visión fiable a la hora de valorar los resultados. En el script se contabilizan el número de paquetes para cada caso, se han ignorado los paquetes correspondientes a asentimientos puesto que éstos siempre son transmitidos usando el modo de transmisión por defecto.

La verificación comienza con la ejecución de la simulación del escenario con los dos parámetros configurados a `false`. Lo que se espera es que se utilicen modos de transmisión del 0 al 7 para un ancho de canal de 20 MHz e intervalo de guarda largo. La batería de simulaciones cortas arrojan que los 5835 paquetes transmitidos de media, se han transmitido con modos de transmisión para ancho de canal de 20 MHz e intervalo de guarda largo. También podemos ver como han quedado distribuidos los índices MCS utilizados.

```
ChannelBonding=false --ShortGuardEnabled=false --simtime=5

20_MHz      5835.70(12.71)  5835.30(12.64)  0.40(0.68)
LGI         5835.70(12.71)  5835.30(12.64)  0.40(0.68)

40_MHz      0.00(0.00)      0.00(0.00)      0.00(0.00)
SGI         0.00(0.00)      0.00(0.00)      0.00(0.00)

MCS         Inicio          Final           Tirados

0           11.00(0.00)     11.00(0.00)     0.00(0.00)
1           10.00(0.00)     10.00(0.00)     0.00(0.00)
2           10.00(0.00)     10.00(0.00)     0.00(0.00)
3           10.00(0.00)     10.00(0.00)     0.00(0.00)
4           10.00(0.00)     10.00(0.00)     0.00(0.00)
5           10.00(0.00)     10.00(0.00)     0.00(0.00)
6           10.00(0.00)     10.00(0.00)     0.00(0.00)
7           5713.70(12.71)  5713.30(12.64)  0.40(0.68)
```

No es momento de valorar la distribución de los índices MCS utilizados pero lo que si podemos decir es que el algoritmo de selección de tasa no ha tenido ningún problema en ir incrementando el índice MCS con el que transmitir hasta alcanzar el índice más alto. De hecho para los índices del 0 al 6 se observa un comportamiento muy estable puesto que la desviación estándar es cero en todos los casos.

En la batería de simulaciones con tiempo de simulación largo, se han transmitido correctamente 651453 paquetes, también con los modo de transmisión deseados. No se observa ninguna anomalía en los resultados dado que son proporcionales a los obtenidos con tiempo de simulación corto, lo que hace indicar que el modelado es estable con tiempos de simulación largos.

La siguiente simulación se hace con el parámetro `ChannelBonding` configurado a `false` y `ShortGuardEnabled` a `true`. En media, el número de paquetes transmitidos correctamente ha sido 6047 en 5 segundos, lógicamente superior al de la simulación anterior puesto que las tasas de los modos de transmisión para esta configuración son mayores que para la configuración anterior. Todos estos paquetes han sido transmitidos en modos de transmisión para canal de 20 MHz. Sin embargo no han sido todos los paquetes transmitidos con intervalo de guarda corto, han sido 5997 de los 6047 paquetes transmitidos. Los 50 paquetes restantes han sido transmitidos con intervalo de guarda largo, estos 50 paquetes son los correspondientes a los paquetes transmitidos en el modo por defecto, es

decir los paquetes que contienen tramas de gestión. Esto concuerda con nuestro modelado puesto que el modo por defecto que se ha elegido es el correspondiente al índice MCS 0 para ancho de canal de 20 MHz e intervalo de guarda largo. La distribución de los índices MCS utilizados es similar a la anterior:

```
ChannelBonding=false --ShortGuardEnabled=true --simtime=5

20_MHz      6047.65(13.61)  6047.25(13.56)  0.40(0.68)
LGI         50.00(0.00)     50.00(0.00)     0.00(0.00)

40_MHz      0.00(0.00)      0.00(0.00)      0.00(0.00)
SGI         5997.65(13.61)  5997.25(13.56)  0.40(0.68)

MCS         Inicio          Final           Tirados

0           11.00(0.00)     11.00(0.00)     0.00(0.00)
1           10.00(0.00)     10.00(0.00)     0.00(0.00)
2           10.00(0.00)     10.00(0.00)     0.00(0.00)
3           10.00(0.00)     10.00(0.00)     0.00(0.00)
4           10.00(0.00)     10.00(0.00)     0.00(0.00)
5           10.00(0.00)     10.00(0.00)     0.00(0.00)
6           10.00(0.00)     10.00(0.00)     0.00(0.00)
7           5925.65(13.61)  5925.25(13.56)  0.40(0.68)
```

La batería de simulaciones largas de este caso sigue confirmando que el comportamiento del modelado es estable. Para cada valor, la media y desviación estándar son del orden de magnitud que se esperan. Se han transmitido correctamente 676329 paquetes, que al igual que para el tiempo de simulación corto supera al del caso anterior.

La siguiente configuración del escenario es en la que el parámetro `ChannelBonding` está configurado a `true` y el parámetro `ShortGuardEnabled` a `false`. En principio, se deberían de esperar más paquetes puesto que las tasas de datos de los actuales modos de transmisión son superiores a los de la configuración del párrafo anterior. En este caso en 5 segundos de simulación se han transmitido correctamente 5556 paquetes, se trata de un numero inferior al del caso anterior. De media, se han iniciado un mayor número de transmisiones de paquetes (6360), pero 804 paquetes se han tirado en su recepción debido a la tasa de error de paquetes.

Vemos como a diferencia de los casos anteriores, nunca se alcanza el índice MCS superior. Esto ocurre porque en el índice MCS 6 no se logran transmitir correctamente 10 paquetes consecutivos. Como vemos, aproximadamente, solo 1 de cada 7 paquetes se trasmiten satisfactoriamente. Que casi 6 de cada 7 paquetes sean tirados para el índice MCS 6 se debe a que la tasa de error de paquete es superior para modos de transmisión con altas tasas de datos. Estos paquetes pasan a ser retransmitidos, el número de retransmisiones es un factor en el que se basan los algoritmos de selección de tasa para disminuir la tasa a la que se transmiten, lo que hace que los paquetes se pasen a transmitir con el índice MCS 5.

```
ChannelBonding=true --ShortGuardEnabled=false --simtime=5

20_MHz      50.00(0.00)      50.00(0.00)      0.00(0.00)
LGI         6360.35(12.52)   5556.10(15.28)   804.25(10.23)

40_MHz      6310.35(12.52)   5506.10(15.28)   804.25(10.23)
SGI         0.00(0.00)       0.00(0.00)       0.00(0.00)

MCS         Inicio           Final            Tirados

0           11.00(0.00)      11.00(0.00)      0.00(0.00)
1           10.00(0.00)      10.00(0.00)      0.00(0.00)
2           10.00(0.00)      10.00(0.00)      0.00(0.00)
3           10.00(0.00)      10.00(0.00)      0.00(0.00)
4           105.00(32.69)    105.00(32.69)    0.00(0.00)
5           5473.05(29.69)   5236.30(30.37)   236.75(10.05)
6           690.30(24.60)    122.80(14.70)    567.50(11.66)
7           0.00(0.00)       0.00(0.00)       0.00(0.00)
```

Para este caso, el índice MCS 5 es al que converge el algoritmo de selección de tasa, en él se pueden dar 10 transmisiones exitosas consecutivas para subir al índice superior, o varias retransmisiones consecutivas que hagan bajar al índice inferior. En el índice MCS 4 no se tira ningún paquete por lo que se vuelve al índice MCS 5 sin problema.

De los 5556 paquetes recibidos de media, 5506 han sido para ancho de banda de 40 MHz y los 50 restantes transmitidos, como en escenarios anteriores, corresponden a paquetes en los que viajan tramas de gestión. Paquetes que como hemos dicho son transmitidos con el modo de transmisión por defecto.

En la batería con tiempo de simulación larga se han alcanzado los 614036 paquetes recibidos correctamente. Al igual que con el tiempo de simulación corto, este valor es inferior al del caso anterior para el mismo tiempo de simulación. Tampoco se ha observado un comportamiento inestable del modelado, los valores estadísticos se siguen manteniendo en cotas razonables.

Por último queda configurar ambos parámetros a `true`. En media, el numero total de paquetes transmitidos es 5684, 50 de esos paquetes han vuelto a ser transmitidos en el modo por defecto (MCS 0, intervalo de guarda largo y canal de 20 MHz). Los 5634 paquetes restantes se transmiten de con intervalo de guarda corto y ancho de canal de 40 MHz. La distribución de los índices MCS para esta configuración es la siguiente:

```
ChannelBonding=true --ShortGuardEnabled=true --simtime=5

20_MHz      50.00(0.00)     50.00(0.00)     0.00(0.00)
LGI         50.00(0.00)     50.00(0.00)     0.00(0.00)

40_MHz      6461.60(18.97)  5634.45(24.81)  827.15(12.50)
SGI         6461.60(18.97)  5634.45(24.81)  827.15(12.50)

MCS         Inicio          Final           Tirados

0           11.00(0.00)     11.00(0.00)     0.00(0.00)
1           10.00(0.00)     10.00(0.00)     0.00(0.00)
2           10.00(0.00)     10.00(0.00)     0.00(0.00)
3           10.00(0.00)     10.00(0.00)     0.00(0.00)
4           112.50(36.11)   112.50(36.11)   0.00(0.00)
5           5588.75(46.81)  5347.55(45.34)  241.20(10.74)
6           718.35(31.96)   132.40(20.26)   585.95(13.13)
7           0.00(0.00)      0.00(0.00)      0.00(0.00)
```

El comportamiento del algoritmo de selección de tasa vuelve a ser el mismo al descrito en el caso anterior. Se ha iniciado un mayor numero de transmisiones de paquetes en media (6511), a su vez el número de paquetes con fallo de transmisión también ha aumentado. Lógico que en un mismo periodo de tiempo el número de paquetes a transmitir haya sido mayor, ya que para este caso las tasas de transmisión son mas altas.

629403 han sido los paquetes recibidos correctamente de media en la batería con tiempo de simulación largo. Con este último caso se ha demostrado también estabilidad en el modelado puesto que los valores obtenidos son proporcionales a los de la batería con tiempo de simulación corto.

A modo de resumen se presenta la siguiente tabla en la que podemos ver el número de paquetes recibidos correctamente, junto con sus valores estadísticos, para cada batería de simulación, se han ordenado de mayor a menor número de paquetes con el objetivo de ver claramente cuales han sido los resultados de cada configuración frente a las demás:

	Tiempo de simulación			
	5 segundos		300 segundos	
	Media	Desviación Estándar	Media	Desviación Estándar
20 Mhz, GI = 400 ns	6047	13.56	676329	447.64
20 Mhz, GI = 800 ns	5835	12.64	651454	158.24
40 Mhz, GI = 800 ns	5684	24.81	629403	513.65
40 Mhz, GI = 400 ns	5556	15.28	614036	302.48

Tabla 4.1: Comparativa de paquetes recibidos correctamente

Vistos los datos arrojados por las simulaciones con el escenario de verificación, podemos decir que el modelado introducido en *ns-3* consigue alcanzar el objetivo de este apartado: que los dispositivos puedan transmitir datos con los modos de transmisión de la tabla 2.2 según la configuración del ancho de banda del canal y de la longitud del intervalo de guarda.

Con estas simulaciones también hemos podido observar como un algoritmo de selección de tasa puede ser ineficiente. A pesar de disponer de tasas de datos mas elevadas para transmitir, la mala gestión de las mismas puede hacer que el número total de paquetes recibidos correctamente sea inferior al que se recibirían con índices MCS con tasas de transmisión inferiores.

4.2. Sensibilidad de recepción de señal para modulaciones OFDM de 802.11n

En el apartado anterior se ha visto como la tasa de error de paquetes puede hacer que algunos paquetes sean recibidos de manera errónea debido a que para una determinada modulación la señal no tiene suficiente energía respecto al ruido. Cuanto más símbolos tenga una modulación más alta será la tasa de error de paquetes.

La demodulación correcta de una señal en el receptor está condicionada por el número de símbolos utilizados en su modulación, por la tasa de codificación y por la potencia de la señal recibida. Es decir, a cada par formado por una modulación y una tasa de codificación le corresponde un valor mínimo de potencia recibida. Este valor mínimo de la potencia necesaria para demodular correctamente una señal es dependiente de las especificaciones del hardware, que otorga diferentes valores según el ancho de banda del canal. Lo denominaremos sensibilidad mínima de recepción.

El objetivo de este apartado es implementar la sensibilidad mínima de recepción en *ns-3*, o lo que es lo mismo modelar la corresponencia entre el valor mínimo de potencia recibida y el par formado por la modulación y tasa de codificación de una transmisión.

4.2.1. Análisis de la solución adoptada

Implementar una nueva característica dentro de un modelado requiere de un análisis previo que determine cual debe ser la manera en el que el nuevo código se va a integrar en el código existente. En este caso la integración de esta característica no requiere de una implementación que afecte a varias capas del modelo OSI. Tampoco necesita que varios archivos del código fuente sean modificados, por lo que el análisis se centra únicamente en el emplazamiento que se le va a dar en el archivo donde se modela la recepción de los paquetes.

La elección del lugar en el que se decide si un paquete llega con una energía suficiente como para ser demodulado se ha hecho bajo criterios de simplicidad y bajo impacto en el modelado de *ns-3*. Es por esta razón que el sitio elegido es el mismo en el que los desarrolladores han decidido poner el umbral de energía ya existente. Lo que se ha hecho es introducir la sensibilidad de recepción de señal solo para paquetes que hayan sido transmitidos en modos HT, ya que son los modos de transmisión correspondientes al modelado de la norma 802.11n.

La implementación de la solución a esta carencia en el modelado de *ns-3* está fundamentada en el conocimiento del proceso de recepción de paquetes recogido en el anexo A.3.

4.2.2. Modelado de la solución

La implementación de esta característica solo necesita modificar los archivos de la clase `YansWifiPhy`. En el método `StartReceivePacket` comienza la recepción de un paquete en la capa PHY de una interfaz de red. En el caso de que el estado de la capa PHY sea ocioso, se comienza con la recepción del paquete. En este punto se ha creado una variable local booleana en la que almacenar si la potencia de señal del paquete que se está recibiendo es superior al límite que impone la modulación con la que ha sido transmitido. Esta variable se utiliza en el mismo bloque `if` que el del umbral existente.

Esta variable booleana se llama `ofdmThreshold`, si es igual a `true` indicará que el paquete supera el umbral de potencia de señal que impone la modulación con la que ha sido transmitido, su valor será `false` para el caso contrario. `ofdmThreshold` se inicia a `true` para que, en el caso de que la modulación con la que llega un paquete no sea alguna de las de la norma 802.11n, no se tire ningún paquete a causa de los umbrales introducidos.

La variable `ofdmThreshold` va a guardar el valor devuelto por el método `IsOfdmThreshold` de la clase `YansWifiPhy`. Este método recibe como parámetros el modo de transmisión y la potencia con la que se está recibiendo el paquete. Con esos dos parámetros el método analiza el modo de transmisión para determinar su índice MCS y el ancho de banda del canal. Estos dos datos son los necesarios para determinar el valor mínimo que debe de tener la potencia para que la señal pueda ser demodulada.

A este valor mínimo lo llamaremos sensibilidad de recepción, y existe una sensibilidad diferente para cada par formado por índice MCS (modulación y tasa de codificación) y ancho de banda. Los valores de sensibilidad de recepción forman parte de las especificaciones hardware del receptor de la interfaz de red de cualquier dispositivo. Por tanto estos valores son dependientes del hardware, en este modelado se van a utilizar unos valores genéricos obtenidos de un documento de investigación de un fabricante de tecnologías inalámbricas [16]. Estos valores garantizan que el receptor es capaz de demodular correctamente una señal a un mínimo nivel de potencia. La demodulación correcta viene determinada por una tasa de error de paquete menor del 10 %.

			Sensibilidad mínima (dBm)	
Índice MCS	**Modulación**	**Tasa de Codificación**	**20 Mhz**	**40 Mhz**
0	BPSK	1/2	-82	-79
1	QPSK	1/2	-79	-76
2	QPSK	3/4	-77	-74
3	16-QAM	1/2	-74	-71
4	16-QAM	3/4	-70	-67
5	64-QAM	2/3	-66	-63
6	64-QAM	3/4	-65	-62
7	64-QAM	5/6	-64	-61

Tabla 4.2: Sensibilidad mínima de recepción

El método `IsOfdmThreshold` utiliza estos valores para decidir si la potencia recibida es suficiente como para alcanzar la sensibilidad mínima correspondiente al modo de transmisión. En caso de que no sea suficiente el método devuelve `false`, por contra si la potencia recibida está por encima del valor correspondiente, el método devuelve `true`.

A la clase `YansWifiPhy` se le ha dotado de un nuevo atributo para dar la posibilidad de configurar la sensibilidad mínima de recepción desde el escenario de simulación. De esta manera se podrá bajar el nivel de sensibilidad de acuerdo con las especificaciones que se le quieran dar al dispositivo. Este parámetro se llama `m_extraSensitivity` y es útil para ajustar los niveles de sensibilidad a los del modelado, es decir, configurar el nivel de sensibilidad mínimo de acuerdo con el comportamiento del modelo de tasa de error que se esté usando. Si se quieren añadir 5 dBm de sensibilidad extra al dispositivo, la tabla verá sus valores modificados.

Ejemplo: `m_extraSensitivity` = 5 dBm, MCS 0 y 20 MHz: -82 - 5 = -87 dBm.

Estas modificaciones realizadas en los archivos `yans-wifi-phy.cc` y `yans-wifi-phy.h`, se pueden ver en el anexo C, parches C.7 y C.8.

De la misma manera que se hizo en el apartado anterior, se han creado funciones de notificación que ayuden tanto al desarrollo de la corrección del modelado como a la recopilación de estadísticas. Estas funciones son muy similares a las anteriormente creadas y por tanto se necesita de la modificación de los archivos `wifi-phy.cc` y `wifi-phy.h`. Los parches C.9 y C.10 del anexo C, recogen la inclusión de estas funciones.

4.2.3. Simulaciones de verificación

El escenario de simulación creado para la verificación del modelado de este apartado es una adaptación del escenario del apartado anterior. A continuación se enumeran las principales novedades de este escenario:

- Un nuevo parámetro de entrada llamado `powerlimit`. Este parámetro se utiliza para calcular cual debe ser la distancia que separa al punto de acceso del cliente para que la potencia de señal recibida por el cliente sea la deseada. Este cálculo se hace en la función creada a tal efecto llamada `powerToDist`. Los datos que se utilizan para este cálculo se extraen de los objetos del escenario: por un lado se extrae la potencia con la que se transmite la señal del emisor, a través del atributo `m_txPowerBaseDbm` de la clase `YansWifiPhy`; y por otro se extraen los datos de propagación del modelo de propagación configurado por defecto, `LogDistancePropagationLossModel` (modelo aceptado para las perdidas por propagación en los ambientes típicos de las redes locales inalámbricas).
- `ExtraSens`: nuevo parámetro de entrada que sirve para poder ejecutar simulaciones con una sensibilidad mayor a la de los valores base.
- Se ha enriquecido la recopilación de estadísticas con el uso de las funciones de notificación creadas. Estas estadísticas aportan el nivel de detalle que se necesita para verificar que las modificaciones introducidas cumplen con su objetivo.

El escenario resultante se puede consultar en el anexo B, escenario B.3.

Para realizar las diferentes simulaciones de verificación se utiliza un script que realiza baterías de 20 simulaciones para las diferentes configuraciones. Dos son las configuraciones, una para ancho de canal de 20 MHz y otra para ancho de 40 MHz. Para cada caso los valores de sensibilidad son diferentes, así que a cada ancho de canal le corresponde una lista de valores límites. Los valores límite no son más que la potencia de señal que se desea recibir para así comprobar que los límites de sensibilidad se han modelado correctamente.

Dentro del script se da también la posibilidad de dotar al modelo con una mayor sensibilidad partiendo de los valores base. Lógicamente este valor se añade también a la lista de valores límites para hacer las simulaciones a los niveles de potencia deseados en cada caso. Para cada configuración por tanto se realizan dos baterías de 20 simulaciones cada una, una batería de simulaciones para un tiempo de simulación corto (20 segundos) y otra batería con tiempo de simulación largo (300 segundos) para una verificación mas sólida. A continuación se analizan los resultados obtenidos de estas simulaciones.

Primero se realiza una ejecución del script con los valores de sensibilidad base, es decir, con el parámetro `ExtraSens` a cero. A priori no se sabe cual va a ser el comportamiento del modelo tras esta comprobación previa al cálculo de la tasa de error de paquete. Los datos que arroja esta ejecución es que el modelado está funcionando a la perfección, pero que los valores de sensibilidad mínima son demasiado restrictivos puesto que nunca se llega a tirar ningún paquete debido a la tasa de error de paquete (ni siquiera con tiempo de simulación largo). Como muestra de lo que ocurre analizamos el resultado de una configuración concreta:

```
simtime=300 ch=false powerlimit=-66 ExtraSens=0

MCS  Inicio              Final             Tirados          Tirados
                                           (Sens)           (PER)
0    15.00(0.00)         15.00(0.00)       0.00(0.00)       0.00(0.00)
1    10.00(0.00)         10.00(0.00)       0.00(0.00)       0.00(0.00)
2    10.00(0.00)         10.00(0.00)       0.00(0.00)       0.00(0.00)
3    10.00(0.00)         10.00(0.00)       0.00(0.00)       0.00(0.00)
4    10.00(0.00)         10.00(0.00)       0.00(0.00)       0.00(0.00)
5    531193.00(313.97)   531193.00(313.97) 0.00(0.00)       0.00(0.00)
6    53118.70(31.28)     0.00(0.00)        53118.70(31.28)  0.00(0.00)
7    0.00(0.00)          0.00(0.00)        0.00(0.00)       0.00(0.00)
```

Si consultamos la tabla 4.2, vemos que la sensibilidad mínima para el índice MCS 5 y ancho de canal de 20 MHz es -66 dBm. Se corresponde con el valor configurado para la potencia con la que la señal llega al receptor (`powerlimit=-66`). Esta correspondencia perfecta se da en cada una de las configuraciones. Cualquier paquete que sea transmitido con un índice MCS superior a 5 se tira por no alcanzar el valor mínimo de potencia en el receptor. Hasta este punto el comportamiento es el deseado. Lo que no concuerda con un comportamiento real es que usando el índice MCS en el que converge el algoritmo de selección de tasa no se tire ningún paquete debido a la tasa de error de paquete.

Para evitar este comportamiento erróneo del modelado se prueban varias ejecuciones del escenario de simulación para diferentes valores del parámetro `ExtraSens`. De estas pequeñas simulaciones se concluye que se deben añadir 6 dB extra a la sensibilidad de recepción, lo que dejaría la tabla de sensibilidad mínima de recepción de la siguiente manera:

			Sensibilidad mínima (dBm)	
Índice MCS	**Modulación**	**Tasa de Codificación**	**20 Mhz**	**40 Mhz**
0	BPSK	1/2	-88	-85
1	QPSK	1/2	-85	-82
2	QPSK	3/4	-83	-80
3	16-QAM	1/2	-80	-77
4	16-QAM	3/4	-76	-73
5	64-QAM	2/3	-72	-69
6	64-QAM	3/4	-71	-68
7	64-QAM	5/6	-70	-67

Tabla 4.3: Sensibilidad mínima de recepción +6 dB

El análisis para estos nuevos valores de sensibilidad, se realiza escogiendo los resultados obtenidos de la misma configuración que el caso anterior:

```
simtime=300 ch=false powerlimit=-72 ExtraSens=6

MCS  Inicio              Final              Tirados          Tirados
                                            (Sens)           (PER)

0    15.00(0.00)         15.00(0.00)        0.00(0.00)       0.00(0.00)
1    10.00(0.00)         10.00(0.00)        0.00(0.00)       0.00(0.00)
2    10.00(0.00)         10.00(0.00)        0.00(0.00)       0.00(0.00)
3    10.00(0.00)         10.00(0.00)        0.00(0.00)       0.00(0.00)
4    1165.30(111.87)     1165.30(111.87)    0.00(0.00)       0.00(0.00)
5    530427.90(348.15)   522193.40(386.11)  0.00(0.00)       8234.50(78.38)
6    49968.65(58.94)     0.00(0.00)         49968.65(58.94)  0.00(0.00)
7    0.00(0.00)          0.00(0.00)         0.00(0.00)       0.00(0.00)
```

Ahora se observa un comportamiento mucho mas acorde a la realidad. El índice MCS 5 es al que converge el algoritmo de la selección de tasa pero a diferencia de los resultados anteriores, aquí se tiran paquetes a causa de la tasa de error de paquetes. El porcentaje de paquetes tirados por la tasa de error de paquete debe ser inferior al 10 % para cumplir con la definición de sensibilidad de recepción. En este caso el porcentaje de paquetes tirados por error de paquete es un 1.57 %.

El resto de resultados de las distintas configuraciones se mantienen en las cotas esperadas y con comportamientos correctos. Por tanto se da por buena la codificación realizada de la sensibilidad mínima de recepción. Se ha demostrado la conexión que hay entre esta sensibilidad mínima y el modelo de tasa de error utilizado. Cada vez que se quiera cambiar algo del modelo de la tasa de error, se deberá calibrar los valores de sensibilidad mínima con la ayuda del parámetro `ExtraSens`.

4.3. Gestión de asociaciones entre APs y STAs

En este último apartado lo que se quiere conseguir es modelar la gestión de asociaciones entre un punto de acceso y un cliente o estación de la manera más acorde al modelo real. En el apartado 4.1.2 se analizó el modelado del proceso de autenticación y se concluyó que modelaba con suficiente rigor el proceso de asociación de un cliente a un punto de acceso. Por otro lado en la gestión de una asociación entre cliente y punto de acceso se tienen que contemplar cada uno de los casos por los que la asociación puede terminar.

El caso que motiva este apartado es la pérdida de conexión entre punto de acceso y cliente debido al movimiento de un cliente. La pérdida de conexión se produce porque en su movimiento el cliente pasa de estar en una zona cubierta por el punto de acceso a estar a una distancia a la que las señales no llegan con potencia suficiente como para ser recibidas correctamente. Es decir, como comúnmente se dice: "El punto de acceso deja de ver al cliente y el cliente deja de ver al punto de acceso".

En este apartado por tanto se trata de modelar, de la manera más fiel a la realidad, el comportamiento de un punto de acceso y un cliente en el caso de que un cliente ya asociado se salga de la zona de cobertura del punto de acceso al que está conectado.

4.3.1. Análisis de la solución adoptada

La solución a esta deficiencia del modelado pasa por dotar al punto de acceso de un mecanismo que le haga detectar que un cliente conectado se sale de la zona cubierta por las radiobalizas y por tanto pierde conexión con el mismo. No se puede conocer cual es el mecanismo utilizado por los puntos de acceso con software propietario, pero si de los controladores de código abierto. Éste es el caso de los puntos de acceso con controladores basados en Linux, que cuentan con un software, llamado hostapd, que implementa la gestión de puntos de acceso basados en la norma 802.11.

De este software son conocidos los códigos de cada motivo por el cual un punto de acceso deja de tener a un cliente conectado/asociado/autenticado. De todos los motivos que existen se elige el que corresponde con el error que se quiere corregir: "Desasociación por inactividad - Tiempo de espera para sesión de cliente superado". La desasociación por inactividad se produce cuando el punto de acceso lleva un tiempo superior al tiempo de espera establecido sin recibir ningún paquete de la estación en cuestión.

Se decide por tanto que la solución sea modelar en *ns-3* el mecanismo por el cual un punto de acceso desasocia a un cliente por inactividad. Para modelar esta solución, se necesita definir lo que se va a considerar inactividad. El punto de acceso va a determinar que un cliente está inactivo cuando pase un tiempo determinado sin recibir ningún paquete del cliente. Esta definición de inactividad puede hacer que un punto de acceso determine que un cliente está inactivo en situaciones indeseadas. Por ejemplo: en *ns-3* se puede crear un escenario en el que el único trafico existente sea un tráfico de datos UDP que el punto de acceso manda al cliente. En este caso el cliente (no envía asentimientos a los datos recibidos) puede pasar un largo periodo de tiempo sin enviar datos al punto de acceso, tiempo en el que el punto de acceso consideraría que el cliente esta inactivo.

Para evitar estas situaciones indeseadas se crea un nuevo paquete de gestión que los clientes podrán enviar periódicamente al punto de acceso al que están asociados. El modelo

contará por tanto con la posibilidad de activar el envío de este paquete de gestión en los escenarios en los que se requiera.

4.3.2. Modelado de la solución

El paquete de gestión que se ha creado ha sido codificado en base al paquete de gestión correspondiente a una radiobaliza. El primer paso para la creación de este paquete de gestión es la creación de la cabecera de la capa MAC, la inclusión de esta cabecera en el modelo necesita de la modificación de los archivos `mgt-headers.cc`, `mgt-headers.h`, `wifi-mac-header.cc` y `wifi-mac-header.h`. Las modificaciones necesarias para crear la cabecera `MgtStaFeedbackHeader` quedan recogidas en el anexo C, parches C.11, C.12, C.13 y C.14. En el modelo real no existe ninguna cabecera de estas características en la capa MAC, en este modelo dicha cabecera se utiliza simplemente como herramienta para modelar la gestión de asociaciones de la forma más fiel posible.

El paquete que utiliza esta cabecera es un paquete vacío que se crea en la capa MAC de la interfaz de red del cliente. Se decide colocar la creación y envío de este paquete en una función que se ejecuta periódicamente en el modelado que hace *ns-3* de la capa MAC de la interfaz de red de un cliente. Se trata de la función `MissedBeacons`, esta función se encarga de comprobar periódicamente que se reciben radiobalizas enviadas por el punto de acceso al que el cliente está conectado. En consecuencia, se enviarán paquetes con la nueva cabecera de gestión siempre y cuando no se dejen de recibir radiobalizas enviadas por el punto de acceso al que está conectado dicho cliente.

Para conocer las modificaciones necesarias de los archivos `sta-wifi-mac.cc` y `sta-wifi-mac.h` para la codificación del modelado de la capa MAC de la interfaz de red de un cliente basta con consultar los parches C.15 y C.16 del anexo C. Destacar la inclusión en la clase `StaWifiMac` del atributo `StaFeedback` que hace accesible desde el escenario la activación o desactivación del envío de este nuevo tipo de trama de gestión creada. Para activar esta característica desde el escenario de simulación, se configura la capa MAC de la interfaz de red del cliente de la siguiente manera:

```
wifiMac.SetType ("ns3::StaWifiMac",
    "Ssid", SsidValue (Ssid ("Wi-Fi PFC")),
    "StaFeedback",BooleanValue (true));
```

El envío de esta nueva trama de gestión por parte de un cliente tiene siempre por destino la interfaz de red de un punto de acceso. Antes de abordar las modificaciones necesarias de la clase que modela dicha interfaz, nos detenemos en su clase padre: `RegularWifiMac`. En esta parte del modelado se debe contemplar la recepción de esta nueva trama de gestión, porque en caso de no contemplarse salta un error fatal que termina con la simulación en proceso. En el punto en el que esta trama necesita ser contemplada no se va a incluir ninguna modificación correspondiente al modelado de la solución que nos ocupa, por lo que simplemente se ignora la trama en cuestión. Esta clase está codificada en el archivo `regular-wifi-mac.cc` y la modificación necesaria para ignorar la nueva trama queda recogida en el anexo C, parche C.17.

El receptor del nuevo mensaje creado para esta solución es el punto de acceso, más concretamente la capa MAC de su interfaz de red. Este mensaje le sirve al punto de acceso para

detectar si un cliente ha superado el tiempo de inactividad límite. En caso de ser superado este tiempo límite, el punto de acceso borra al cliente en cuestión de la lista de estaciones conectadas. En la clase `ApWifiMac` se encuentra el modelado de la capa MAC de la interfaz de red de un punto de acceso, y por tanto son los archivos `ap-wifi-mac.cc` y `ap-wifi-mac.h` los que se han modificado. Las modificaciones que a continuación se explican se encuentran en el anexo C, parches C.18 y C.19.

Para dotar a la clase `ApWifiMac` de las variables necesarias para la implementación de la solución adoptada, se han incluido cuatro nuevos atributos privados:

- `m_macFrom` es un vector de direcciones MAC en el que se almacenan cada una de las direcciones físicas de la tarjeta de red de las estaciones conectadas al punto de acceso.
- `m_feedbackTimeout` es un vector de variables de tiempo que almacena, para cada estación conectada, el tiempo en el que llega el último paquete con cabecera `MgtStaFeedbackHeader`. El índice en el que se almacena el tiempo correspondiente a una estación en concreto, es el mismo índice que en `m_macFrom`, donde se guarda la dirección MAC de la estación en cuestión. Este tiempo servirá para compararlo con el tiempo actual y así determinar si la estación muestra inactividad o no.
- `m_sessionTimeout` es la variable de tiempo que almacena el tiempo de inactividad límite a partir del cual una estación pasa a considerarse desconectada.
- `m_feedbackEnabled` variable booleana iniciada a `false` y que cambia a `true` en caso de que el punto de acceso detecte que las estaciones están utilizando el mecanismo creado para el modelado de la solución de este apartado.

En el momento en el que un punto de acceso termina el proceso de asociación de una estación y la considera como conectada, se crea una entrada en el vector `m_macFrom` con la dirección MAC de dicha estación y otra entrada vacía en el vector `m_feedbackTimeout` para tener correspondencia entre los índices de los vectores. Este es el momento también en el que se arranca la ejecución periódica de la función `StaFeedback`. Ésta es la función encargada de comprobar que el tiempo de inactividad de una estación no es superior al umbral establecido (`m_sessionTimeout`). En caso de superarse dicho umbral, el punto de acceso elimina de los vectores las entradas correspondientes a la estación que se registra como desasociada. Los valores de las entradas del vector `m_feedbackTimeout` se actualizan con cada recepción de un paquete con la cabecera `MgtStaFeedbackHeader`.

El atributo `SessionTimeout` se ha creado para dar acceso, desde el escenario de simulación, a la configuración de su valor. En su definición se impone que su valor sea superior a 1500 ms puesto que cualquier valor inferior podría desencadenar una desconexión indeseada de una estación por tratarse de un valor demasiado bajo para un umbral de inactividad. A continuación se muestra un ejemplo de como configurar este valor desde el escenario para un tiempo umbral de 10 segundos:

```
wifiMac.SetType ("ns3::ApWifiMac",
  "Ssid", SsidValue (Ssid ("Wi-Fi PFC")),
  "SessionTimeout", TimeValue (MilliSeconds (10000)));
```

Estas modificaciones de la clase `ApWifiMac` garantizan un correcto registro de las desconexiones de las estaciones, pero no corrigen el hecho de que el punto de acceso continúe enviando datos a la estación como si ésta estuviera aun conectada. Para corregir este comportamiento se han hecho pequeñas modificaciones en las clases `DcaTxop` y `EdcaTxopN` que son las clases que modelan el manejo de la cola de paquetes, la fragmentación y retransmisión de paquetes en la capa MAC sin soporte para calidad de servicio y con soporte para calidad de servicio respectivamente.

Lo que se ha hecho en ambos casos es condicionar el envío de paquetes de datos al estado de la estación a la que van a ser enviados. En caso de que la dirección MAC a la que van a ser enviados los datos corresponda con una estación registrada como conectada, se envían los datos con normalidad. En caso contrario, se purga la cola de transmisión y se suspende toda transmisión de paquetes de datos a esa estación. El código de las clases modificadas se encuentra en los archivos `dca-txop.cc` y `edca-txop-n.cc`. Las modificaciones necesarias en estas clases quedan recogidas en los parches C.20 y C.21 del anexo C.

4.3.3. Simulaciones de verificación

Para este apartado se ha creado un escenario de simulación apropiado para la verificación del correcto comportamiento del modelado de la gestión de asociaciones. La necesidad de un patrón de movilidad concreto, ha obligado a crear un nuevo modelo de movilidad a partir de uno ya existente. El modelo base elegido ha sido el correspondiente a un movimiento de velocidad constante en cada una de las tres coordenadas posibles. Su nombre es ConstantVelocityMobilityModel y se ha modificado de tal manera que cuando el nodo en el cual esta instalado alcance un determinado punto, la constante de su velocidad cambie de signo y por tanto, el nodo en cuestión pase a moverse en la misma dirección pero en sentido contrario.

El escenario de simulación cuenta con un punto de acceso con posición fija y dos clientes con el nuevo patrón de movimiento instalado. Los dos clientes se irán alejando del punto de acceso a una velocidad constante hasta alcanzar una distancia 5 metros superior a la distancia máxima a la que se pueden comunicar un cliente y un punto de acceso. Alcanzada esta distancia los dos clientes recorrerán el mismo camino pero en sentido opuesto al inicial, de esta manera, ambos volverán a entrar en la zona cubierta por el punto de acceso, recuperando la conexión y el flujo de datos que tenían anteriormente.

Al algoritmo de selección de tasa se le deben asignar unos parámetros menos restrictivos que le dote de la flexibilidad necesaria para adaptar los cambios de tasa al movimiento de los clientes. Otorgando valores 10 veces superiores a los valores por defecto, se consigue que los cambios de tasa se produzcan con una frecuencia menor. Con una menor frecuencia de cambio de tasa se baja la probabilidad de que el algoritmo seleccione una tasa superior justo en el momento en el que el cliente se distancia lo suficiente como para no ser capaz de recibir correctamente los datos transmitidos a esa nueva tasa, lo que provocaría un aumento del número de paquetes tirados o en el peor de los casos una pérdida de la conexión.

Como ya se ha visto, ArfWifiManager es el algoritmo de selección de tasa por defecto para transmisiones de datos inalámbricas basadas en la norma 802.11n. La asignación de los nuevos valores para sus parámetros se puede realizar de manera sencilla desde el

escenario, simplemente se deben incluir el siguiente bloque:

```
wifi.SetRemoteStationManager ("ns3::ArfWifiManager",
  "TimerThreshold", UintegerValue (150),
  "SuccessThreshold", UintegerValue (100));
```

El resto de detalles del escenario se pueden consultar en el anexo B, escenario B.4.

Para verificar el correcto modelado introducido en este apartado, se prueba el comportamiento correcto del nuevo modelado en tres situaciones que se producen durante la ejecución del escenario descrito:

Descripción de la situación	Comportamiento correcto	Verificación
El cliente alcanza el primer punto de pérdida de conexión con el punto de acceso	Transcurrido el tiempo configurado, el punto de acceso interrumpe el tráfico de datos tras detectar la situación de inactividad del cliente	El número de paquetes que inician su transmisión en el punto de acceso permanece constante tras la pérdida de conexión
El cliente recupera la conexión con el punto de acceso tras la reentrada en la zona de cobertura provocada por el cambio de sentido en su desplazamiento	El tráfico entre punto de acceso y cliente existente antes de la desconexión es reiniciado	El número de paquetes que son recibidos correctamente en el cliente aumenta
El cliente alcanza el segundo punto de pérdida de conexión con el punto de acceso	Transcurrido el tiempo configurado, el punto de acceso interrumpe el tráfico de datos tras detectar la situación de inactividad del cliente	El número de paquetes que inician su transmisión en el punto de acceso permanece constante tras la pérdida de conexión

Tabla 4.4: Protocolo de verificación del modelado de la gestión de asociaciones

Para verificar el comportamiento correcto del modelado se han realizado dos baterías de 20 simulaciones, una batería para un ancho de canal de 20 MHz y otra batería para un ancho de canal de 40 MHz. Se ha seguido el protocolo de verificación de la tabla para cada una de las simulaciones. Son 3 las verificaciones por cada semilla, por tanto tendremos 60 pruebas para cada ancho de canal.

El resultado para 20 MHz es de 59 pruebas satisfactorias y 1 fallida, mientras que para 40 MHz las 60 pruebas han sido satisfactorias. La prueba fallida corresponde a la situación de la tercera fila de la tabla, 18 ha sido la diferencia en número de paquetes que ha provocado este único fallo.

A la vista de los resultados, las modificaciones introducidas hacen un modelado correcto de la gestión de la conexión entre un punto de acceso y cliente en el caso de que un cliente ya asociado se salga de la zona de cobertura del punto de acceso al que está conectado

5. Líneas de continuación

El correcto modelado de cualquier elemento de la vida real esta pensado para ser utilizado en simulaciones que ayuden a desarrollar, de manera mas ágil, elementos que serán reproducidos en la realidad. El modelado desarrollado en este proyecto fin de carrera está pensado para que en un futuro sirva de base al diseño de un algoritmo de selección de tasa. El diseño de este algoritmo tiene como objetivo optimizar la selección del índice MCS con el fin de evitar retransmisiones excesivas o por el contrario el desaprovechamiento de la velocidad de transmisión.

A continuación se ofrece una sólida base de conocimiento con la que debe partir toda persona que tenga como objetivo diseñar un algoritmo de estas características y que posteriormente quiera simular su comportamiento.

5.1. Motivación

Las redes inalámbricas han sido una de las tecnologías que más han evolucionado en los últimos tiempos, incorporando a sus sucesivas normas nuevas herramientas con las que mejorar el rendimiento de este tipo de redes. Para que estas tecnologías sean implementadas de manera que se mantenga compatibilidad entre dispositivos, existen organizaciones internacionales que se encargan de definir como implementar los nuevos avances tecnológicos manteniendo la compatibilidad global. En el caso de las redes inalámbricas la encargada de esta tarea es la IEEE.

Así para las redes de área local inalámbricas la IEEE publicó la norma 802.11 y la revisión que nos atañe la norma 802.11n[2]. En ella se recogen todas las novedades tecnológicas que ofrece la norma, pero también se dejan sin definir algunos procedimientos de utilización de dichas novedades tecnológicas. A estos procedimientos sin definir los llamaremos grados de libertad de la norma. Estos grados de libertad son los procedimientos que, respetando la compatibilidad entre dispositivos, quedan sin definir y que el fabricante o creador del controlador deberá decidir como implementar para sacarle el mayor partido a la tecnología.

5.1.1. Grados de libertad de la norma

En la norma 802.11n podemos encontrar como queda abierta la elección de algoritmos tan críticos como el de selección del índice MCS con el que un AP transmite en un determinado momento. Asimismo se pueden elegir el número de reintentos de una cierta operación o el índice MCS y potencia con la que un AP transmite las tramas de datos, control o gestión.

Desde la aparición de la primera versión de la norma 802.11, han sido muchas las implementaciones de estos grados de libertad con el objetivo de obtener un rendimiento óptimo. Muchas de las implementaciones de estos grados de libertad habían alcanzado su madurez con la norma 802.11g, haciéndose populares por ejemplo varios algoritmos de selección de la tasa de datos. Lo que ha ocurrido con la llegada de la norma 802.11n es que estos algoritmos, que se habían hecho populares, han dejado de ser efectivos para la norma 802.11n. Estos algoritmos dejan de ser efectivos por las incorporaciones tecnológicas de la norma 802.11n (MIMO, índice MCS, ancho de banda de 20/40 MHz).

Así que si se quiere tener un rendimiento óptimo de la red Wi-Fi lo que hay que hacer es implementar estos algoritmos tan críticos adaptados al nuevo contexto tecnológico que la norma 802.11n ofrece. A continuación queda recogida la investigación realizada con el objetivo de conocer en que punto de desarrollo tecnológico se encuentra la materia con la que se propone trabajar, o lo que es lo mismo, el estado del arte.

5.1.2. Estado del arte

Se pasa a describir a continuación en que punto de desarrollo se encuentran los algoritmos de interés para la línea de continuación propuesta, haciendo hincapié únicamente en los que en los últimos tiempos se han hecho más populares.

Algoritmos de selección de tasa de datos

Muchos son los algoritmos existentes que abordan la problemática de la selección de la tasa de datos con la que un punto de acceso transmite sus tramas. Como muestra de todos ellos, destacamos tres artículos que guardan una estrecha relación con el objetivo inicial de este proyecto.

On link rate adaptation in 802.11n WLANs [3] - este primer artículo no presenta ningún algoritmo, pero su contenido proporciona una muy buena introducción a la temática que nos ocupa. El artículo investiga la adaptación de enlace en sistemas 802.11n prácticos a través de experimentos fuera de la plataforma hardware. Los experimentos ponen de manifiesto varias conclusiones no triviales:

1. Las sencillas expansiones de algoritmos desarrollados para 802.11g resultan en mínimos beneficios para sistemas 802.11n.
2. En contra de las expectativas teóricas, en la práctica el hecho de tener varias antenas transmisoras no siempre resulta en un rendimiento mayor.
3. La selección tanto de los flujos como de las antenas es esencial para tener pleno beneficio de la tecnología MIMO.

Las conclusiones obtenidas por el artículo sirven para desarrollar una nueva métrica para la selección de flujo. Recomiendan el uso de esta nueva métrica para el desarrollo de algoritmos inteligentes de selección de tasa que pretenden alcanzar un alto rendimiento con solo cambios en el software.

MIMO Link Adaptation Algorithm [4] - propone un diseño para adaptar el enlace MIMO (MIMO Link Adaptation(LA)) en el que las capas MAC y PHY deben trabajar

de manera conjunta para sacar el mayor partido de la tecnología MIMO incorporada en la norma 802.11n. El algoritmo hace uso de la información del estado del canal (Channel State Information (CSI)) y trabaja con esa información en un bucle cerrado. Para los diferentes modos de operación MIMO, el algoritmo tiene en cuenta el impacto de los diferentes entornos de radiofrecuencia.

SmartSender [5] - Se trata de un algoritmo práctico de adaptación de la tasa que utiliza valores estadísticos y el indicador de intensidad de señal recibida (RSSI - Received Signal Strength Indicator) de los paquetes con asentimientos (ACK) para determinar la tasa de transmisión que maximiza el rendimiento. Este algoritmo por tanto combina métodos basados en estadísticas con métodos basados en mediciones de señal para tener los beneficios de ambos métodos. Este artículo en su sección *"Related Work"* hace un gran análisis del estado del arte de los algoritmos de selección de la tasa de datos. Por tanto, si se quiere ampliar los conocimientos acerca del estado del arte, se recomienda encarecidamente su lectura.

Algoritmos de selección de potencia de transmisión

Los métodos para gestionar la potencia de transmisión en redes inalámbricas de area local son muy importantes a la hora de minimizar la interferencia entre redes vecinas. Reduciendo los niveles de interferencia se consigue una mejora en el rendimiento de los enlaces de la red gracias a la consecución de una mejor relación señal-ruido.

El problema es que muchos de los dispositivos de red transmiten a la máxima potencia de salida de la que disponen. Transmitiendo al nivel más alto de potencia se producen interferencias, y por tanto, aumentará la probabilidad de que colisionen paquetes. Esto hace que, en redes con varios puntos de acceso, el rendimiento global de la red se reduzca por colisiones entre paquetes de puntos de acceso vecinos que transmiten a su máxima potencia.

Una manera de reducir las interferencias es utilizando las técnicas de control de potencia de transmisión (Transmit Power Control - TPC), que permiten a los dispositivos usar el mínimo nivel de potencia de transmisión. Un ejemplo es el conjunto de métodos de control de potencia de transmisión en 802.11 que actuando sobre cada enlace reduce las interferencias, aumentando la capacidad de la red y mejorando la reutilización del espectro. Se entiende que para minimizar la potencia de transmisión y reducir las interferencias no hay que comprometer la tasa de datos que cada cliente recibe.

A continuación se describen algunos de los mecanismos de adaptación de la potencia de transmisión que la norma 802.11 contempla. La funcionalidad de control de potencia de transmisión (TPC) especificada por la IEEE en la mejora de la norma 802.11h tiene dos funciones principales:

- **Limitar la potencia de transmisión**: El punto de acceso transmite al nivel de potencia máximo para el canal en uso. El nivel máximo de potencia es el valor mínimo entre el nivel máximo de potencia permitido por la legislación local y el nivel de potencia máximo que el dispositivo pueda ofrecer. Las estaciones en el BSS puede utilizar cualquier nivel de potencia de transmisión menor o igual que este nivel de potencia máximo fijado por el AP.

- **Mecanismo de notificación de la potencia de transmisión**: este mecanismo define un elemento de notificación del control de la potencia de transmisión que contiene un campo para la potencia de transmisión y otro para el margen del enlace. Este elemento se usa para minimizar el nivel de potencia de transmisión de los dispositivos.

La segunda función se usa para determinar el nivel de potencia de transmisión apropiado para un determinado enlace inalámbrico. Si un punto de acceso recibe una trama con el elemento de notificación TCP, incluyendo la intensidad de la señal recibida RSSI (Receiver Signal Strength Indicator - Indicador de la intensidad de señal en el receptor) y la potencia con la que el dispositivo emisor transmite, entonces el punto de acceso puede estimar la calidad del enlace (en términos de pérdidas de propagación) realizando una simple resta utilizando los valores proporcionados por el dispositivo conectado. Una vez que se han calculado las pérdidas de propagación, los dispositivos pueden ajustar su potencia de transmisión.

Muchos son los mecanismos que se han desarrollado, la funcionalidad TPC definida en la norma 802.11h permite ajustar dinámicamente la potencia con la que transmiten los dispositivos de una WLAN. Existen algunos inconvenientes del TPC para WLANs que deben de ser solucionados:

◇ Interferencias en el receptor, el problema del terminal oculto (hidden-terminal)
◇ Acceso asimétrico al canal en sistemas MIMO

Ambos problemas desembocan en retransmisiones innecesarias de paquetes y en una disminución de la tasa de datos del enlace. A continuación se explican brevemente algunos de los métodos existentes que acometen la resolución de estos problemas que surgen del control dinámico de la potencia de transmisión en una WLAN.

Miser (Minimum-energy transmission Strategy) [6] - es un algoritmo basado en el esquema de estimación de la calidad del enlace definido por TPC en la norma 802.11h. El cliente (STA) de la WLAN estima las perdidas por propagación entre si mismo y el transmisor, actualiza el estado de la transmisión de datos y luego elige, por inspección de tablas creadas, la tasa y potencia de transmisión adecuados para la transmisión de datos en curso.

Cuanto menor es la potencia de transmisión o mayor es la tasa de datos en el nivel PHY (menor tiempo de transmisión), menor será la energía consumida por intento de transmisión. A su vez, también será más probable que la transmisión falle, ésto deriva en retransmisiones y en un mayor consumo de energía. La clave del algoritmo Miser es que combina TPC con la adaptación de la tasa de datos en el nivel PHY y hace una construcción previa de una tabla que relaciona tasa de datos y potencia de transmisión. Esta tabla se indexa según el estado de la tasa de datos PHY, las pérdidas de propagación y los contadores de reenvío de tramas. Cada entrada de la tabla es la combinación óptima de tasa y potencia en términos de maximización de eficiencia energética. Las entradas de la tabla se obtienen mediante un ajuste inicial del los dispositivos transmisor y receptor, dependiendo ambos del entorno en el que están instalados y de su localización física. Un cambio significativo de la localización de los dispositivos implica un nuevo ajuste inicial.

En tiempo de ejecución, una estación inalámbrica determina tanto la mejor potencia de transmisión como la mejor tasa de datos PHY para cada intento de transmitir datos en

función de la tabla que relaciona tasa y potencia de transmisión.

Contour- Slotted power control managing [7] - el objetivo principal de este método es minimizar el nivel de interferencia cuando diferentes puntos de acceso y redes están trabajando sobre la misma zona. El algoritmo define un esquema de comunicación controlado para WLANs, los puntos de acceso de diferentes redes están sincronizados con el objetivo de evitar enlaces asimétricos. Así en cualquier instante de tiempo, todos los puntos de acceso en la red emitirán con el mismo nivel de potencia para así evitar enlaces asimétricos. Con el tiempo, mediante el uso de diferentes niveles de potencia, el sistema consigue un control de potencia por cliente para maximizar la reutilización espacial. Cada punto de acceso puede transmitir, para cada uno de sus clientes, al nivel de potencia más bajo que minimiza la interferencia con las comunicaciones inalámbricas de otros puntos de acceso.

Power control MAC for ad-hoc networks [8] - Este método define una tabla donde se consideran 10 niveles de potencia de transmisión para los puntos de acceso de una WLAN: 1 mW, 2 mW, 3.45 mW, 4.8 mW, 7.25 mW, 10.6 mW, 15 mW, 36.6 mW, 75.8 mW, y 281.8 mW, que de manera aproximada corresponden respectivamente con las siguientes distancias de alcance de transmisión: 40 m, 60 m, 80 m, 90 m, 100 m, 110 m, 120 m, 150 m, 180 m, y 250 m.

Dependiendo de la distancia entre el punto de acceso y el cliente asociado, el punto de acceso transmitirá al nivel de potencia definido en la tabla. Una vez que el sistema está calibrado y desplegado, las transmisiones experimentan un aumento del rendimiento en comparación con las transmisiones sin este método.

Symphonies [9] - Este método considera una combinación del control del nivel de potencia y del ajuste de la tasa de transmisión para cumplir con la calidad del enlace que se requiera. Symphonies define una interacción con la adaptación de tasa. En redes de área local inalámbricas (WLANs) que operan bajo las normas 802.11a/b/g, los emisores utilizan una de las diferentes tasas de transmisión para enviar paquetes. La selección de dicha tasa esta determinada por una estimación de las condiciones del canal en función de las pérdidas de paquetes, la tasa entrega, el rendimiento o la relación señal-interferencia y ruido (Signal to Interference and Noise Ratio - SINR). Conceptualmente, se espera que un enlace tenga un buen comportamiento para una tasa si el valor de SINR en el receptor esta por encima de un umbral definido en una tabla de adaptación. La selección de la tasa y el control de la potencia de transmisión tienen que estar relacionados, el control de la potencia sin considerar la tasa puede reducir el valor del SINR, y por tanto reducir la tasa y el rendimiento del enlace y la red.

Symphonies tiene en cuenta este problema y no compromete la tasa requerida. Por ejemplo, para transmitir a 54 Mbit/s, la potencia de transmisión puede ser reducida hasta que alcance el umbral SINR límite de 24.56 dB. Se necesita tener un valor preciso de la SINR recibida con el fin de ajustar el nivel de potencia de transmisión.

En los artículos investigados podemos encontrar como los algoritmos se basan en características no obligatorias de la norma 802.11n y que por tanto la mayoría de los dispositivos del mercado no lo soportan. También se ha visto como algunos algoritmos solo son efectivos si se instala un software adicional a los clientes que se encargue de recolectar y enviar al punto de acceso la información necesaria. En otros casos la topología de la red no es

la mas común, existen algoritmos que solo son efectivos en redes ad-hoc, siendo las redes infraestructura las populares.

Por lo tanto, desde aquí animo a cualquier persona que lea este proyecto, a utilizarlo como herramienta tanto de simulación como de documentación en el desarrollo y simulación de algoritmos que optimicen el rendimiento de una red inalámbrica de área local.

Anexos

A. Manual ns-3

A.1. Construcción del escenario para la norma 802.11n

La instalación de una interfaz de red se hace a través del método `Install` de la clase **`WifiHelper`**. El ejemplar se crea haciendo uso de la llamada al método `Default` de esta misma clase. Este método devuelve un ejemplar de esta clase con el gestor de estaciones remoto configurado para que `ArfWifiManager`[12] sea el algoritmo de control de la tasa. En el constructor de la clase `WifiHelper` se configura la norma en la que se basa el modelado de la capa PHY, la configuración por defecto utiliza la norma 802.11a. A continuación se presentan los tres parámetros que recibe el método `Install`:

- Un objeto de la clase **`WifiPhyHelper`**: sirve para crear objetos **`WifiPhy`**, éstos son los que modelan en *ns-3* la capa PHY.
- Un Objeto de la clase **`WifiMacHelper`**: de la misma manera sirve para crear objetos de la clase **`WifiMac`** que son los que modelan la capa MAC.
- El nodo en el que se quiere instalar la interfaz Wi-Fi.

El ejemplar de la clase **`WifiPhyHelper`** se crea con el método `Default` de la clase `YansWifiPhyHelper`, clase de ayuda que sirve para hacer fácil crear y gestionar objetos PHY para el modelo yans [10]. El ejemplar devuelto por este método está configurado para que el modelo de tasa de error sea `NistErrorRateModel`. El ejemplar de la clase `WifiPhyHelper` es necesario que tenga asignado un objeto de la clase `YansWifiChannelHelper`, este objeto se crea haciendo uso del método `Default` de esta misma clase. Este método devuelve un ejemplar de su clase configurado para que el retraso de propagación esté modelado por `ConstantSpeedPropagationDelayModel` y las pérdidas por propagación se modelen mediante `LogDistancePropagationLossModel`.

Para crear el ejemplar de la clase **`WifiMacHelper`** se utiliza el método `Default` de la clase `QosWifiMacHelper` que crea el ejemplar configurado para comportarse como una estación `StaWifiMac`, con soporte para calidad de servicio activado [`"QosSupported", BooleanValue (true)`].

Por último, para el nodo se crea una instancia de la clase **`NodeContainer`** y con ella utilizar el método `Create` con el número de nodos a crear como único parámetro.

La configuración del ejemplar de la clase `WifiMacHelper` se finaliza con el método `SetType`, con este metodo se configura el tipo de capa MAC que se usa, para el caso de un punto de acceso el tipo es `ApWifiMac` y para las estaciones o STA `StaWifiMac`. En

este mismo método se puede dar valor a los parámetros de la clase `WifiMac`, para este caso se usan los valores por defecto a excepción del Ssid que cambiamos a "Wi-Fi PFC" tanto para la estación como para el punto de acceso.

El código correspondiente se encuentra en el anexo B.1.

En resumen, la configuración por defecto que se hace a través de las clases `Helper` es la siguiente:

- Norma para el modelado de la capa PHY: 802.11a.
- Algoritmo de control de tasa: `ArfWifiManager`[12].
- Modelo de tasa de error: `NistErrorRateModel`[17].
- Modelo de retraso de propagación: `ConstantSpeedPropagationDelayModel`.
- Modelo de pérdidas de propagación: `LogDistancePropagationLossModel`.

Existe un amplio abanico de alternativas para cada una de estas configuraciones. La configuración que se trata en este apartado es la elección de la norma con la que se modela la capa PHY. Para ello no hay mas que usar el método `SetStandard` del ejemplar de la clase `WifiHelper`. Se configura la capa PHY para la norma 802.11n trabajando a 2.4 GHz:

```
SetStandard (WIFI_PHY_STANDARD_80211n_2_4GHZ);
```

La configuración para la norma 802.11n tiene un requerimiento en *ns-3*, que es habilitar el soporte para HT en la capa MAC de los dispositivos. Esto se consigue fácilmente sustituyendo el ejemplar de la clase `QosWifiMacHelper` por uno de la clase `HtWifiMacHelper`. El método `Default` de esta clase devuelve el ejemplar:

```
HtWifiMacHelper wifiMac = HtWifiMacHelper::Default ();
```

Con esto quedaría terminado el escenario para trabajar bajo la norma 802.11n según el manual de *ns-3*.

A.2. 802.11n - Modos de transmisión

El conjunto de modos de transmisión con los que un dispositivo puede trabajar bajo la norma 802.11n en *ns-3* es configurado por el método `Configure802.11n` de la clase `YansWifiPhy`:

```
void
YansWifiPhy::Configure80211n (void)
{
  NS_LOG_FUNCTION (this);

  m_deviceRateSet.push_back (WifiPhy::GetDsssRate1Mbps ());
  m_deviceRateSet.push_back (WifiPhy::GetDsssRate2Mbps ());
  m_deviceRateSet.push_back (WifiPhy::GetDsssRate5_5Mbps ());
  m_deviceRateSet.push_back (WifiPhy::GetErpOfdmRate6Mbps ());
  m_deviceRateSet.push_back (WifiPhy::GetDsssRate11Mbps ());
  m_deviceRateSet.push_back (WifiPhy::GetErpOfdmRate12Mbps ());
  m_deviceRateSet.push_back (WifiPhy::GetErpOfdmRate24Mbps ());
  m_bssMembershipSelectorSet.push_back(HT_PHY);

  for (uint8_t i=0; i < 8; i++)
    {
      m_deviceMcsSet.push_back(i);
    }
}
```

Esta configuración de modos soportados está hecha en base a las directrices que la IEEE establece en la norma 802.11n[2]. La norma dice que las tasas de datos que obligatoriamente deben soportar los dispositivos de la norma 802.11n son las mismas que las de los dispositivos 802.11g. En la siguiente tabla se recogen las tasas de datos obligatorias y opcionales de la norma 802.11g:

		802.11g @ 2.4 Ghz	
Tasa (Mbit/s)	**Portadora única/múltiple**	**Obligatorio**	**Opcional**
1	**única**	DSSS	
2	**única**	DSSS	
5.5	**única**	CCK	PBCC
6	**múltiple**	OFDM	CCK-ODFDM
9	**múltiple**		OFDM, CCK-OFDM
11	**única**	CCK	PBCC
12	**múltiple**	OFDM	CCK-ODFDM
18	**múltiple**		OFDM, CCK-OFDM
22	**única**		PBCC
24	**múltiple**	OFDM	CCK-ODFDM
33	**única**		PBCC
36	**múltiple**		OFDM, CCK-OFDM
48	**múltiple**		OFDM, CCK-OFDM
54	**múltiple**		OFDM, CCK-OFDM

Tabla A.1: Tasas de datos para la norma 802.11g

Además de las tasas de datos obligatorias que impone la norma también se añaden los 8 primeros índices MCS definidos en la norma 802.11n. En el código anterior podemos ver como se añaden elementos a tres conjuntos diferentes, a continuación se explica en que consiste cada uno de estos tres conjuntos:

- `m_deviceRateSet` es un ejemplar de la clase `WifiModeList`. Consiste en un vector de objetos de la clase `WifiMode`, clase que modela los diferentes modos de transmisión. Este vector se rellena con el conjunto de modos de transmisión de soporte obligatorio por la norma que corresponda.
- `m_bssMembershipSelectorSet` es un vector de enteros en el que se almacenan los índices de los conjuntos de modos de transmisión que los dispositivos soportan. En este caso la macro `HT_PHY` referencia el índice en el que se encuentra el conjunto de modos de transmisión soportados para la norma 802.11n. Esta lista de modos se puede obtener a través del método `GetMembershipSelectorModes` de la clase `YansWifiPhy`. Para ver cuales son estos modos podemos consultar el código fuente:

```
WifiModeList
YansWifiPhy::GetMembershipSelectorModes(uint32_t selector)
{
  uint32_t id=GetBssMembershipSelector(selector);
  WifiModeList supportedmodes;
  if (id == HT_PHY)
    {
      //mandatory MCS 0 to 7
      supportedmodes.push_back (WifiPhy::GetOfdmRate6_5MbpsBW20MHz ());
      supportedmodes.push_back (WifiPhy::GetOfdmRate13MbpsBW20MHz ());
      supportedmodes.push_back (WifiPhy::GetOfdmRate19_5MbpsBW20MHz ());
      supportedmodes.push_back (WifiPhy::GetOfdmRate26MbpsBW20MHz ());
      supportedmodes.push_back (WifiPhy::GetOfdmRate39MbpsBW20MHz ());
      supportedmodes.push_back (WifiPhy::GetOfdmRate52MbpsBW20MHz ());
      supportedmodes.push_back (WifiPhy::GetOfdmRate58_5MbpsBW20MHz ());
      supportedmodes.push_back (WifiPhy::GetOfdmRate65MbpsBW20MHz ());
    }
  return supportedmodes;
}
```

  Se trata, como bien dice el comentario del código fuente, de los modos de transmisión obligatorios para la norma 802.11n. Son los modos correspondientes a los índices MCS del 0 al 7 para 20 MHz de canal e intervalo de guarda largo (800 ns).
- Por último, `m_deviceMcsSet` es un vector de enteros en el que se guardan los índices MCS soportados por el dispositivo. Este vector está pensado para trabajar con el método `McsToWifiMode` de la clase `YansWifiPhy`, este método recibe como parámetro el índice MCS y devuelve el modo de transmisión correspondiente, dependiendo del canal (20 o 40 MHz) y del intervalo de guarda (corto o largo). Es decir, es un selector del modo de operación dentro de la tabla 2.2.

Una vez conocidos estos conjuntos y los valores que toman para la norma 802.11n es momento de saber como son utilizados. La clase `WifiRemoteStationManager` es la encargada de la gestión de estos conjuntos de modos de transmisión. Nos centramos únicamente en los métodos que gestionan los modos de transmisión. `SetupPhy` es el primer método en el que hay que detenerse puesto que es el método en el que se inicializa el modo de transmisión por defecto. El modo de transmisión por defecto toma aquí el valor

correspondiente al modo alojado en el primer índice del conjunto `m_deviceRateSet`, que en este caso es el modo `DsssRate1Mbps`.

Para obtener el modo con el que se transmiten las tramas de control se recurre al método `GetControlAnswerMode`. Es un método extenso y programado de manera deficiente puesto que existen líneas de código que nunca llegan a ser ejecutadas, los desarrolladores de este método han intentado seguir las directrices de la norma para este aspecto pero la ambigüedad de la misma ha hecho que el modelado resultante sea caótico. Tras el análisis de este método se concluye que el modo utilizado para transmitir las tramas de control es el modo por defecto, (`DsssRate1Mbps`).

Para finalizar con los métodos de interés de la clase `WifiRemoteStationManager`, examinamos el método `GetSupported`. Este método recibe como parámetro el índice del elemento deseado en el conjunto de modos de transmisión de `m_operationalRateSet`. Este conjunto contiene la lista de modos de transmisión devuelto por el método `GetMembershipSelectorModes` visto anteriormente.

A.3. Proceso de recepción de paquetes

La recepción de un paquete se modela en el archivo `yans-wifi-phy.cc`, el método `StartReceivePacket` de la clase `YansWifiPhy` es el primer paso. Se pueden dar dos casos por los que se tire el paquete: que la capa PHY se encuentre en el proceso de transmisión/recepción de un paquete o que se esté cambiando de canal. En caso de que el estado de la capa PHY sea ocioso, se continúa con la recepción del paquete.

En este punto es donde se realiza la comprobación anteriormente comentada: verificar que la potencia con la que llega el paquete está por encima del umbral de detección de una señal. El paquete se tira en caso de que la potencia de la señal recibida sea demasiado pequeña. Si por el contrario la potencia es suficiente, se cambia el estado del canal para indicar que se está recibiendo un paquete. Luego se encola el evento de fin de recepción del paquete para que sea lanzado transcurrido el tiempo que tarda el paquete en ser recibido.

El evento de fin de recepción del paquete está asociado al método `EndReceive` de la misma clase `YansWifiPhy`. En este método se calcula la tasa de error de paquete para después compararla con una variable aleatoria y así decidir si el paquete termina de ser recibido correctamente o si por el contrario se tira por error de paquete. En la clase `YansWifiPhy`, el atributo privado `m_interference` es un puntero a un ejemplar de la clase `InterferenceHelper`. Esta clase se ha creado para servir de ayuda a los cálculos relacionados con las interferencias que se puedan dar en una transmisión inalámbrica. A través de este ejemplar se accede al método `CalculateSnrPer` que devuelve una estructura con dos variables tipo `double`, una para la tasa de error de paquete y la otra para la tasa señal-ruido.

No entraremos en detalles de como se calculan estas dos tasas, pero lo que si se va a analizar es la relación entre el método `CalculateSnrPer` y la clase que modela la tasa de error en `ns-3`:

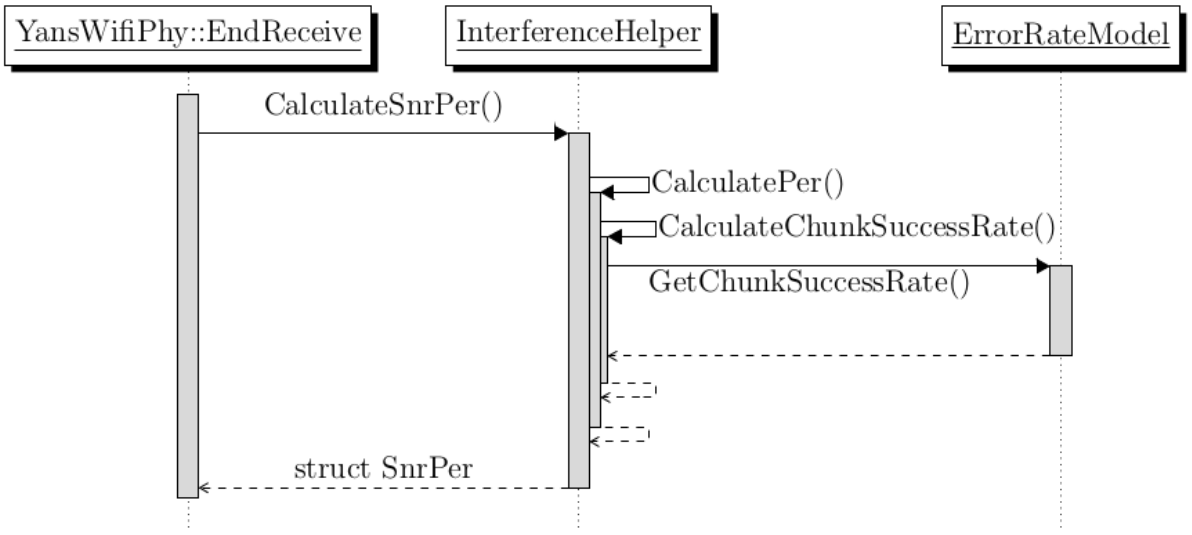


Figura A.1: Diagrama de secuencia para el cálculo de PER

El diagrama se ha simplificado para una mejor comprensión, se han omitido los parámetros de cada método y algunos valores de retorno. El diagrama comienza con la llamada

al método `CalculateSnrPer` de la clase `InterferenceHelper` desde el método `EndReceive` de la clase `YansWifiPhy`. Dentro de este método se hacen dos llamadas a métodos de la misma clase, `CalculateSnr` y `CalculatePer`. Dado que es el cálculo de PER es el objeto de nuestro interés, continuamos el diagrama de secuencia siguiendo la llamada a este método.

El método `CalculatePer` contempla un gran número de variables y casos para el cálculo de PER. Cualquiera que sea el caso, el cálculo de la PER está condicionado al valor devuelto por el método `CalculateChunkSuccessRate` de la misma clase. Dentro de este método se hace la llamada al método virtual puro `GetChunkSuccessRate` de la clase `ErrorRateModel`. Este método tiene que ser implementado por una subclase de `ErrorRateModel` y devuelve la probabilidad de que sean recibidos correctamente los bits con igual SNR de un paquete.

Existen dos subclases de la clase `ErrorRateModel`:

- `YansErrorRateModel` se encarga de modelar la tasa de error de las modulaciones de la norma 802.11b.
- `NistErrorRateModel` es el modelo de tasa de error por defecto en ns-3. Además de modelar la tasa de error de las modulaciones de la norma 802.11b, también modela dicho cálculo para las modulaciones OFDM.

La clase `NistErrorRateModel`[17] está diseñada para modelar el cálculo de la tasa de error de las modulaciones OFDM de la norma 802.11g. A pesar de eso también soporta el cálculo de esta tasa para la mayoría de los modos de transmisión de la norma 802.11n. En este modelo se quedan sin contemplar las modulaciones OFDM con tasa de codificación de 5/6. En este proyecto no se ha abordado la solución de esta deficiencia de `ns-3` porque es conocida por la comunidad de desarrolladores de `ns-3`. El fallo está registrado y solucionado a la espera de la aprobación de la comunidad [Bugzilla - Bug 1758: Missing Yans and Nist error rate models for 5/6 code rate of 802.11n HT].

Código

B. Escenarios

```
// ##########################
NS_LOG_INFO ("Crear nodos");
// ##########################

// wifiApNode
NodeContainer wifiApNodes;
wifiApNodes.Create(nAp);
// StaNodes
NodeContainer wifiStaNodes;
wifiStaNodes.Create(nSta);

WifiHelper wifi = WifiHelper::Default ();

// ############################
NS_LOG_INFO ("Crear canal");
// ############################

/** PHY **/
YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ();
YansWifiChannelHelper channel = YansWifiChannelHelper::Default ();
wifiPhy.SetChannel (channel.Create());

/** MAC **/
QosWifiMacHelper wifiMac = QosWifiMacHelper::Default ();

// ##############################
NS_LOG_INFO ("Crear dispositivos");
// ##############################

NetDeviceContainer apDevice;
wifiMac.SetType ("ns3::ApWifiMac",
    "Ssid", SsidValue (Ssid ("Wi-Fi PFC")));
apDevice = wifi.Install (wifiPhy, wifiMac, wifiApNodes);

NetDeviceContainer staDevice;
wifiMac.SetType ("ns3::StaWifiMac",
    "Ssid", SsidValue (Ssid ("Wi-Fi PFC")));
staDevice = wifi.Install (wifiPhy, wifiMac, wifiStaNodes);
```

Código B.1: 2.1.1. Escenario Inicial - Crear dispositivos

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/applications-module.h"
#include "ns3/wifi-module.h"
#include "ns3/mobility-module.h"
#include "ns3/internet-module.h"
#include "ns3/propagation-delay-model.h"
#include "ns3/propagation-loss-model.h"

using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("Scenario");

bool getGI(WifiMode m)
{
  if (m.GetUniqueName() == "OfdmRate135MbpsBW40MHzShGi" || m.GetUniqueName() == "↩
      OfdmRate65MbpsBW20MHzShGi" ){
      return true;
  }else if(m.GetUniqueName() == "OfdmRate135MbpsBW40MHz" || m.GetUniqueName() == "↩
      OfdmRate65MbpsBW20MHz" ){
      return false;
  }
  switch (m.GetDataRate()) {
    case 6500000:
    case 13500000:
    case 13000000:
    case 27000000:
    case 19500000:
    case 40500000:
    case 26000000:
    case 54000000:
    case 39000000:
    case 81000000:
    case 52000000:
    case 108000000:
    case 58500000:
    case 121500000:
      return false;
      break;
    case 7200000:
    case 15000000:
    case 14400000:
    case 30000000:
    case 21700000:
    case 45000000:
    case 28900000:
    case 60000000:
    case 43300000:
    case 90000000:
    case 57800000:
    case 120000000:
    case 72200000:
    case 150000000:
      return true;
      break;
  }
  return false;
}

uint32_t PhyRxBeginMcs0, PhyRxBeginMcs1, PhyRxBeginMcs2, PhyRxBeginMcs3, PhyRxBeginMcs4, ↩
    PhyRxBeginMcs5, PhyRxBeginMcs6, PhyRxBeginMcs7;
uint32_t beginsimplechannel, begindoublechannel, beginlongGI, beginshortGI;

void
PhyRxBeginMode(std::string path, Ptr<const Packet> p, WifiMode m)
{
  Ptr<Packet> copyp = p->Copy();
  WifiMacHeader hdr;
  copyp->RemoveHeader (hdr);
```

```
  if(!hdr.IsAck())
  {
    if(m.GetBandwidth() == 20000000){
      beginsimplechannel++;
    }else {
      begindoublechannel++;
    }
    getGI(m) ? beginshortGI++ : beginlongGI++;
  }

  if(hdr.IsData())
  {
    YansWifiPhy yans_wifi;
    uint32_t mcs = yans_wifi.WifiModeToMcs(m);
    switch (mcs) {
      case 0:
        PhyRxBeginMcs0++;
        break;
      case 1:
        PhyRxBeginMcs1++;
        break;
      case 2:
        PhyRxBeginMcs2++;
        break;
      case 3:
        PhyRxBeginMcs3++;
        break;
      case 4:
        PhyRxBeginMcs4++;
        break;
      case 5:
        PhyRxBeginMcs5++;
        break;
      case 6:
        PhyRxBeginMcs6++;
        break;
      case 7:
        PhyRxBeginMcs7++;
        break;
      default:
        break;
    }
  }
}

uint32_t PhyRxEndMcs0, PhyRxEndMcs1, PhyRxEndMcs2, PhyRxEndMcs3, PhyRxEndMcs4, PhyRxEndMcs5,↩
     PhyRxEndMcs6, PhyRxEndMcs7;
uint32_t endsimplechannel, enddoublechannel, endlongGI, endshortGI;

void
PhyRxEndMode(std::string path, Ptr<const Packet> p, WifiMode m)
{
  Ptr<Packet> copyp = p->Copy();
  WifiMacHeader hdr;
  copyp->RemoveHeader (hdr);

  if(!hdr.IsAck())
  {
    if(m.GetBandwidth() == 20000000){
      endsimplechannel++;
    }else {
      enddoublechannel++;
    }
    getGI(m) ? endshortGI++ : endlongGI++;
  }

  if(hdr.IsData())
  {
    YansWifiPhy yans_wifi;
    uint32_t mcs = yans_wifi.WifiModeToMcs(m);
```

```
    switch (mcs) {
      case 0:
        PhyRxEndMcs0++;
        break;
      case 1:
        PhyRxEndMcs1++;
        break;
      case 2:
        PhyRxEndMcs2++;
        break;
      case 3:
        PhyRxEndMcs3++;
        break;
      case 4:
        PhyRxEndMcs4++;
        break;
      case 5:
        PhyRxEndMcs5++;
        break;
      case 6:
        PhyRxEndMcs6++;
        break;
      case 7:
        PhyRxEndMcs7++;
        break;
      default:
        break;
    }
  }
}

uint32_t PhyRxDropMcs0, PhyRxDropMcs1, PhyRxDropMcs2, PhyRxDropMcs3, PhyRxDropMcs4, ↩
    PhyRxDropMcs5, PhyRxDropMcs6, PhyRxDropMcs7;
uint32_t dropsimplechannel, dropdoublechannel, droplongGI, dropshortGI;

void
PhyRxDropMode(std::string path, Ptr<const Packet> p, WifiMode m)
{
  Ptr<Packet> copyp = p->Copy();
  WifiMacHeader hdr;
  copyp->RemoveHeader (hdr);

  if(!hdr.IsAck())
  {
    if(m.GetBandwidth() == 20000000){
      dropsimplechannel++;
    }else {
      dropdoublechannel++;
    }
    getGI(m) ? dropshortGI++ : droplongGI++;
  }

  if(hdr.IsData())
  {
    YansWifiPhy yans_wifi;
    uint32_t mcs = yans_wifi.WifiModeToMcs(m);
    switch (mcs) {
      case 0:
        PhyRxDropMcs0++;
        break;
      case 1:
        PhyRxDropMcs1++;
        break;
      case 2:
        PhyRxDropMcs2++;
        break;
      case 3:
        PhyRxDropMcs3++;
        break;
      case 4:
```

```
        PhyRxDropMcs4++;
        break;
      case 5:
        PhyRxDropMcs5++;
        break;
      case 6:
        PhyRxDropMcs6++;
        break;
      case 7:
        PhyRxDropMcs7++;
        break;
      default:
        break;
    }
  }
}

void
Print()
{
  NS_LOG_INFO ("--------------------------------");
  NS_LOG_INFO ("\tInicio\tFinal\tTirados");
  NS_LOG_INFO ("MCS 0\t" << PhyRxBeginMcs0 << "\t" << PhyRxEndMcs0 << "\t" << PhyRxDropMcs0 ↩
      << "\tpaquetes");
  NS_LOG_INFO ("MCS 1\t" << PhyRxBeginMcs1 << "\t" << PhyRxEndMcs1 << "\t" << PhyRxDropMcs1 ↩
      << "\tpaquetes");
  NS_LOG_INFO ("MCS 2\t" << PhyRxBeginMcs2 << "\t" << PhyRxEndMcs2 << "\t" << PhyRxDropMcs2 ↩
      << "\tpaquetes");
  NS_LOG_INFO ("MCS 3\t" << PhyRxBeginMcs3 << "\t" << PhyRxEndMcs3 << "\t" << PhyRxDropMcs3 ↩
      << "\tpaquetes");
  NS_LOG_INFO ("MCS 4\t" << PhyRxBeginMcs4 << "\t" << PhyRxEndMcs4 << "\t" << PhyRxDropMcs4 ↩
      << "\tpaquetes");
  NS_LOG_INFO ("MCS 5\t" << PhyRxBeginMcs5 << "\t" << PhyRxEndMcs5 << "\t" << PhyRxDropMcs5 ↩
      << "\tpaquetes");
  NS_LOG_INFO ("MCS 6\t" << PhyRxBeginMcs6 << "\t" << PhyRxEndMcs6 << "\t" << PhyRxDropMcs6 ↩
      << "\tpaquetes");
  NS_LOG_INFO ("MCS 7\t" << PhyRxBeginMcs7 << "\t" << PhyRxEndMcs7 << "\t" << PhyRxDropMcs7 ↩
      << "\tpaquetes");
  NS_LOG_INFO ("--------------------------------");
  NS_LOG_INFO ("20 MHz\t" << beginsimplechannel << "\t" << endsimplechannel << "\t" << ↩
      dropsimplechannel << "\tpaquetes");
  NS_LOG_INFO ("40 MHz\t" << begindoublechannel << "\t" << enddoublechannel << "\t" << ↩
      dropdoublechannel << "\tpaquetes");
  NS_LOG_INFO ("LGI\t" << beginlongGI << "\t"  << endlongGI << "\t" << droplongGI << "\↩
      tpaquetes");
  NS_LOG_INFO ("SGI\t" << beginshortGI << "\t" << endshortGI << "\t" << dropshortGI << "\↩
      tpaquetes");
  NS_LOG_INFO ("--------------------------------");
}

int main (int argc, char *argv[])
{
  LogComponentEnable ("Scenario", LOG_LEVEL_INFO);

  uint32_t simtime = 5;
  uint32_t seed = 1;
  uint32_t nAp = 1;
  uint32_t nSta = 1;
  bool ch = false;
  bool sge = false;

  CommandLine cmd;
  cmd.AddValue ("simtime", "Simulation time (seconds)", simtime);
  cmd.AddValue ("seed", "Seed for multiple simulations", seed);
  cmd.AddValue ("nAp", "Number of APs", nAp);
  cmd.AddValue ("nSta", "Number of STAs", nSta);
  cmd.AddValue ("ChannelBonding", "ChannelBonding", ch);
  cmd.AddValue ("ShortGuardEnabled", "ShortGuardEnabled", sge);
  cmd.Parse (argc,argv);
```

```
NS_LOG_INFO ("SimTime: " << simtime << "\tseed: " << seed << "\tch: " << ch << "\tsgr: " ↩
    << sge);

//Set the run number to generate a different random stream
SeedManager::SetRun(seed);

// ########################
//NS_LOG_INFO ("Crear nodos");
// ########################

// wifiApNode
NodeContainer wifiApNodes;
wifiApNodes.Create(nAp);
// StaNodes
NodeContainer wifiStaNodes;
wifiStaNodes.Create(nSta);

WifiHelper wifi = WifiHelper::Default ();
wifi.SetStandard (WIFI_PHY_STANDARD_80211n_2_4GHZ);

// ############################
//NS_LOG_INFO ("Crear canal");
// ############################

/** PHY **/
YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ();
YansWifiChannelHelper channel = YansWifiChannelHelper::Default ();
wifiPhy.SetChannel (channel.Create());

wifiPhy.Set ("GreenfieldEnabled", BooleanValue(true));

wifiPhy.Set ("ShortGuardEnabled", BooleanValue(sge));
wifiPhy.Set ("ChannelBonding", BooleanValue(ch));

/** MAC **/
HtWifiMacHelper wifiMac = HtWifiMacHelper::Default ();

// ##############################
//NS_LOG_INFO ("Crear dispositivos");
// ##############################

NetDeviceContainer apDevice;
wifiMac.SetType ("ns3::ApWifiMac",
    "Ssid", SsidValue (Ssid ("Wi-Fi PFC")));
apDevice = wifi.Install (wifiPhy, wifiMac, wifiApNodes);

NetDeviceContainer staDevice;
wifiMac.SetType ("ns3::StaWifiMac",
    "Ssid", SsidValue (Ssid ("Wi-Fi PFC")));
staDevice = wifi.Install (wifiPhy, wifiMac, wifiStaNodes);

Ptr<ListPositionAllocator> allocator = CreateObject<ListPositionAllocator> ();
allocator->Add(Vector (0.0, 0.0, 0.0));    // AP
allocator->Add(Vector (16.0, 16.0, 0.0));  // STA

MobilityHelper mobility;
mobility.SetPositionAllocator(allocator);
mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");
mobility.Install (wifiApNodes);
mobility.Install (wifiStaNodes);

// Pilas de protocolos
InternetStackHelper stack;
stack.Install (wifiApNodes);
stack.Install (wifiStaNodes);

// ###################################
//NS_LOG_INFO ("Asignar direcciones IP");
// ###################################
```

```
  Ipv4AddressHelper address;
  // Para los nodos STA y nodo AP
  address.SetBase ("192.168.1.0", "255.255.255.0");
  Ipv4InterfaceContainer apInterfaces = address.Assign (apDevice);
  Ipv4InterfaceContainer staInterfaces = address.Assign (staDevice);

  // ################################
  //NS_LOG_INFO ("Crear trafico");
  // ################################
  Ptr<Ipv4> ipv4Server = staDevice.Get(0)->GetNode()->GetObject<Ipv4>();
  Ipv4InterfaceAddress iaddrServer = ipv4Server->GetAddress(1,0);
  Ipv4Address ipv4AddrServer = iaddrServer.GetLocal ();

  OnOffHelper onoff ("ns3::UdpSocketFactory",
      Address(InetSocketAddress(ipv4AddrServer,5000)));
  onoff.SetAttribute ("OnTime",
      StringValue ("ns3::ConstantRandomVariable[Constant=1]"));
  onoff.SetAttribute ("OffTime",
      StringValue ("ns3::ConstantRandomVariable[Constant=0]"));
  onoff.SetAttribute("DataRate", StringValue("20Mbps"));
  onoff.SetAttribute ("PacketSize", UintegerValue (1024));

  ApplicationContainer apps;
  apps.Add(onoff.Install (wifiApNodes.Get (0)));
  apps.Start (Seconds (1.5));
  apps.Stop (Seconds (simtime-1));

  ApplicationContainer sinkApp;
  PacketSinkHelper sinkHelper ("ns3::UdpSocketFactory",
      Address(InetSocketAddress(ipv4AddrServer,5000)));
  sinkApp = sinkHelper.Install (wifiStaNodes.Get (0));
  sinkApp.Start (Seconds (1));
  sinkApp.Stop (Seconds (simtime));
  //*******************************************************************
  Ipv4GlobalRoutingHelper::PopulateRoutingTables ();

  Config::Connect("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxBeginMode", ↩
      MakeCallback(&PhyRxBeginMode));
  Config::Connect("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxEndMode", ↩
      MakeCallback(&PhyRxEndMode));
  Config::Connect("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDropMode", ↩
      MakeCallback(&PhyRxDropMode));

  // ################################
  //NS_LOG_INFO ("Iniciar Simulacion");
  // ################################
  Simulator::Stop (Seconds (simtime));

  Simulator::Run ();

  Print();
  Simulator::Destroy ();

  return 0;
}
```

Código B.2: 2.1.1. Escenario verificador

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/applications-module.h"
#include "ns3/wifi-module.h"
#include "ns3/mobility-module.h"
#include "ns3/internet-module.h"
#include "ns3/propagation-delay-model.h"
#include "ns3/propagation-loss-model.h"

using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("Scenario");

double powerToDist(double power)
{
  Ptr<YansWifiPhy> yans_wifi = CreateObject <YansWifiPhy> ();
  double txPower = yans_wifi->GetTxPowerStart();
  Ptr<LogDistancePropagationLossModel> propagation = CreateObject <↩
      LogDistancePropagationLossModel> ();
  DoubleValue tmp;
  propagation->GetAttribute("ReferenceLoss", tmp);
  double ReferenceLoss = tmp.Get();
  double exponent = propagation->GetPathLossExponent ();

  return std::pow(10, (power-2-txPower+ReferenceLoss)/(-exponent*10) );
}

uint32_t PhyRxEndMcs0, PhyRxEndMcs1, PhyRxEndMcs2, PhyRxEndMcs3,
         PhyRxEndMcs4, PhyRxEndMcs5, PhyRxEndMcs6, PhyRxEndMcs7;
uint32_t endsimplechannel, enddoublechannel;

void
PhyRxEndMode(std::string path, Ptr<const Packet> p, WifiMode m)
{
  Ptr<Packet> copyp = p->Copy();
  WifiMacHeader hdr;
  copyp->RemoveHeader (hdr);

  if(hdr.IsData())
  {
    if(m.GetBandwidth() == 20000000){
      endsimplechannel++;
    }else {
      enddoublechannel++;
    }
    YansWifiPhy yans_wifi;
    uint32_t mcs = yans_wifi.WifiModeToMcs(m);
    switch (mcs) {
      case 0:
        PhyRxEndMcs0++;
        break;
      case 1:
        PhyRxEndMcs1++;
        break;
      case 2:
        PhyRxEndMcs2++;
        break;
      case 3:
        PhyRxEndMcs3++;
        break;
      case 4:
        PhyRxEndMcs4++;
        break;
      case 5:
        PhyRxEndMcs5++;
        break;
      case 6:
        PhyRxEndMcs6++;
        break;
      case 7:
```

```
        PhyRxEndMcs7++;
        break;
      default:
        break;
    }
  }
}

uint32_t PhyRxDropMcs0, PhyRxDropMcs1, PhyRxDropMcs2, PhyRxDropMcs3,
         PhyRxDropMcs4, PhyRxDropMcs5, PhyRxDropMcs6, PhyRxDropMcs7;
uint32_t dropsimplechannel, dropdoublechannel;

void
PhyRxDropMode(std::string path, Ptr<const Packet> p, WifiMode m)
{
  Ptr<Packet> copyp = p->Copy();
  WifiMacHeader hdr;
  copyp->RemoveHeader (hdr);

  if(hdr.IsData())
  {
    if(m.GetBandwidth() == 20000000){
      dropsimplechannel++;
    }else {
      dropdoublechannel++;
    }
    YansWifiPhy yans_wifi;
    uint32_t mcs = yans_wifi.WifiModeToMcs(m);
    switch (mcs) {
      case 0:
        PhyRxDropMcs0++;
        break;
      case 1:
        PhyRxDropMcs1++;
        break;
      case 2:
        PhyRxDropMcs2++;
        break;
      case 3:
        PhyRxDropMcs3++;
        break;
      case 4:
        PhyRxDropMcs4++;
        break;
      case 5:
        PhyRxDropMcs5++;
        break;
      case 6:
        PhyRxDropMcs6++;
        break;
      case 7:
        PhyRxDropMcs7++;
        break;
      default:
        break;
    }
  }
}

uint32_t PhyRxBeginMcs0, PhyRxBeginMcs1, PhyRxBeginMcs2, PhyRxBeginMcs3,
         PhyRxBeginMcs4, PhyRxBeginMcs5, PhyRxBeginMcs6, PhyRxBeginMcs7;
uint32_t beginsimplechannel, begindoublechannel;

double last_powerMcs0, last_powerMcs1, last_powerMcs2, last_powerMcs3,
       last_powerMcs4, last_powerMcs5, last_powerMcs6, last_powerMcs7;

void
PhyRxBeginModePower(std::string path, Ptr<const Packet> p, WifiMode m, double dbmpower)
{
  Ptr<Packet> copyp = p->Copy();
```

```
  WifiMacHeader hdr;
  copyp->RemoveHeader (hdr);

  if(hdr.IsData())
  {
    if(m.GetBandwidth() == 20000000){
      beginsimplechannel++;
    }else {
      begindoublechannel++;
    }

    YansWifiPhy yans_wifi;
    uint32_t mcs = yans_wifi.WifiModeToMcs(m);
    switch (mcs) {
      case 0:
        PhyRxBeginMcs0++;
        break;
      case 1:
        PhyRxBeginMcs1++;
        break;
      case 2:
        PhyRxBeginMcs2++;
        break;
      case 3:
        PhyRxBeginMcs3++;
        break;
      case 4:
        PhyRxBeginMcs4++;
        break;
      case 5:
        PhyRxBeginMcs5++;
        break;
      case 6:
        PhyRxBeginMcs6++;
        break;
      case 7:
        PhyRxBeginMcs7++;
        break;
      default:
        break;
    }
  }
}

uint32_t PhyRxDropPowerMcs0, PhyRxDropPowerMcs1, PhyRxDropPowerMcs2, PhyRxDropPowerMcs3,
         PhyRxDropPowerMcs4, PhyRxDropPowerMcs5, PhyRxDropPowerMcs6, PhyRxDropPowerMcs7;
uint32_t droppowersimplechannel, droppowerdoublechannel;

void
NotifyRxDropModePower(std::string path, Ptr<const Packet> p, WifiMode m, double dbmpower)
{
  Ptr<Packet> copyp = p->Copy();
  WifiMacHeader hdr;
  copyp->RemoveHeader (hdr);

  if(hdr.IsData())
  {
    if(m.GetBandwidth() == 20000000){
      droppowersimplechannel++;
    }else {
      droppowerdoublechannel++;
    }
    YansWifiPhy yans_wifi;
    uint32_t mcs = yans_wifi.WifiModeToMcs(m);
    switch (mcs) {
      case 0:
        PhyRxDropPowerMcs0++;
        last_powerMcs0 = dbmpower;
        break;
      case 1:
```

```
        PhyRxDropPowerMcs1++;
        last_powerMcs1 = dbmpower;
        break;
      case 2:
        PhyRxDropPowerMcs2++;
        last_powerMcs2 = dbmpower;
        break;
      case 3:
        PhyRxDropPowerMcs3++;
        last_powerMcs3 = dbmpower;
        break;
      case 4:
        PhyRxDropPowerMcs4++;
        last_powerMcs4 = dbmpower;
        break;
      case 5:
        PhyRxDropPowerMcs5++;
        last_powerMcs5 = dbmpower;
        break;
      case 6:
        PhyRxDropPowerMcs6++;
        last_powerMcs6 = dbmpower;
        break;
      case 7:
        PhyRxDropPowerMcs7++;
        last_powerMcs7 = dbmpower;
        break;
      default:
        break;
    }

  }
}

void
Print()
{
  NS_LOG_INFO ("-----------------------------------------");
  NS_LOG_INFO ("\tInicio\tFinal\tTirados\tTirados");
  NS_LOG_INFO ("\t\t\t(Sens)\t(PER)");

  NS_LOG_INFO ("MCS 0\t" << PhyRxBeginMcs0 << "\t" << PhyRxEndMcs0 << "\t"
                         << PhyRxDropPowerMcs0 << "\t" << PhyRxDropMcs0 << "\tpaquetes");
  NS_LOG_INFO ("MCS 1\t" << PhyRxBeginMcs1 << "\t" << PhyRxEndMcs1 << "\t"
                         << PhyRxDropPowerMcs1 << "\t" << PhyRxDropMcs1 << "\tpaquetes");
  NS_LOG_INFO ("MCS 2\t" << PhyRxBeginMcs2 << "\t" << PhyRxEndMcs2 << "\t"
                         << PhyRxDropPowerMcs2 << "\t" << PhyRxDropMcs2 << "\tpaquetes");
  NS_LOG_INFO ("MCS 3\t" << PhyRxBeginMcs3 << "\t" << PhyRxEndMcs3 << "\t"
                         << PhyRxDropPowerMcs3 << "\t" << PhyRxDropMcs3 << "\tpaquetes");
  NS_LOG_INFO ("MCS 4\t" << PhyRxBeginMcs4 << "\t" << PhyRxEndMcs4 << "\t"
                         << PhyRxDropPowerMcs4 << "\t" << PhyRxDropMcs4 << "\tpaquetes");
  NS_LOG_INFO ("MCS 5\t" << PhyRxBeginMcs5 << "\t" << PhyRxEndMcs5 << "\t"
                         << PhyRxDropPowerMcs5 << "\t" << PhyRxDropMcs5 << "\tpaquetes");
  NS_LOG_INFO ("MCS 6\t" << PhyRxBeginMcs6 << "\t" << PhyRxEndMcs6 << "\t"
                         << PhyRxDropPowerMcs6 << "\t" << PhyRxDropMcs6 << "\tpaquetes");
  NS_LOG_INFO ("MCS 7\t" << PhyRxBeginMcs7 << "\t" << PhyRxEndMcs7 << "\t"
                         << PhyRxDropPowerMcs7 << "\t" << PhyRxDropMcs7 << "\tpaquetes");
  NS_LOG_INFO ("-----------------------------------------");
  NS_LOG_INFO ("20 MHz\t" << beginsimplechannel << "\t" << endsimplechannel << "\t"
                          << droppowersimplechannel << "\t" << dropsimplechannel << "\↩
                              tpaquetes");
  NS_LOG_INFO ("40 MHz\t" << begindoublechannel << "\t" << enddoublechannel << "\t"
                          << droppowerdoublechannel << "\t" << dropdoublechannel << "\↩
                              tpaquetes");
  NS_LOG_INFO ("-----------------------------------------");
}
```

```
int main (int argc, char *argv[])
{
  LogComponentEnable ("Scenario", LOG_LEVEL_INFO);

  uint32_t simtime = 5;
  uint32_t seed = 1;
  uint32_t nAp = 1;
  uint32_t nSta = 1;
  double dist = 10;
  double powerlimit = -64;
  uint32_t extraSens = 0;
  bool ch = false;
  bool sge = false;

  CommandLine cmd;
  cmd.AddValue ("simtime", "Simulation time (seconds)", simtime);
  cmd.AddValue ("seed", "Seed for multiple simulations", seed);
  cmd.AddValue ("nAp", "Number of APs", nAp);
  cmd.AddValue ("nSta", "Number of STAs", nSta);
  cmd.AddValue ("ChannelBonding", "ChannelBonding", ch);
  cmd.AddValue ("ShortGuardEnabled", "ShortGuardEnabled", sge);
  cmd.AddValue ("distance", "Distance between AP and STA (meters)", dist);
  cmd.AddValue ("powerlimit", "Potencia que recibe el STA", powerlimit);
  cmd.AddValue ("ExtraSens", "Extra sensitivity hardware specifications", extraSens);
  cmd.Parse (argc,argv);

  dist = powerToDist(powerlimit);
  NS_LOG_INFO ("SimTime: " << simtime << "\tseed: " << seed << "\tRxPower(distance): "
                                                          << powerlimit
                                                          << " dBm (" << dist << " m)"
                                                          << "\tExtraSens: " << extraSens
                                                          << "\tch: " << ch);

  //Set the run number to generate a different random stream
  SeedManager::SetRun(seed);

  // ##########################
  //NS_LOG_INFO ("Crear nodos");
  // ##########################

  // wifiApNode
  NodeContainer wifiApNodes;
  wifiApNodes.Create(nAp);
  // StaNodes
  NodeContainer wifiStaNodes;
  wifiStaNodes.Create(nSta);

  WifiHelper wifi = WifiHelper::Default ();
  wifi.SetStandard (WIFI_PHY_STANDARD_80211n_2_4GHZ);

  // ############################
  //NS_LOG_INFO ("Crear canal");
  // ############################

  /** PHY **/
  YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ();
  YansWifiChannelHelper channel = YansWifiChannelHelper::Default ();
  wifiPhy.SetChannel (channel.Create());

  wifiPhy.Set ("GreenfieldEnabled", BooleanValue(true));

  wifiPhy.Set ("ShortGuardEnabled", BooleanValue(sge));
  wifiPhy.Set ("ChannelBonding", BooleanValue(ch));

  wifiPhy.Set ("ExtraSens", UintegerValue(extraSens));

  /** MAC **/
  HtWifiMacHelper wifiMac = HtWifiMacHelper::Default ();
```

```
// ###############################
//NS_LOG_INFO ("Crear dispositivos");
// ###############################

NetDeviceContainer apDevice;
wifiMac.SetType ("ns3::ApWifiMac",
    "Ssid", SsidValue (Ssid ("Wi-Fi PFC")));
apDevice = wifi.Install (wifiPhy, wifiMac, wifiApNodes);

NetDeviceContainer staDevice;
wifiMac.SetType ("ns3::StaWifiMac",
    "Ssid", SsidValue (Ssid ("Wi-Fi PFC")));
staDevice = wifi.Install (wifiPhy, wifiMac, wifiStaNodes);

Ptr<ListPositionAllocator> allocator = CreateObject<ListPositionAllocator> ();
allocator->Add(Vector (0.0, 0.0, 0.0));   // AP
allocator->Add(Vector (dist, 0.0, 0.0));  // STA

MobilityHelper mobility;
mobility.SetPositionAllocator(allocator);
mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");
mobility.Install (wifiApNodes);
mobility.Install (wifiStaNodes);

// Pilas de protocolos
InternetStackHelper stack;
stack.Install (wifiApNodes);
stack.Install (wifiStaNodes);

// ###################################
//NS_LOG_INFO ("Asignar direcciones IP");
// ###################################

Ipv4AddressHelper address;
// Para los nodos STA y nodo AP
address.SetBase ("192.168.1.0", "255.255.255.0");
Ipv4InterfaceContainer apInterfaces = address.Assign (apDevice);
Ipv4InterfaceContainer staInterfaces = address.Assign (staDevice);

// ###############################
//NS_LOG_INFO ("Crear trafico");
// ###############################
Ptr<Ipv4> ipv4Server = staDevice.Get(0)->GetNode()->GetObject<Ipv4>();
Ipv4InterfaceAddress iaddrServer = ipv4Server->GetAddress(1,0);
Ipv4Address ipv4AddrServer = iaddrServer.GetLocal ();

OnOffHelper onoff ("ns3::UdpSocketFactory",
    Address(InetSocketAddress(ipv4AddrServer,5000)));
onoff.SetAttribute ("OnTime",
    StringValue ("ns3::ConstantRandomVariable[Constant=1]"));
onoff.SetAttribute ("OffTime",
    StringValue ("ns3::ConstantRandomVariable[Constant=0]"));
onoff.SetAttribute("DataRate", StringValue("20Mbps"));
onoff.SetAttribute ("PacketSize", UintegerValue (1024));

ApplicationContainer apps;
apps.Add(onoff.Install (wifiApNodes.Get (0)));
apps.Start (Seconds (1.5));
apps.Stop (Seconds (simtime-1));

ApplicationContainer sinkApp;
PacketSinkHelper sinkHelper ("ns3::UdpSocketFactory",
    Address(InetSocketAddress(ipv4AddrServer,5000)));
sinkApp = sinkHelper.Install (wifiStaNodes.Get (0));
sinkApp.Start (Seconds (1));
sinkApp.Stop (Seconds (simtime));
//*******************************************************************
Ipv4GlobalRoutingHelper::PopulateRoutingTables ();

Config::Connect("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxBeginModePower", ↩
```

```
      MakeCallback(&PhyRxBeginModePower));
  Config::Connect("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxEndMode", ↩
      MakeCallback(&PhyRxEndMode));
  Config::Connect("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDropMode", ↩
      MakeCallback(&PhyRxDropMode));
  Config::Connect("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDropModePower", ↩
      MakeCallback(&NotifyRxDropModePower));

  // ###############################
  //NS_LOG_INFO ("Iniciar Simulacion");
  // ###############################
  Simulator::Stop (Seconds (simtime));

  Simulator::Run ();

  Print();
  Simulator::Destroy ();

  return 0;
}
```

Código B.3: 2.1.2. Escenario verificador

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/applications-module.h"
#include "ns3/wifi-module.h"
#include "ns3/mobility-module.h"
#include "ns3/internet-module.h"
#include "ns3/propagation-delay-model.h"
#include "ns3/propagation-loss-model.h"

using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("Scenario");

uint32_t simt = 0;
NetDeviceContainer nap;
NetDeviceContainer nsta;

UniformVariable rv (0.5,1.5);

std::vector< std::vector<bool> > installed;
void
InitBoolMatrix()
{
  installed.resize(nap.GetN());
  for(uint32_t i = 0 ; i < nap.GetN() ; ++i)
  {
      installed[i].resize(nsta.GetN());
  }
}

void
SetCmdParams(uint32_t stime, NetDeviceContainer aps, NetDeviceContainer stas)
{
  simt = stime;
  nap = aps;
  nsta = stas;
  Simulator::Schedule (MilliSeconds (0), &InitBoolMatrix);
}

int
GetNodeNumber(std::string path)
{
  int nodeindex = 0;
  char indexChar = 0;
  std::string indexString;

  if(path[11]=='/')
    {
      indexChar = path[10];
      nodeindex = indexChar - '0';
    }
  else if(path[12]=='/')
    {
      indexChar = path[10];
      indexString += indexChar;
      indexChar = path[11];
      indexString += indexChar;
      nodeindex = atoi(indexString.c_str());
    }
  else
    {
      indexChar = path[10];
      indexString += indexChar;
      indexChar = path[11];
      indexString += indexChar;
      indexChar = path[12];
      indexString += indexChar;
      nodeindex = atoi(indexString.c_str());
    }
```

```
  return nodeindex;
}

void
Install_traffic(NetDeviceContainer apDevice, int ap_nodenumber, NetDeviceContainer staDevice↩
    , int sta_nodenumber)
{
  if(!installed[ap_nodenumber-1][sta_nodenumber-nap.GetN()])
    {
      std::cout << "INSTALAR " << ap_nodenumber  << "(" << ap_nodenumber-1 << ")" << " a " ↩
          << sta_nodenumber << "(" << sta_nodenumber-nap.GetN() << ")" << std::endl;
      Ptr<Ipv4> ipv4Server = staDevice.Get(sta_nodenumber-nap.GetN())->GetNode()->GetObject<↩
          Ipv4>();
      Ipv4InterfaceAddress iaddrServer = ipv4Server->GetAddress(1,0);
      Ipv4Address ipv4AddrServer = iaddrServer.GetLocal ();

      OnOffHelper onoff ("ns3::UdpSocketFactory", Address(InetSocketAddress(ipv4AddrServer↩
          ,5000)));
      onoff.SetAttribute ("OnTime", StringValue ("ns3::ConstantRandomVariable[Constant=1]"))↩
          ;
      onoff.SetAttribute ("OffTime", StringValue ("ns3::ConstantRandomVariable[Constant=0]")↩
          );
      onoff.SetAttribute ("DataRate", StringValue("20Mbps"));
      onoff.SetAttribute ("PacketSize", UintegerValue (1024));

      ApplicationContainer apps;
      apps.Add(onoff.Install (apDevice.Get(ap_nodenumber-1)->GetNode()));
      apps.Start (Simulator::Now ()+Seconds(rv.GetValue()));
      apps.Stop (Seconds (simt - 3));

      ApplicationContainer sinkApp;
      PacketSinkHelper sinkHelper ("ns3::UdpSocketFactory", Address(InetSocketAddress(↩
          ipv4AddrServer,5000)));
      sinkApp = sinkHelper.Install (staDevice.Get(sta_nodenumber-nap.GetN())->GetNode());
      sinkApp.Start (Simulator::Now ()+Seconds(rv.GetValue()));
      sinkApp.Stop (Seconds (simt));

      Ptr<WifiNetDevice> device = DynamicCast<WifiNetDevice>(staDevice.Get(sta_nodenumber-↩
          nap.GetN()));
      Ptr<WifiMac> mac = device->GetMac();
      Ptr<StaWifiMac> stamac = mac->GetObject<StaWifiMac>();

      installed[ap_nodenumber-1][sta_nodenumber-nap.GetN()] = true;
    }
}

void
Associate(std::string path, ns3::Mac48Address macaddr)
{
  std::cout << GetNodeNumber(path) << " associated to " << macaddr << std::endl;
  int nodenumber = GetNodeNumber(path);
  int apnumber = 0;
  for(uint32_t mac = 0; mac < nap.GetN()+1; mac++)
    {
      std::ostringstream convert;
      convert << mac;
      std::string result = convert.str();
      std::string refmac = "00:00:00:00:00:0" + result;
      ns3::Mac48Address refmacaddr(refmac.c_str());

      if ( operator == (refmacaddr, macaddr) )
        {
          apnumber = mac;
        }
      result.clear();
    }
  // ######################################
  // NS_LOG_INFO ("Crear e instalar trafico");
  // ######################################
  Simulator::Schedule (Seconds (rv.GetValue()), &Install_traffic, nap, apnumber, nsta, ↩
```

```
      nodenumber);
}

void
Dissociate(std::string path, ns3::Mac48Address macaddr)
{
  std::cout << GetNodeNumber(path) << " dissociated of " << macaddr << std::endl;
}

uint32_t PhyRxEndMcs0, PhyRxEndMcs1, PhyRxEndMcs2, PhyRxEndMcs3,
         PhyRxEndMcs4, PhyRxEndMcs5, PhyRxEndMcs6, PhyRxEndMcs7;
uint32_t endsimplechannel, enddoublechannel;

void
PhyRxEndMode(std::string path, Ptr<const Packet> p, WifiMode m)
{
  Ptr<Packet> copyp = p->Copy();
  WifiMacHeader hdr;
  copyp->RemoveHeader (hdr);

  if(hdr.IsData())
  {
    if(m.GetBandwidth() == 20000000){
      endsimplechannel++;
    }else {
      enddoublechannel++;
    }
    YansWifiPhy yans_wifi;
    uint32_t mcs = yans_wifi.WifiModeToMcs(m);
    switch (mcs) {
      case 0:
        PhyRxEndMcs0++;
        break;
      case 1:
        PhyRxEndMcs1++;
        break;
      case 2:
        PhyRxEndMcs2++;
        break;
      case 3:
        PhyRxEndMcs3++;
        break;
      case 4:
        PhyRxEndMcs4++;
        break;
      case 5:
        PhyRxEndMcs5++;
        break;
      case 6:
        PhyRxEndMcs6++;
        break;
      case 7:
        PhyRxEndMcs7++;
        break;
      default:
        break;
    }
  }
}

uint32_t PhyRxDropMcs0, PhyRxDropMcs1, PhyRxDropMcs2, PhyRxDropMcs3,
         PhyRxDropMcs4, PhyRxDropMcs5, PhyRxDropMcs6, PhyRxDropMcs7;
uint32_t dropsimplechannel, dropdoublechannel;

void
PhyRxDropMode(std::string path, Ptr<const Packet> p, WifiMode m)
{
  Ptr<Packet> copyp = p->Copy();
  WifiMacHeader hdr;
  copyp->RemoveHeader (hdr);
```

```
  if(hdr.IsData())
  {
    if(m.GetBandwidth() == 20000000){
      dropsimplechannel++;
    }else {
      dropdoublechannel++;
    }
    YansWifiPhy yans_wifi;
    uint32_t mcs = yans_wifi.WifiModeToMcs(m);
    switch (mcs) {
      case 0:
        PhyRxDropMcs0++;
        break;
      case 1:
        PhyRxDropMcs1++;
        break;
      case 2:
        PhyRxDropMcs2++;
        break;
      case 3:
        PhyRxDropMcs3++;
        break;
      case 4:
        PhyRxDropMcs4++;
        break;
      case 5:
        PhyRxDropMcs5++;
        break;
      case 6:
        PhyRxDropMcs6++;
        break;
      case 7:
        PhyRxDropMcs7++;
        break;
      default:
        break;
    }
  }
}

uint32_t PhyRxBeginMcs0, PhyRxBeginMcs1, PhyRxBeginMcs2, PhyRxBeginMcs3,
         PhyRxBeginMcs4, PhyRxBeginMcs5, PhyRxBeginMcs6, PhyRxBeginMcs7;
uint32_t beginsimplechannel, begindoublechannel;

double last_powerMcs0, last_powerMcs1, last_powerMcs2, last_powerMcs3,
       last_powerMcs4, last_powerMcs5, last_powerMcs6, last_powerMcs7;

void
PhyRxBeginModePower(std::string path, Ptr<const Packet> p, WifiMode m, double dbmpower)
{
  Ptr<Packet> copyp = p->Copy();
  WifiMacHeader hdr;
  copyp->RemoveHeader (hdr);

  if(hdr.IsData())
  {
    if(m.GetBandwidth() == 20000000){
      beginsimplechannel++;
    }else {
      begindoublechannel++;
    }

    YansWifiPhy yans_wifi;
    uint32_t mcs = yans_wifi.WifiModeToMcs(m);
    switch (mcs) {
      case 0:
        PhyRxBeginMcs0++;
        break;
      case 1:
```

```
        PhyRxBeginMcs1++;
        break;
      case 2:
        PhyRxBeginMcs2++;
        break;
      case 3:
        PhyRxBeginMcs3++;
        break;
      case 4:
        PhyRxBeginMcs4++;
        break;
      case 5:
        PhyRxBeginMcs5++;
        break;
      case 6:
        PhyRxBeginMcs6++;
        break;
      case 7:
        PhyRxBeginMcs7++;
        break;
      default:
        break;
    }
  }
}

uint32_t PhyRxDropPowerMcs0, PhyRxDropPowerMcs1, PhyRxDropPowerMcs2, PhyRxDropPowerMcs3,
         PhyRxDropPowerMcs4, PhyRxDropPowerMcs5, PhyRxDropPowerMcs6, PhyRxDropPowerMcs7;
uint32_t droppowersimplechannel, droppowerdoublechannel;

void
NotifyRxDropModePower(std::string path, Ptr<const Packet> p, WifiMode m, double dbmpower)
{
  Ptr<Packet> copyp = p->Copy();
  WifiMacHeader hdr;
  copyp->RemoveHeader (hdr);

  if(hdr.IsData())
  {
    if(m.GetBandwidth() == 20000000){
      droppowersimplechannel++;
    }else {
      droppowerdoublechannel++;
    }
    YansWifiPhy yans_wifi;
    uint32_t mcs = yans_wifi.WifiModeToMcs(m);
    switch (mcs) {
      case 0:
        PhyRxDropPowerMcs0++;
        last_powerMcs0 = dbmpower;
        break;
      case 1:
        PhyRxDropPowerMcs1++;
        last_powerMcs1 = dbmpower;
        break;
      case 2:
        PhyRxDropPowerMcs2++;
        last_powerMcs2 = dbmpower;
        break;
      case 3:
        PhyRxDropPowerMcs3++;
        last_powerMcs3 = dbmpower;
        break;
      case 4:
        PhyRxDropPowerMcs4++;
        last_powerMcs4 = dbmpower;
        break;
      case 5:
        PhyRxDropPowerMcs5++;
        last_powerMcs5 = dbmpower;
```

```
        break;
      case 6:
        PhyRxDropPowerMcs6++;
        last_powerMcs6 = dbmpower;
        break;
      case 7:
        PhyRxDropPowerMcs7++;
        last_powerMcs7 = dbmpower;
        break;
      default:
        break;
    }

  }
}

void
Print()
{
  NS_LOG_INFO ("-----------------------------------------");
  NS_LOG_INFO ("\tInicio\tFinal\tTirados\tTirados");
  NS_LOG_INFO ("\t\t\t(Sens)\t(PER)");

  NS_LOG_INFO ("MCS 0\t" << PhyRxBeginMcs0 << "\t" << PhyRxEndMcs0 << "\t"
                         << PhyRxDropPowerMcs0 << "\t" << PhyRxDropMcs0 << "\tpaquetes");
  NS_LOG_INFO ("MCS 1\t" << PhyRxBeginMcs1 << "\t" << PhyRxEndMcs1 << "\t"
                         << PhyRxDropPowerMcs1 << "\t" << PhyRxDropMcs1 << "\tpaquetes");
  NS_LOG_INFO ("MCS 2\t" << PhyRxBeginMcs2 << "\t" << PhyRxEndMcs2 << "\t"
                         << PhyRxDropPowerMcs2 << "\t" << PhyRxDropMcs2 << "\tpaquetes");
  NS_LOG_INFO ("MCS 3\t" << PhyRxBeginMcs3 << "\t" << PhyRxEndMcs3 << "\t"
                         << PhyRxDropPowerMcs3 << "\t" << PhyRxDropMcs3 << "\tpaquetes");
  NS_LOG_INFO ("MCS 4\t" << PhyRxBeginMcs4 << "\t" << PhyRxEndMcs4 << "\t"
                         << PhyRxDropPowerMcs4 << "\t" << PhyRxDropMcs4 << "\tpaquetes");
  NS_LOG_INFO ("MCS 5\t" << PhyRxBeginMcs5 << "\t" << PhyRxEndMcs5 << "\t"
                         << PhyRxDropPowerMcs5 << "\t" << PhyRxDropMcs5 << "\tpaquetes");
  NS_LOG_INFO ("MCS 6\t" << PhyRxBeginMcs6 << "\t" << PhyRxEndMcs6 << "\t"
                         << PhyRxDropPowerMcs6 << "\t" << PhyRxDropMcs6 << "\tpaquetes");
  NS_LOG_INFO ("MCS 7\t" << PhyRxBeginMcs7 << "\t" << PhyRxEndMcs7 << "\t"
                         << PhyRxDropPowerMcs7 << "\t" << PhyRxDropMcs7 << "\tpaquetes");
  NS_LOG_INFO ("-----------------------------------------");
  NS_LOG_INFO ("20 MHz\t" << beginsimplechannel << "\t" << endsimplechannel << "\t"
                          << droppowersimplechannel << "\t" << dropsimplechannel << "\↩
                              tpaquetes");
  NS_LOG_INFO ("40 MHz\t" << begindoublechannel << "\t" << enddoublechannel << "\t"
                          << droppowerdoublechannel << "\t" << dropdoublechannel << "\↩
                              tpaquetes");
  NS_LOG_INFO ("-----------------------------------------");
}

void
SimProgress ()
{
  Time time = Simulator::Now ();
  NS_LOG_INFO ("Progreso " << 100 * time.GetSeconds() / simt << "%");
  Simulator::Schedule (Seconds (5)-NanoSeconds(1),&SimProgress);
  Print();
}

int main (int argc, char *argv[])
{
  LogComponentEnable ("Scenario", LOG_LEVEL_INFO);

  uint32_t simtime = 5;
  uint32_t seed = 1;
  uint32_t nAp = 1;
  uint32_t nSta = 2;
  bool ch = false;
  bool sge = false;
  bool stafeed = false;
  uint32_t sessionTout = 2000;
```

```
CommandLine cmd;
cmd.AddValue ("simtime", "Simulation time (seconds)", simtime);
cmd.AddValue ("seed", "Seed for multiple simulations", seed);
cmd.AddValue ("nAp", "Number of APs", nAp);
cmd.AddValue ("nSta", "Number of STAs", nSta);
cmd.AddValue ("ChannelBonding", "ChannelBonding", ch);
cmd.AddValue ("ShortGuardEnabled", "ShortGuardEnabled", sge);
cmd.AddValue ("StaFeedback", "StaFeedback", stafeed);
cmd.AddValue ("SessionTimeout", "SessionTimeout in MilliSeconds (bigger than 1500 ms)", ↩
    sessionTout);
cmd.Parse (argc,argv);

if (sessionTout >= 1500)
{
  NS_LOG_INFO ("\nSimTime: " << simtime << "\tseed: " << seed << "\tch: " << ch
                                                    << "\tStaFeedback: " << ↩
                                                        stafeed
                                                    << "\tSessionTimeout: " << ↩
                                                        sessionTout << " ms\n");
}
else
{
  NS_LOG_INFO ("\nSessionTimeout must be bigger than 1500 ms\n");
  simtime = 0;
}

//Set the run number to generate a different random stream
SeedManager::SetRun(seed);

// ##########################
//NS_LOG_INFO ("Crear nodos");
// ##########################

// wifiApNode
NodeContainer wifiApNodes;
wifiApNodes.Create(nAp);
// StaNodes
NodeContainer wifiStaNodes;
wifiStaNodes.Create(nSta);

WifiHelper wifi = WifiHelper::Default ();
wifi.SetStandard (WIFI_PHY_STANDARD_80211n_2_4GHZ);

wifi.SetRemoteStationManager ("ns3::ArfWifiManager",
    "TimerThreshold", UintegerValue (150),
    "SuccessThreshold", UintegerValue (100));

// ############################
//NS_LOG_INFO ("Crear canal");
// ############################

/** PHY **/
YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ();
YansWifiChannelHelper channel = YansWifiChannelHelper::Default ();
wifiPhy.SetChannel (channel.Create());

wifiPhy.Set ("GreenfieldEnabled", BooleanValue(true));

wifiPhy.Set ("ShortGuardEnabled", BooleanValue(sge));
wifiPhy.Set ("ChannelBonding", BooleanValue(ch));

/** MAC **/
HtWifiMacHelper wifiMac = HtWifiMacHelper::Default ();

// ##############################
//NS_LOG_INFO ("Crear dispositivos");
// ##############################

NetDeviceContainer apDevice;
```

```
wifiMac.SetType ("ns3::ApWifiMac",
    "Ssid", SsidValue (Ssid ("Wi-Fi PFC")),
    "SessionTimeout", TimeValue (MilliSeconds (sessionTout)));
apDevice = wifi.Install (wifiPhy, wifiMac, wifiApNodes);

NetDeviceContainer staDevice;

wifiMac.SetType ("ns3::StaWifiMac",
    "Ssid", SsidValue (Ssid ("Wi-Fi PFC")),
    "StaFeedback",BooleanValue (stafeed));
staDevice = wifi.Install (wifiPhy, wifiMac, wifiStaNodes);

Ptr<ListPositionAllocator> allocator = CreateObject<ListPositionAllocator> ();
allocator->Add(Vector (0.0, 0.0, 0.0));       // AP 1
allocator->Add(Vector (-10.0, -10.0, 0.0));   // STA 1
allocator->Add(Vector (10.0, 10.0, 0.0));     // STA 2

MobilityHelper mobility;
mobility.SetPositionAllocator(allocator);
mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");
mobility.Install (wifiApNodes);

mobility.SetMobilityModel ("ns3::ConstantVelocityMobilityModel");
mobility.Install (wifiStaNodes);

Ptr<ConstantVelocityMobilityModel> mob = wifiStaNodes.Get(0)->GetObject<↩
    ConstantVelocityMobilityModel>();
mob->SetVelocity(Vector(3.0, 0, 0));
mob = wifiStaNodes.Get(1)->GetObject<ConstantVelocityMobilityModel>();
mob->SetVelocity(Vector(-3.0, 0, 0));

// Pilas de protocolos
InternetStackHelper stack;
stack.Install (wifiApNodes);
stack.Install (wifiStaNodes);

// ###################################
//NS_LOG_INFO ("Asignar direcciones IP");
// ###################################

Ipv4AddressHelper address;
// Para los nodos STA y nodo AP
address.SetBase ("192.168.1.0", "255.255.255.0");
Ipv4InterfaceContainer apInterfaces = address.Assign (apDevice);
Ipv4InterfaceContainer staInterfaces = address.Assign (staDevice);

Ipv4GlobalRoutingHelper::PopulateRoutingTables ();

Config::Connect("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxBeginModePower", ↩
    MakeCallback(&PhyRxBeginModePower));
Config::Connect("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxEndMode", ↩
    MakeCallback(&PhyRxEndMode));
Config::Connect("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDropMode", ↩
    MakeCallback(&PhyRxDropMode));
Config::Connect("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Phy/PhyRxDropModePower", ↩
    MakeCallback(&NotifyRxDropModePower));

Config::Connect("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/↩
    $ns3::StaWifiMac/DeAssoc", MakeCallback(&Dissociate));
Config::Connect("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/↩
    $ns3::StaWifiMac/Assoc", MakeCallback(&Associate));

Simulator::Schedule (Seconds (0), &SetCmdParams, simtime, apDevice, staDevice);
Simulator::Schedule (Seconds (5), &SimProgress);

// ###############################
//NS_LOG_INFO ("Iniciar Simulaci n");
// ###############################
if (simtime != 0)
{
```

```
    Simulator::Stop (Seconds (simtime));

    Simulator::Run ();

    //NS_LOG_INFO ("Simulaci n finalizada");
    Print();
  }
  Simulator::Destroy ();

  return 0;
}
```

Código B.4: 2.1.3. Escenario verificador

C. Modelado corregido

```
@@ -27,6 +27,7 @@
 #include "ns3/double.h"
 #include "ns3/uinteger.h"
 #include "ns3/wifi-phy.h"
+#include "ns3/yans-wifi-phy.h"
 #include "ns3/trace-source-accessor.h"
 #include "wifi-mac-header.h"
 #include "wifi-mac-trailer.h"
@@ -345,14 +346,22 @@ WifiRemoteStationManager::SetupPhy (Ptr<WifiPhy> phy)
   // transmit rate for automatic control responses like
   // acknowledgements.
   m_wifiPhy = phy;
-  m_defaultTxMode = phy->GetMode (0);
-  if(HasHtSupported())
+  if(DynamicCast<YansWifiPhy>(phy)->GetGreenfield())
     {
-       m_defaultTxMcs = phy->GetMcs (0);
+      m_defaultTxMode = phy->GetMembershipSelectorModes(0).at(0);
+      m_defaultTxMcs = phy->GetMcs (0);
     }
   else
     {
-      m_defaultTxMcs = 0;
+      m_defaultTxMode = phy->GetMode (0);
+      if(HasHtSupported())
+      {
+        m_defaultTxMcs = phy->GetMcs (0);
+      }
+    else
+      {
+        m_defaultTxMcs = 0;
+      }
     }
   Reset ();
 }
@@ -1324,7 +1333,14 @@ WifiMode
 WifiRemoteStationManager::GetSupported (const WifiRemoteStation *station, uint32_t i) const
 {
   NS_ASSERT (i < GetNSupported (station));
-  return station->m_state->m_operationalRateSet[i];
+  if (!DynamicCast<YansWifiPhy>(m_wifiPhy)->GetGreenfield())
+    {
+      return station->m_state->m_operationalRateSet[i];
+    }
+  else
+    {
+      return m_wifiPhy->McsToWifiMode(i);
+    }
 }
 uint8_t
 WifiRemoteStationManager::GetMcsSupported (const WifiRemoteStation *station, uint32_t i) ↩
    const
```

Código C.1: 2.1.1. Parche `wifi-remote-station-manager.cc`

```
@@ -30,6 +30,7 @@
 #include "qos-tag.h"
 #include "wifi-phy.h"
+#include "yans-wifi-phy.h"
 #include "dcf-manager.h"
 #include "mac-rx-middle.h"
 #include "mac-tx-middle.h"
@@ -309,10 +310,21 @@ ApWifiMac::GetSupportedRates (void) const
     }
   // send the set of supported rates and make sure that we indicate
   // the Basic Rate set in this set of supported rates.
-  for (uint32_t i = 0; i < m_phy->GetNModes (); i++)
+  if(DynamicCast<YansWifiPhy>(m_phy)->GetGreenfield())
     {
-      WifiMode mode = m_phy->GetMode (i);
-      rates.AddSupportedRate (mode.GetDataRate ());
+      for (uint32_t i = 0; i < m_phy->GetMembershipSelectorModes(0).size(); i++)
+        {
+          WifiMode mode = m_phy->GetMembershipSelectorModes(0).at(i);
+          rates.AddSupportedRate (mode.GetDataRate ());
+        }
+    }
+  else
+    {
+      for (uint32_t i = 0; i < m_phy->GetNModes (); i++)
+        {
+          WifiMode mode = m_phy->GetMode (i);
+          rates.AddSupportedRate (mode.GetDataRate ());
+        }
     }
   // set the basic rates
   for (uint32_t j = 0; j < m_stationManager->GetNBasicModes (); j++)
@@ -590,12 +602,26 @@ ApWifiMac::Receive (Ptr<Packet> packet, const WifiMacHeader *hdr)
                 {
                   // station supports all rates in Basic Rate Set.
                   // record all its supported modes in its associated WifiRemoteStation
-                  for (uint32_t j = 0; j < m_phy->GetNModes (); j++)
+                  if(DynamicCast<YansWifiPhy>(m_phy)->GetGreenfield())
                     {
+                      for (uint32_t i=0;i<m_phy->GetMembershipSelectorModes(0).size();i++)
+                        {
+                          WifiMode mode = m_phy->GetMembershipSelectorModes(0).at(i);
+                          if (rates.IsSupportedRate (mode.GetDataRate ()))
+                            {
+                              m_stationManager->AddSupportedMode (from, mode);
+                            }
+                        }
+                    }
+                  else
+                    {
-                      WifiMode mode = m_phy->GetMode (j);
-                      if (rates.IsSupportedRate (mode.GetDataRate ()))
+                      for (uint32_t j = 0; j < m_phy->GetNModes (); j++)
                         {
-                          m_stationManager->AddSupportedMode (from, mode);
+                          WifiMode mode = m_phy->GetMode (j);
+                          if (rates.IsSupportedRate (mode.GetDataRate ()))
+                            {
+                              m_stationManager->AddSupportedMode (from, mode);
+                            }
                         }
                     }
                   if (m_htSupported)
```

Código C.2: 2.1.1. Parche `ap-wifi-mac.cc`

```
@@ -29,6 +29,7 @@
 #include "qos-tag.h"
+#include "yans-wifi-phy.h"
 #include "mac-low.h"
 #include "dcf-manager.h"
@@ -552,16 +553,29 @@ StaWifiMac::Receive (Ptr<Packet> packet, const WifiMacHeader *hdr)
                 HtCapabilities htcapabilities = assocResp.GetHtCapabilities ();
                 m_stationManager->AddStationHtCapabilities(hdr->GetAddr2(),htcapabilities);
                 }
-
-              for (uint32_t i = 0; i < m_phy->GetNModes (); i++)
+              if(DynamicCast<YansWifiPhy>(m_phy)->GetGreenfield())
+                {
+                  for (uint32_t i=0; i<m_phy->GetMembershipSelectorModes(0).size(); i++)
+                    {
+                      WifiMode mode = m_phy->GetMembershipSelectorModes(0).at(i);
+                      if (rates.IsSupportedRate (mode.GetDataRate ()))
+                        {
+                          m_stationManager->AddSupportedMode (hdr->GetAddr2 (), mode);
+                        }
+                    }
+                }
+              else
                 {
-                  WifiMode mode = m_phy->GetMode (i);
-                  if (rates.IsSupportedRate (mode.GetDataRate ()))
+                  for (uint32_t i = 0; i < m_phy->GetNModes (); i++)
                     {
-                      m_stationManager->AddSupportedMode (hdr->GetAddr2 (), mode);
-                      if (rates.IsBasicRate (mode.GetDataRate ()))
+                      WifiMode mode = m_phy->GetMode (i);
+                      if (rates.IsSupportedRate (mode.GetDataRate ()))
                         {
-                          m_stationManager->AddBasicMode (mode);
+                          m_stationManager->AddSupportedMode (hdr->GetAddr2 (), mode);
+                          if (rates.IsBasicRate (mode.GetDataRate ()))
+                            {
+                              m_stationManager->AddBasicMode (mode);
+                            }
                         }
                     }
                 }
@@ -609,10 +623,21 @@ StaWifiMac::GetSupportedRates (void) const
           rates.SetBasicRate(m_phy->GetBssMembershipSelector(i));
         }
     }
-  for (uint32_t i = 0; i < m_phy->GetNModes (); i++)
+  if(DynamicCast<YansWifiPhy>(m_phy)->GetGreenfield())
     {
-      WifiMode mode = m_phy->GetMode (i);
-      rates.AddSupportedRate (mode.GetDataRate ());
+      for (uint32_t i = 0; i < m_phy->GetMembershipSelectorModes(0).size(); i++)
+        {
+          WifiMode mode = m_phy->GetMembershipSelectorModes(0).at(i);
+          rates.AddSupportedRate (mode.GetDataRate ());
+        }
+    }
+  else
+    {
+      for (uint32_t i = 0; i < m_phy->GetNModes (); i++)
+        {
+          WifiMode mode = m_phy->GetMode (i);
+          rates.AddSupportedRate (mode.GetDataRate ());
+        }
     }
   return rates;
 }
```

Código C.3: 2.1.1. Parche `sta-wifi-mac.cc`

```
@@ -68,12 +68,21 @@ WifiPhy::GetTypeId (void)
     .AddTraceSource ("PhyRxBegin",
                      "Trace source indicating a packet has begun being received from the ↩
                          channel medium by the device",
                      MakeTraceSourceAccessor (&WifiPhy::m_phyRxBeginTrace))
+    .AddTraceSource ("PhyRxBeginMode",
+                     "Trace source indicating a packet has begun being received from the ↩
    channel medium by the device (mode)",
+                     MakeTraceSourceAccessor (&WifiPhy::m_phyRxBeginModeTrace))
     .AddTraceSource ("PhyRxEnd",
                      "Trace source indicating a packet has been completely received from ↩
                          the channel medium by the device",
                      MakeTraceSourceAccessor (&WifiPhy::m_phyRxEndTrace))
+    .AddTraceSource ("PhyRxEndMode",
+                     "Trace source indicating a packet has been completely received from ↩
    the channel medium by the device (mode)",
+                     MakeTraceSourceAccessor (&WifiPhy::m_phyRxEndModeTrace))
     .AddTraceSource ("PhyRxDrop",
                      "Trace source indicating a packet has been dropped by the device ↩
                          during reception",
                      MakeTraceSourceAccessor (&WifiPhy::m_phyRxDropTrace))
+    .AddTraceSource ("PhyRxDropMode",
+                     "Trace source indicating a packet has been dropped by the device ↩
    during reception (mode)",
+                     MakeTraceSourceAccessor (&WifiPhy::m_phyRxDropModeTrace))
     .AddTraceSource ("MonitorSnifferRx",
                      "Trace source simulating a wifi device in monitor mode sniffing all ↩
                          received frames",
                      MakeTraceSourceAccessor (&WifiPhy::m_phyMonitorSniffRxTrace))
@@ -475,18 +484,36 @@ WifiPhy::NotifyRxBegin (Ptr<const Packet> packet)
 void
+WifiPhy::NotifyRxBeginMode (Ptr<const Packet> packet, WifiMode wifimode)
+{
+  m_phyRxBeginModeTrace (packet, wifimode);
+}
+
+void
 WifiPhy::NotifyRxEnd (Ptr<const Packet> packet)
 {
   m_phyRxEndTrace (packet);
 }

 void
+WifiPhy::NotifyRxEndMode (Ptr<const Packet> packet, WifiMode wifimode)
+{
+  m_phyRxEndModeTrace (packet, wifimode);
+}
+
+void
 WifiPhy::NotifyRxDrop (Ptr<const Packet> packet)
 {
   m_phyRxDropTrace (packet);
 }

 void
+WifiPhy::NotifyRxDropMode (Ptr<const Packet> packet, WifiMode wifimode)
+{
+  m_phyRxDropModeTrace (packet, wifimode);
+}
+
+void
 WifiPhy::NotifyMonitorSniffRx (Ptr<const Packet> packet, uint16_t channelFreqMhz, uint16_t ↩
     channelNumber, uint32_t rate, bool isShortPreamble, double signalDbm, double noiseDbm)
 {
   m_phyMonitorSniffRxTrace (packet, channelFreqMhz, channelNumber, rate, isShortPreamble, ↩
       signalDbm, noiseDbm);
```

Código C.4: 2.1.1. Parche `wifi-phy.cc`

```
@@ -921,6 +921,8 @@ public:
    */
   void NotifyRxBegin (Ptr<const Packet> packet);

+  void NotifyRxBeginMode (Ptr<const Packet> packet, WifiMode wifimode);
+
   /**
    * Public method used to fire a PhyRxEnd trace.  Implemented for encapsulation
    * purposes.
@@ -929,6 +931,8 @@ public:
    */
   void NotifyRxEnd (Ptr<const Packet> packet);

+  void NotifyRxEndMode (Ptr<const Packet> packet, WifiMode wifimode);
+
   /**
    * Public method used to fire a PhyRxDrop trace.  Implemented for encapsulation
    * purposes.
@@ -937,6 +941,8 @@ public:
    */
   void NotifyRxDrop (Ptr<const Packet> packet);

+  void NotifyRxDropMode (Ptr<const Packet> packet, WifiMode wifimode);
+
   /**
    *
    * Public method used to fire a MonitorSniffer trace for a wifi packet being received.  ↩
        Implemented for encapsulation
@@ -1086,6 +1092,8 @@ private:
    */
   TracedCallback<Ptr<const Packet> > m_phyRxBeginTrace;

+  TracedCallback<Ptr<const Packet>, WifiMode > m_phyRxBeginModeTrace;
+
   /**
    * The trace source fired when a packet ends the reception process from
    * the medium.
@@ -1094,6 +1102,8 @@ private:
    */
   TracedCallback<Ptr<const Packet> > m_phyRxEndTrace;

+  TracedCallback<Ptr<const Packet>, WifiMode > m_phyRxEndModeTrace;
+
   /**
    * The trace source fired when the phy layer drops a packet it has received.
    *
@@ -1101,6 +1111,8 @@ private:
    */
   TracedCallback<Ptr<const Packet> > m_phyRxDropTrace;

+  TracedCallback<Ptr<const Packet>, WifiMode > m_phyRxDropModeTrace;
+
   /**
    * A trace source that emulates a wifi device in monitor mode
    * sniffing a packet being received.
```

Código C.5: 2.1.1. Parche `wifi-phy.h`

```
@@ -514,6 +514,7 @@ WifiMode txMode=txVector.GetMode();
           m_state->SwitchToRx (rxDuration);
           NS_ASSERT (m_endRxEvent.IsExpired ());
           NotifyRxBegin (packet);
+          NotifyRxBeginMode (packet, txVector.GetMode());
           m_interference.NotifyRxStart ();
           m_endRxEvent = Simulator::Schedule (rxDuration, &YansWifiPhy::EndReceive, this,
                                               packet,
@@ -800,6 +801,7 @@ YansWifiPhy::EndReceive (Ptr<Packet> packet, Ptr<InterferenceHelper::↩
    Event> even
   if (m_random->GetValue () > snrPer.per)
     {
       NotifyRxEnd (packet);
+      NotifyRxEndMode (packet, event->GetTxVector().GetMode());
       uint32_t dataRate500KbpsUnits = event->GetPayloadMode ().GetDataRate () * event->↩
           GetTxVector().GetNss()/ 500000;
       bool isShortPreamble = (WIFI_PREAMBLE_SHORT == event->GetPreambleType ());
       double signalDbm = RatioToDb (event->GetRxPowerW ()) + 30;
@@ -811,6 +813,7 @@ YansWifiPhy::EndReceive (Ptr<Packet> packet, Ptr<InterferenceHelper::↩
    Event> even
     {
       /* failure. */
       NotifyRxDrop (packet);
+      NotifyRxDropMode (packet, event->GetTxVector().GetMode());
       m_state->SwitchFromRxEndError (packet, snrPer.snr);
     }
 }
```

Código C.6: 2.1.1. Parche `yans-wifi-phy.cc`

```
@@ -24,6 +24,7 @@
 #include "wifi-mode.h"
 #include "wifi-preamble.h"
 #include "wifi-phy-state-helper.h"
+#include "wifi-mac-header.h"
 #include "error-rate-model.h"
 #include "ns3/simulator.h"
 #include "ns3/packet.h"
@@ -162,8 +163,10 @@ YansWifiPhy::GetTypeId (void)
                    MakeBooleanAccessor (&YansWifiPhy::GetChannelBonding,
                                         &YansWifiPhy::SetChannelBonding),
                    MakeBooleanChecker ())
-
-
+    .AddAttribute ("ExtraSens", "Extra sensitivity hardware specifications.",
+                   UintegerValue (0),
+                   MakeUintegerAccessor (&YansWifiPhy::m_extraSensitivity),
+                   MakeUintegerChecker<uint32_t> ())
   ;
   return tid;
 }
@@ -451,6 +454,7 @@ YansWifiPhy::StartReceivePacket (Ptr<Packet> packet,
   NS_LOG_FUNCTION (this << packet << rxPowerDbm << txVector.GetMode()<< preamble);
   rxPowerDbm += m_rxGainDb;
   double rxPowerW = DbmToW (rxPowerDbm);
+  bool ofdmThreshold = true;
   Time rxDuration = CalculateTxDuration (packet->GetSize (), txVector, preamble);
 WifiMode txMode=txVector.GetMode();
   Time endRx = Simulator::Now () + rxDuration;
@@ -507,14 +511,23 @@ WifiMode txMode=txVector.GetMode();
       break;
     case YansWifiPhy::CCA_BUSY:
     case YansWifiPhy::IDLE:
-      if (rxPowerW > m_edThresholdW)
+
+      Ptr<Packet> copyp = packet->Copy();
+      WifiMacHeader hdr;
+      copyp->RemoveHeader (hdr);
+
+      if(txVector.GetMode().GetModulationClass() == WIFI_MOD_CLASS_HT)
+        {
+          ofdmThreshold = IsOfdmThreshold(rxPowerDbm, txVector.GetMode());
+        }
+      NotifyRxBeginModePower (packet, txVector.GetMode(), WToDbm(rxPowerW));
+      if (rxPowerW > m_edThresholdW && ofdmThreshold)
         {
           NS_LOG_DEBUG ("sync to signal (power=" << rxPowerW << "W)");
           // sync to signal
           m_state->SwitchToRx (rxDuration);
           NS_ASSERT (m_endRxEvent.IsExpired ());
           NotifyRxBegin (packet);
-          NotifyRxBeginMode (packet, txVector.GetMode());
           m_interference.NotifyRxStart ();
           m_endRxEvent = Simulator::Schedule (rxDuration, &YansWifiPhy::EndReceive, this,
                                               packet,
@@ -522,8 +535,17 @@ WifiMode txMode=txVector.GetMode();
         }
       else
         {
-          NS_LOG_DEBUG ("drop packet because signal power too Small (" <<
-                        rxPowerW << "<" << m_edThresholdW << ")");
+          if(!ofdmThreshold)
+            {
+              NS_LOG_DEBUG ("drop packet because signal (" << rxPowerDbm <<
+                    ") is to small for " << txVector.GetMode());
+              NotifyRxDropModePower (packet, txVector.GetMode(), WToDbm(rxPowerW));
+            }
+          else
+            {
+              NS_LOG_DEBUG ("drop packet because signal power too Small (" <<
```

```
+                    rxPowerW << "<" << m_edThresholdW << ")");
+             }
            NotifyRxDrop (packet);
            goto maybeCcaBusy;
          }
@@ -1210,4 +1232,54 @@ YansWifiPhy::McsToWifiMode (uint8_t mcs)
          }
   return mode;
 }
+
+bool
+YansWifiPhy::IsOfdmThreshold (double rxPowerDbm, WifiMode txMode)
+{
+  bool decition = true;
+  uint32_t bw = txMode.GetBandwidth();
+  uint32_t cbw = 20000000;
+  rxPowerDbm += m_extraSensitivity;
+
+  switch (WifiModeToMcs(txMode))
+  {
+  case 0:
+    if( rxPowerDbm < ((bw == cbw) ? -82 : -79) )
+      decition = false;
+    break;
+  case 1:
+    if( rxPowerDbm < ((bw == cbw) ? -79 : -76) )
+      decition = false;
+    break;
+  case 2:
+    if( rxPowerDbm < ((bw == cbw) ? -77 : -74) )
+      decition = false;
+    break;
+  case 3:
+    if( rxPowerDbm < ((bw == cbw) ? -74 : -71) )
+      decition = false;
+    break;
+  case 4:
+    if( rxPowerDbm < ((bw == cbw) ? -70 : -67) )
+      decition = false;
+    break;
+  case 5:
+    if( rxPowerDbm < ((bw == cbw) ? -66 : -63) )
+      decition = false;
+    break;
+  case 6:
+    if( rxPowerDbm < ((bw == cbw) ? -65 : -62) )
+      decition = false;
+    break;
+  case 7:
+    if( rxPowerDbm < ((bw == cbw) ? -64 : -61) )
+      decition = false;
+    break;
+  default:
+    NS_ASSERT (false);
+    break;
+  }
+  return decition;
+}
+
 } // namespace ns3
```

Código C.7: 2.1.2. Parche `yans-wifi-phy.cc`

```
@@ -451,6 +451,10 @@ private:
    * \param event the corresponding event of the first time the packet arrives
    */
   void EndReceive (Ptr<Packet> packet, Ptr<InterferenceHelper::Event> event);
+  /**
+   *
+   */
+  bool IsOfdmThreshold (double rxPowerDbm, WifiMode txMode);

 private:
   double   m_edThresholdW;        //!< Energy detection threshold in watts
@@ -473,6 +477,7 @@ private:
   bool     m_greenfield;            //!< Flag if GreenField format is supported
   bool     m_guardInterval;         //!< Flag if short guard interval is used
   bool     m_channelBonding;        //!< Flag if channel conding is used
+  uint32_t m_extraSensitivity;      //!< Add extra sensitivity to hardware specifications


   /**
```

Código C.8: 2.1.2. Parche `yans-wifi-phy.h`

```
@@ -68,9 +68,9 @@ WifiPhy::GetTypeId (void)
     .AddTraceSource ("PhyRxBegin",
                      "Trace source indicating a packet has begun being received from the ↩
                          channel medium by the device",
                      MakeTraceSourceAccessor (&WifiPhy::m_phyRxBeginTrace))
-    .AddTraceSource ("PhyRxBeginMode",
-                     "Trace source indicating a packet has begun being received from the ↩
    channel medium by the device (mode)",
-                     MakeTraceSourceAccessor (&WifiPhy::m_phyRxBeginModeTrace))
+    .AddTraceSource ("PhyRxBeginModePower",
+                     "Trace source indicating a packet has begun being received from the ↩
    channel medium by the device (mode and power)",
+                     MakeTraceSourceAccessor (&WifiPhy::m_phyRxBeginModePowerTrace))
     .AddTraceSource ("PhyRxEnd",
                      "Trace source indicating a packet has been completely received from ↩
                          the channel medium by the device",
                      MakeTraceSourceAccessor (&WifiPhy::m_phyRxEndTrace))
@@ -83,6 +83,9 @@ WifiPhy::GetTypeId (void)
     .AddTraceSource ("PhyRxDropMode",
                      "Trace source indicating a packet has been dropped by the device ↩
                          during reception (mode)",
                      MakeTraceSourceAccessor (&WifiPhy::m_phyRxDropModeTrace))
+    .AddTraceSource ("PhyRxDropModePower",
+                     "Trace source indicating a packet has been dropped by the device ↩
    during reception (mode and power)",
+                     MakeTraceSourceAccessor (&WifiPhy::m_phyRxDropModePowerTrace))
     .AddTraceSource ("MonitorSnifferRx",
                      "Trace source simulating a wifi device in monitor mode sniffing all ↩
                          received frames",
                      MakeTraceSourceAccessor (&WifiPhy::m_phyMonitorSniffRxTrace))
@@ -484,9 +487,9 @@ WifiPhy::NotifyRxBegin (Ptr<const Packet> packet)
 }

 void
-WifiPhy::NotifyRxBeginMode (Ptr<const Packet> packet, WifiMode wifimode)
+WifiPhy::NotifyRxBeginModePower (Ptr<const Packet> packet, WifiMode wifimode, double power)
 {
-  m_phyRxBeginModeTrace (packet, wifimode);
+  m_phyRxBeginModePowerTrace (packet, wifimode, power);
 }

 void
@@ -514,6 +517,12 @@ WifiPhy::NotifyRxDropMode (Ptr<const Packet> packet, WifiMode wifimode)
 }

 void
+WifiPhy::NotifyRxDropModePower (Ptr<const Packet> packet, WifiMode wifimode, double power)
+{
+  m_phyRxDropModePowerTrace (packet, wifimode, power);
+}
+
+void
 WifiPhy::NotifyMonitorSniffRx (Ptr<const Packet> packet, uint16_t channelFreqMhz, uint16_t ↩
     channelNumber, uint32_t rate, bool isShortPreamble, double signalDbm, double noiseDbm)
 {
   m_phyMonitorSniffRxTrace (packet, channelFreqMhz, channelNumber, rate, isShortPreamble, ↩
       signalDbm, noiseDbm);
```

Código C.9: 2.1.2. Parche `wifi-phy.cc`

```
@@ -921,7 +921,7 @@ public:
    */
   void NotifyRxBegin (Ptr<const Packet> packet);

-  void NotifyRxBeginMode (Ptr<const Packet> packet, WifiMode wifimode);
+  void NotifyRxBeginModePower (Ptr<const Packet> packet, WifiMode wifimode, double power);

   /**
    * Public method used to fire a PhyRxEnd trace.  Implemented for encapsulation
@@ -943,6 +943,8 @@ public:

   void NotifyRxDropMode (Ptr<const Packet> packet, WifiMode wifimode);

+  void NotifyRxDropModePower (Ptr<const Packet> packet, WifiMode wifimode, double power);
+
   /**
    *
    * Public method used to fire a MonitorSniffer trace for a wifi packet being received.  ↩
        Implemented for encapsulation
@@ -1092,7 +1094,7 @@ private:
    */
   TracedCallback<Ptr<const Packet> > m_phyRxBeginTrace;

-  TracedCallback<Ptr<const Packet>, WifiMode > m_phyRxBeginModeTrace;
+  TracedCallback<Ptr<const Packet>, WifiMode, double > m_phyRxBeginModePowerTrace;

   /**
    * The trace source fired when a packet ends the reception process from
@@ -1113,6 +1115,8 @@ private:

   TracedCallback<Ptr<const Packet>, WifiMode > m_phyRxDropModeTrace;

+  TracedCallback<Ptr<const Packet>, WifiMode, double > m_phyRxDropModePowerTrace;
+
   /**
    * A trace source that emulates a wifi device in monitor mode
    * sniffing a packet being received.
```

Código C.10: 2.1.2. Parche `wifi-phy.h`

```
@@ -1089,4 +1089,55 @@ MgtDelBaHeader::SetParameterSet (uint16_t params)
   m_tid = (params >> 12) & 0x0f;
 }

+/**************************************************
+ *          STA Feedback
+ **************************************************/
+
+NS_OBJECT_ENSURE_REGISTERED (MgtStaFeedbackHeader)
+  ;
+
+MgtStaFeedbackHeader::MgtStaFeedbackHeader ()
+  : m_padding (0)
+{
+}
+MgtStaFeedbackHeader::~MgtStaFeedbackHeader ()
+{
+}
+
+TypeId
+MgtStaFeedbackHeader::GetTypeId (void)
+{
+  static TypeId tid = TypeId ("ns3::MgtStaFeedbackHeader")
+    .SetParent<Header> ()
+    .AddConstructor<MgtStaFeedbackHeader> ()
+  ;
+  return tid;
+}
+TypeId
+MgtStaFeedbackHeader::GetInstanceTypeId (void) const
+{
+  return GetTypeId ();
+}
+void
+MgtStaFeedbackHeader::Print (std::ostream &os) const
+{
+}
+uint32_t
+MgtStaFeedbackHeader::GetSerializedSize (void) const
+{
+  return 1;
+}
+void
+MgtStaFeedbackHeader::Serialize (Buffer::Iterator start) const
+{
+  start.WriteU8 (m_padding);
+}
+uint32_t
+MgtStaFeedbackHeader::Deserialize (Buffer::Iterator start)
+{
+  Buffer::Iterator i = start;
+  m_padding = i.ReadU8 ();
+  return i.GetDistanceFrom (start);
+}
+
 } // namespace ns3
```

Código C.11: 2.1.3. Parche `mgt-headers.cc`

```
@@ -756,6 +756,27 @@ private:
   uint16_t m_reasonCode;
 };

+/**
+ * \ingroup wifi
+ * Implement the header for management frames of sta feedback.
+ */
+class MgtStaFeedbackHeader : public Header
+{
+public:
+  MgtStaFeedbackHeader ();
+  ~MgtStaFeedbackHeader ();
+
+  static TypeId GetTypeId (void);
+  virtual TypeId GetInstanceTypeId (void) const;
+  virtual void Print (std::ostream &os) const;
+  virtual uint32_t GetSerializedSize (void) const;
+  virtual void Serialize (Buffer::Iterator start) const;
+  virtual uint32_t Deserialize (Buffer::Iterator start);
+
+private:
+  uint8_t m_padding;
+};
+
 } // namespace ns3

 #endif /* MGT_HEADERS_H */
```

Código C.12: 2.1.3. Parche `mgt-headers.h`

```
@@ -161,6 +161,12 @@ WifiMacHeader::SetMultihopAction (void)
   m_ctrlSubtype = 0x0F;
 }
 void
+WifiMacHeader::SetStaFeedback (void)
+{
+  m_ctrlType = TYPE_MGT;
+  m_ctrlSubtype = 6;
+}
+void
 WifiMacHeader::SetType (enum WifiMacType type)
 {
   switch (type)
@@ -241,6 +247,10 @@ WifiMacHeader::SetType (enum WifiMacType type)
       m_ctrlType = TYPE_MGT;
       m_ctrlSubtype = 15;
       break;
+    case WIFI_MAC_MGT_STA_FEEDBACK:
+      m_ctrlType = TYPE_MGT;
+      m_ctrlSubtype = 6;
+      break;

     case WIFI_MAC_DATA:
       m_ctrlType = TYPE_DATA;
@@ -460,6 +470,9 @@ WifiMacHeader::GetType (void) const
         case 5:
           return WIFI_MAC_MGT_PROBE_RESPONSE;
           break;
+        case 6:
+          return WIFI_MAC_MGT_STA_FEEDBACK;
+          break;
         case 8:
           return WIFI_MAC_MGT_BEACON;
           break;
@@ -688,6 +701,11 @@ WifiMacHeader::IsMultihopAction (void) const
   return (GetType () == WIFI_MAC_MGT_MULTIHOP_ACTION);
 }
 bool
+WifiMacHeader::IsStaFeedback (void) const
+{
+  return (GetType () == WIFI_MAC_MGT_STA_FEEDBACK);
+}
+bool
 WifiMacHeader::IsBlockAckReq (void) const
 {
   return (GetType () == WIFI_MAC_CTL_BACKREQ) ? true : false;
@@ -928,6 +946,7 @@ case WIFI_MAC_ ## x: \
       FOO (MGT_ACTION);
       FOO (MGT_ACTION_NO_ACK);
       FOO (MGT_MULTIHOP_ACTION);
+      FOO (MGT_STA_FEEDBACK);

       FOO (DATA);
       FOO (DATA_CFACK);
@@ -1053,6 +1072,7 @@ WifiMacHeader::Print (std::ostream &os) const
       os << ", FragNumber=" << std::hex << (int) m_seqFrag << std::dec
          << ", SeqNumber=" << m_seqSeq;
       break;
+    case WIFI_MAC_MGT_STA_FEEDBACK:
     case WIFI_MAC_DATA_CFACK:
     case WIFI_MAC_DATA_CFPOLL:
     case WIFI_MAC_DATA_CFACK_CFPOLL:
```

Código C.13: 2.1.3. Parche `wifi-mac-header.cc`

```
@@ -54,6 +54,7 @@ enum WifiMacType
   WIFI_MAC_MGT_ACTION,
   WIFI_MAC_MGT_ACTION_NO_ACK,
   WIFI_MAC_MGT_MULTIHOP_ACTION,
+  WIFI_MAC_MGT_STA_FEEDBACK,

   WIFI_MAC_DATA,
   WIFI_MAC_DATA_CFACK,
@@ -154,6 +155,10 @@ public:
    */
   void SetMultihopAction ();
   /**
+   * Set Type/Subtype values for sta feedback header.
+   */
+  void SetStaFeedback ();
+  /**
    * Set the From DS bit in the Frame Control field.
    */
   void SetDsFrom (void);
@@ -477,6 +482,13 @@ public:
    */
   bool IsMultihopAction () const;
   /**
+   * Check if the header is sta feedback header.
+   *
+   * \return true if the header is sta feedback header,
+   *         false otherwise
+   */
+  bool IsStaFeedback () const;
+  /**
    * Return the raw duration from the Duration/ID field.
    *
    * \return the raw duration from the Duration/ID field
```

Código C.14: 2.1.3. Parche `wifi-mac-header.h`

```
@@ -88,6 +88,10 @@ StaWifiMac::GetTypeId (void)
                    MakeBooleanAccessor (&StaWifiMac::SetActiveProbing),
                    MakeBooleanChecker ())
+    .AddAttribute ("StaFeedback", "If true, we send sta feedback ",
+                   BooleanValue (false),
+                   MakeBooleanAccessor (&StaWifiMac::SetStaFeedback),
+                   MakeBooleanChecker ())
     .AddTraceSource ("Assoc", "Associated with an access point.",
                      MakeTraceSourceAccessor (&StaWifiMac::m_assocLogger))
     .AddTraceSource ("DeAssoc", "Association with an access point lost.",
@@ -100,7 +104,8 @@ StaWifiMac::StaWifiMac ()
   : m_state (BEACON_MISSED),
     m_probeRequestEvent (),
     m_assocRequestEvent (),
-    m_beaconWatchdogEnd (Seconds (0.0))
+    m_beaconWatchdogEnd (Seconds (0.0)),
+    m_tid ()
 {
   NS_LOG_FUNCTION (this);

@@ -157,6 +162,12 @@ StaWifiMac::SetActiveProbing (bool enable)
 }

 void
+StaWifiMac::SetStaFeedback (bool enable)
+{
+  m_staFeedback = enable;
+}
+
+void
 StaWifiMac::SendProbeRequest (void)
 {
   NS_LOG_FUNCTION (this);
@@ -297,6 +308,22 @@ StaWifiMac::MissedBeacons (void)
         {
           m_beaconWatchdog.Cancel ();
         }
+      if(m_staFeedback)
+        {
+          Ptr<Packet> packet = Create<Packet> ();
+          WifiMacHeader hdr;
+          hdr.SetStaFeedback ();
+          hdr.SetAddr1 (GetBssid ());
+          hdr.SetAddr2 (GetAddress ());
+          hdr.SetAddr3 (GetBssid ());
+          hdr.SetDsNotFrom ();
+          hdr.SetDsNotTo ();
+          MgtStaFeedbackHeader stafeedback;
+          packet->AddHeader (stafeedback);
+          m_dca->Queue (packet, hdr);
+        }
       m_beaconWatchdog = Simulator::Schedule (m_beaconWatchdogEnd - Simulator::Now (),
                                               &StaWifiMac::MissedBeacons, this);
       return;
@@ -346,7 +373,7 @@ StaWifiMac::Enqueue (Ptr<const Packet> packet, Mac48Address to)
   // If we are not a QoS AP then we definitely want to use AC_BE to
   // transmit the packet. A TID of zero will map to AC_BE (through \c
   // QosUtilsMapTidToAc()), so we use that as our default here.
-  uint8_t tid = 0;
+  //uint8_t tid = 0;

   // For now, an AP that supports QoS does not support non-QoS
   // associations, and vice versa. In future the AP model should
@@ -364,15 +391,15 @@ StaWifiMac::Enqueue (Ptr<const Packet> packet, Mac48Address to)
       hdr.SetQosTxopLimit (0);

       // Fill in the QoS control field in the MAC header
-      tid = QosUtilsGetTidForPacket (packet);
+      m_tid = QosUtilsGetTidForPacket (packet);
       // Any value greater than 7 is invalid and likely indicates that
```

```
       // the packet had no QoS tag, so we revert to zero, which'll
       // mean that AC_BE is used.
-      if (tid >= 7)
+      if (m_tid >= 7)
         {
-          tid = 0;
+          m_tid = 0;
         }
-      hdr.SetQosTid (tid);
+      hdr.SetQosTid (m_tid);
     }
   else
@@ -392,8 +419,8 @@ if (m_htSupported)
   if (m_qosSupported)
     {
       // Sanity check that the TID is valid
-      NS_ASSERT (tid < 8);
-      m_edca[QosUtilsMapTidToAc (tid)]->Queue (packet, hdr);
+      NS_ASSERT (m_tid < 8);
+      m_edca[QosUtilsMapTidToAc (m_tid)]->Queue (packet, hdr);
     }
   else
     {
@@ -667,6 +694,7 @@ StaWifiMac::SetState (MacState value)
            && m_state == ASSOCIATED)
     {
       m_deAssocLogger (GetBssid ());
+      m_edca[QosUtilsMapTidToAc (m_tid)]->NotifyChannelSwitching();
     }
   m_state = value;
 }
```

Código C.15: 2.1.3. Parche `sta-wifi-mac.cc`

```
@@ -31,6 +31,9 @@
 #include "supported-rates.h"
 #include "amsdu-subframe-header.h"

+#include "ap-wifi-mac.h"
+#include "ns3/wifi-module.h"
+
 namespace ns3 {

 class MgtAddBaRequestHeader;
@@ -103,6 +106,12 @@ private:
    */
   void SetActiveProbing (bool enable);
   /**
+   * Enable or disable sta feedback.
+   *
+   * \param enable enable or disable sta feedback
+   */
+  void SetStaFeedback (bool enable);
+  /**
    * Return whether active probing is enabled.
    *
    * \return true if active probing is enabled, false otherwise
@@ -187,6 +196,8 @@ private:
   EventId m_beaconWatchdog;
   Time m_beaconWatchdogEnd;
   uint32_t m_maxMissedBeacons;
+  uint8_t m_tid;
+  bool m_staFeedback;

   TracedCallback<Mac48Address> m_assocLogger;
   TracedCallback<Mac48Address> m_deAssocLogger;
```

Código C.16: 2.1.3. Parche `sta-wifi-mac.h`

```
@@ -559,6 +559,11 @@ RegularWifiMac::Receive (Ptr<Packet> packet, const WifiMacHeader *hdr)
           return;
         }
     }
+  else if(hdr->IsStaFeedback())
+    {
+      // STA feedback: ignore
+      return;
+    }
   NS_FATAL_ERROR ("Don't know how to handle frame (type=" << hdr->GetType ());
 }
```

Código C.17: 2.1.3. Parche `regular-wifi-mac.cc`

```
@@ -70,6 +70,10 @@ ApWifiMac::GetTypeId (void)
                    MakeBooleanAccessor (&ApWifiMac::SetBeaconGeneration,
                                         &ApWifiMac::GetBeaconGeneration),
                    MakeBooleanChecker ())
+    .AddAttribute ("SessionTimeout", "The client inactivity time accepted (bigger than 1500↩
     ms).",
+                   TimeValue (MilliSeconds (2000)),
+                   MakeTimeAccessor (&ApWifiMac::m_sessionTimeout),
+                   MakeTimeChecker ())
   ;
   return tid;
 }
@@ -83,6 +87,7 @@ ApWifiMac::ApWifiMac ()
   m_beaconDca->SetMaxCw (0);
   m_beaconDca->SetLow (m_low);
   m_beaconDca->SetManager (m_dcfManager);
+  m_feedbackEnabled = false;

   // Let the lower layers know that we are acting as an AP.
   SetTypeOfStation (AP);
@@ -93,6 +98,8 @@ ApWifiMac::ApWifiMac ()
 ApWifiMac::~ApWifiMac ()
 {
   NS_LOG_FUNCTION (this);
+  m_macFrom.clear();
+  m_feedbackTimeout.clear();
 }

 void
@@ -455,6 +462,10 @@ ApWifiMac::TxOk (const WifiMacHeader &hdr)
   if (hdr.IsAssocResp ()
       && m_stationManager->IsWaitAssocTxOk (hdr.GetAddr1 ()))
     {
+      m_macFrom.push_back(hdr.GetAddr1());
+      m_feedbackTimeout.push_back(Seconds(0.0));
+      StaFeedback(hdr.GetAddr1 ());
+
       NS_LOG_DEBUG ("associated with sta=" << hdr.GetAddr1 ());
       m_stationManager->RecordGotAssocTxOk (hdr.GetAddr1 ());
     }
@@ -481,6 +492,21 @@ ApWifiMac::Receive (Ptr<Packet> packet, const WifiMacHeader *hdr)

   Mac48Address from = hdr->GetAddr2 ();

+  if(hdr->IsStaFeedback() && m_stationManager->IsAssociated (from))
+    {
+      if(m_feedbackEnabled == false)
+        m_feedbackEnabled = true;
+
+      uint32_t index = 0;
+      for(uint32_t j = 0; j < m_macFrom.size(); j++)
+        {
+          if(m_macFrom.at(j)==from)
+            index = j;
+        }
```

```
+      m_feedbackTimeout.at(index) = Simulator::Now ();
+      return;
+    }
+
   if (hdr->IsData ())
     {
       Mac48Address bssid = hdr->GetAddr1 ();
@@ -706,4 +732,26 @@ ApWifiMac::DoInitialize (void)
   RegularWifiMac::DoInitialize ();
 }

+void
+ApWifiMac::StaFeedback (Mac48Address from)
+{
+  uint32_t index = 0;
+  for(uint32_t i = 0; i < m_macFrom.size(); i++)
+    {
+      if(m_macFrom.at(i)==from)
+        index = i;
+    }
+  if(Simulator::Now ().GetSeconds() - m_feedbackTimeout.at(index).GetSeconds() < ↩
+   m_sessionTimeout.GetSeconds() || !m_stationManager->IsAssociated(from))
+    {
+      Simulator::Schedule (m_sessionTimeout, &ApWifiMac::StaFeedback, this, from);
+    }
+  else if(m_feedbackEnabled)
+    {
+      NS_LOG_DEBUG (GetAddress() <<  " RecordDisassociated " << from);
+      m_stationManager->RecordDisassociated (from);
+      m_feedbackTimeout.erase(m_feedbackTimeout.begin()+index);
+      m_macFrom.erase(m_macFrom.begin()+index);
+    }
+}
+
 } // namespace ns3
```

Código C.18: 2.1.3. Parche `ap-wifi-mac.cc`

```
@@ -204,12 +204,18 @@ private:
   virtual void DoDispose (void);
   virtual void DoInitialize (void);

+  void StaFeedback (Mac48Address from);
+
   Ptr<DcaTxop> m_beaconDca; //!< Dedicated DcaTxop for beacons
   Time m_beaconInterval; //!< Interval between beacons
   bool m_enableBeaconGeneration; //!< Flag if beacons are being generated
   EventId m_beaconEvent; //!< Event to generate one beacon
   Ptr<UniformRandomVariable> m_beaconJitter; //!< UniformRandomVariable used to randomize ↩
       the time of the first beacon
   bool m_enableBeaconJitter; //!< Flag if the first beacon should be generated at random ↩
       time
+  std::vector <Time> m_feedbackTimeout;
+  std::vector <Mac48Address> m_macFrom;
+  bool m_feedbackEnabled;
+  Time m_sessionTimeout;
 };

 } // namespace ns3
```

Código C.19: 2.1.3. Parche `ap-wifi-mac.h`

```
@@ -481,9 +481,19 @@ DcaTxop::NotifyAccessGranted (void)
               params.DisableRts ();
               NS_LOG_DEBUG ("tx unicast");
             }
-          params.DisableNextData ();
-          Low ()->StartTransmission (m_currentPacket, &m_currentHdr,
-                                     params, m_transmissionListener);
+
+          if(m_stationManager->IsAssociated(m_currentHdr.GetAddr1 ()) || m_currentHdr.IsMgt↩
    ())
+            {
+              params.DisableNextData ();
+              Low ()->StartTransmission (m_currentPacket, &m_currentHdr,
+                  params, m_transmissionListener);
+            }
+          else
+            {
+              m_queue->Flush ();
+              m_currentPacket = 0;
+              NS_LOG_DEBUG ("tx unicast cancelled: destination device is not associated");
+            }
         }
     }
 }
```

Código C.20: 2.1.3. Parche `dca-txop.cc`

```
@@ -495,9 +495,19 @@ EdcaTxopN::NotifyAccessGranted (void)
               NS_LOG_DEBUG ("tx unicast");
             }
           params.DisableNextData ();
-          m_low->StartTransmission (m_currentPacket, &m_currentHdr,
-                                    params, m_transmissionListener);
-          CompleteTx ();
+
+          if(m_stationManager->IsAssociated(m_currentHdr.GetAddr1 ()) || m_currentHdr.↩
    IsToDs())
+            {
+              m_low->StartTransmission (m_currentPacket, &m_currentHdr,
+                  params, m_transmissionListener);
+              CompleteTx ();
+            }
+          else
+            {
+              m_queue->Flush ();
+              m_currentPacket = 0;
+              NS_LOG_DEBUG ("tx unicast cancelled: destination device is not associated");
+            }
         }
     }
 }
```

Código C.21: 2.1.3. Parche `edca-txop-n.cc`

Bibliografía

[1] Wikipedia: *Ieee 802.11*. http://en.wikipedia.org/wiki/IEEE_802.11, Última modificación: 18 de Febrero de 2014.

[2] *Ieee standard for information technology–telecommunications and information exchange between systems local and metropolitan area networks–specific requirements part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications*. IEEE Std 802.11-2012 (Revision of IEEE Std 802.11-2007), pages 1–2793, 2012.

[3] S. Lakshmanan, S. Sanadhya, and R. Sivakumar: *On link rate adaptation in 802.11n wlans*. In *INFOCOM, 2011 Proceedings IEEE*, pages 366–370, April 2011.

[4] Weihua Helen Xi, Alistair Munro, and Michael Barton: *Link adaptation algorithm for the ieee 802.11n mimo system*. In *Proceedings of the 7th International IFIP-TC6 Networking Conference on AdHoc and Sensor Networks, Wireless Networks, Next Generation Internet*, NETWORKING'08, pages 780–791, Berlin, Heidelberg, 2008. Springer-Verlag, ISBN 3-540-79548-0, 978-3-540-79548-3. http://dl.acm.org/citation.cfm?id=1792514.1792601.

[5] Qiuyan Xia and M. Hamdi: *Smart sender: a practical rate adaptation algorithm for multirate ieee 802.11 wlans*. Wireless Communications, IEEE Transactions on, 7(5):1764–1775, May 2008, ISSN 1536-1276.

[6] Daji Qiao, Sunghyun Choi, Amit Jain, and Kang G. Shin: *Miser: An optimal low-energy transmission strategy for ieee 802.11a/h*. In *Proceedings of the 9th Annual International Conference on Mobile Computing and Networking*, MobiCom '03, pages 161–175, New York, NY, USA, 2003. ACM, ISBN 1-58113-753-2. http://doi.acm.org/10.1145/938985.939003.

[7] Vishnu Navda, Ravi Kokku, Samrat Ganguly, and Samir Das: *Slotted symmetric power control in managed wireless lans*, 2007. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.127.4654&rep=rep1&type=pdf.

[8] Eun Sun Jung and Nitin H. Vaidya: *A power control mac protocol for ad hoc networks*. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, MobiCom '02, pages 36–47, New York, NY, USA, 2002. ACM, ISBN 1-58113-486-X. http://doi.acm.org/10.1145/570645.570651.

[9] Kishore Ramachandran, Ravi Kokku, Honghai Zhang, and Marco Gruteser: *Symphony: Synchronous two-phase rate and power control in 802.11 wlans*. In

Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services, MobiSys '08, pages 132–145, New York, NY, USA, 2008. ACM, ISBN 978-1-60558-139-2. http://doi.acm.org/10.1145/1378600.1378616.

[10] Mathieu Lacage and Thomas R. Henderson: *Yet another network simulator.* In *Proceeding from the 2006 Workshop on Ns-2: The IP Network Simulator*, WNS2 '06, New York, NY, USA, 2006. ACM, ISBN 1-59593-508-8. http://doi.acm.org/10.1145/1190455.1190467.

[11] Mathieu Lacage, Mohammad Hossein Manshaei, and Thierry Turletti: *Ieee 802.11 rate adaptation: A practical approach.* In *Proceedings of the 7th ACM International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, MSWiM '04, pages 126–134, New York, NY, USA, 2004. ACM, ISBN 1-58113-953-5. http://doi.acm.org/10.1145/1023663.1023687.

[12] Ad Kamerman and Leo Monteban: *Wavelan®-ii: a high-performance wireless lan for the unlicensed band.* Bell Labs Technical Journal, 2(3):118–133, 1997, ISSN 1538-7305. http://dx.doi.org/10.1002/bltj.2069.

[13] Jongseok Kim, Seongkwan Kim, Sunghyun Choi, and D. Qiao: *Cara: Collision-aware rate adaptation for ieee 802.11 wlans.* In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–11, April 2006.

[14] Starsky H. Y. Wong, Hao Yang, Songwu Lu, and Vaduvur Bharghavan: *Robust rate adaptation for 802.11 wireless networks.* In *Proceedings of the 12th Annual International Conference on Mobile Computing and Networking*, MobiCom '06, pages 146–157, New York, NY, USA, 2006. ACM, ISBN 1-59593-286-0. http://doi.acm.org/10.1145/1161089.1161107.

[15] Andrew Myles Brian Hart: *Renewing the viability of 2.4 ghz by actively encouraging the use of ofdm rates*, 2014. https://mentor.ieee.org/802.11/dcn/14/11-14-0099-00-000m-renewing-2-4ghz-band.pptx.

[16] Rohde and Schwarz: *802.11ac technology introduction white paper," march 2012. [online].* In *M. Park, "IEEE 802.11ac: Dynamic Bandwidth Channel Access," in Proc. of IEEE ICC*, 2011. http://cdn.rohde-schwarz.com/dl_downloads/dl_application/application_notes/1ma192/1MA192_7e_80211ac_technology.pdf.

[17] G. Pei and T.R. Henderson: *Validation of ofdm error rate model in ns-3.* Boeing Research Technology, pages 1–15, 2010. http://www.nsnam.org/~pei/80211ofdm.pdf.