ANEXOS

Anexo A

Simulador de red ns-3

En este anexo vamos a explicar en detalle el simulador de red ns-3, en el que hemos desarrollado el algoritmo propuesto en este trabajo y donde hemos realizado todas las pruebas de funcionamiento del mismo.

En primer lugar, realizaremos una breve introducción a ns-3. Tras esto, analizaremos sus características principales y explicaremos cuál es su estructura en detalle, centrándonos en los módulos utilizados para el desarrollo de las simulaciones realizadas en este trabajo. Posteriormente, explicaremos cómo crear un escenario similar a los utilizados en el Capítulo 4, razonando paso a paso las estructuras a añadir.

A.1 Introducción a ns-3

El simulador ns-3 es un simulador de redes basado en eventos discretos, que se emplea principalmente para investigación y educación. El proyecto ns-3 se inició en el año 2006, y es un proyecto de desarrollo de código abierto bajo licencia GNU GPLv2.

ns-3 proporciona una plataforma de simulación implementada principalmente en C++, aunque algunas de sus estructuras están escritas en Python. Es compatible con Linux, Mac OS y FreeBSD.

Este simulador proporciona modelos para trabajar con paquetes de datos y redes, además de incorporar modelos de movilidad y propagación. Todo ello forma un motor de simulación en el que los usuarios pueden programar sus simulaciones, de forma que se pueden realizar estudios sobre el comportamiento de los sistemas cuando no se dispone de los dispositivos físicos.

Al estar organizado como una librería y estar escrito en C++ permite realizar enlaces estáticos y dinámicos. Una de sus características más importantes y que más se ha utilizado en este trabajo es el hecho de permitir la definición de nuevas topologías, escenarios y modelos de red, que pueden ser fácilmente integrados y depurados en el simulador. Todo ello se organiza en módulos, ya que de esta forma se facilita la independencia entre los diferentes niveles de la arquitectura de ns-3.

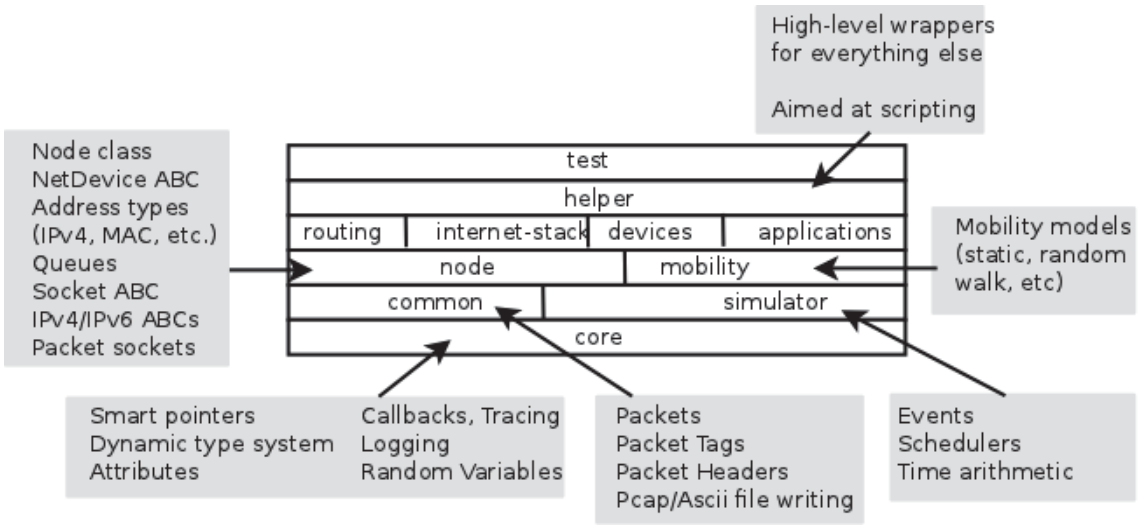

Figura A.1: Módulos existentes en ns-3 ***[15]***.

El código fuente de ns-3 se encuentra en su mayoría en el directorio src. Como se aprecia en la figura A.1, los módulos solo tienen dependencias de los módulos que están sobre ellos.

El núcleo del simulador (core) lo forman los componentes que son comunes en todos los protocolos, hardware y modelos de entorno. El núcleo esta implementado en src/core. Los paquetes son objetos fundamentales en un simulador de red, y están implementados en src/network. Estos dos módulos por sí mismos forman el núcleo de un simulador genérico que puede usarse en diferentes tipos de redes. Los módulos que aparecen por encima de estos dos módulos son independientes de los modelos de red o de los dispositivos de red, y cubren los diferentes módulos del simulador.

En el siguiente apartado analizaremos los módulos empleados en el desarrollo de los escenarios de este trabajo.

A.2 Módulos de ns-3 utilizados

Aunque ns-3 se compone de gran variedad de módulos, en este apartado veremos con más detalle los que han sido utilizados en el desarrollo del trabajo y se consideran básicos. Estos son los módulos que abarcan WiFi y los elementos de red básicos de ns-3.

Módulo network

Este módulo se compone de los elementos básicos de red, como vimos anteriormente, para que puedan proveer de funcionalidades básicas a cualquier topología y simulación. Este módulo contiene los elementos: Packets, NetDevices, Node, etc. Esta sección se centrará en los elementos de los nodos y dispositivos de red.

- Node: se trata de la clase fundamental para diseñar los escenarios, ya que es la que crea los elementos que colocaremos en dichos escenarios. A estos elementos luego se les asocian diferentes características, como veremos dentro del módulo WiFi.

Módulo WiFi

Los nodos de ns-3 pueden contener una variedad de diferentes objetos NetDevice, y en ambos casos uno de esos objetos/interfaces puede ser WiFi. En esta sección se describe el objeto WifiNetDevice y cómo se crean las redes 802.11.

El módulo wifi para ns-3 incluye diversas características y prestaciones, tales como: diferentes modelos de pérdidas de propagación, diferentes implementaciones de capa física, de estándar, de tipo de AP (con calidad de servicio o sin ella), etc.

Al crear una red WiFi, necesitamos los siguientes elementos:

- **YansWifiChannelHelper**: Este Helper es el encargado de crear un canal WiFi con un modelo de pérdidas de propagación por defecto. El modelo de pérdidas de propagación es el que hemos analizado en el capítulo 4, LogDistancePropagationLossModel.

- **YansWifiPhyHelper**: Esta clase crea un objeto que creará instancias de la capa física, pudiendo añadirle un modelo de movilidad (MobilityModel).

- **NqosWifiMacHelper** y **QosWifiMacHelper**: ambas clases sirven para crear instancias de un objeto ns3::WifiMac, los cuales son configurados con parámetros típicos como la clase de MAC. La primera de ellas lo configura de tal manera que no existe calidad de servicio, mientras que en la segunda sí. Es por esto que nosotros en nuestro trabajo utilizaremos QosWifiMacHelper.

- **WifiHelper**: Es la clase que se encarga de crear los WifiNetDevices que asociaremos a los nodos. Para crearlos es necesario haber creado tanto la capa física como la capa MAC.

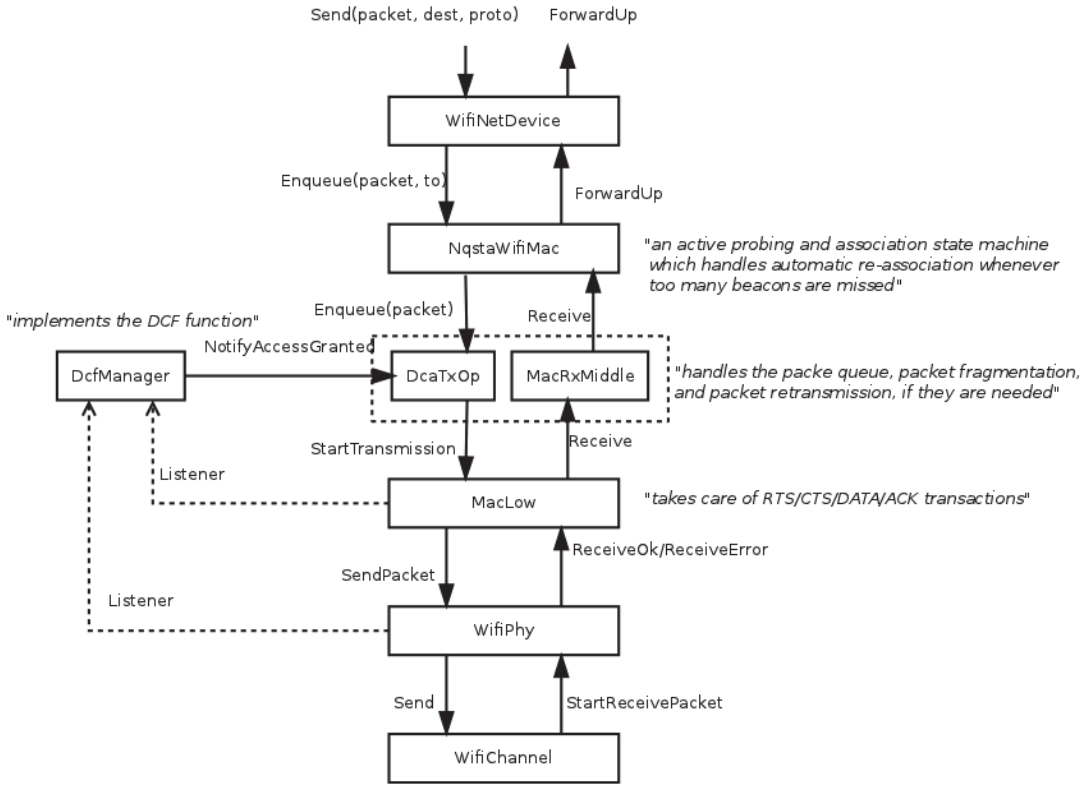

*Figura A.2: Arquitectura de un WifiNetDevice **[16]**.*

Dentro de este módulo se encuentran los elementos clave de nuestro trabajo: los parámetros característicos del protocolo EDCA. Dentro de cada nodo definido como un AP, podemos obtener sus parámetros EDCA (de cada una de las colas AC) de la siguiente manera:

```
Ptr<NetDevice> dev = node ->GetDevice(0);
Ptr<WifiNetDevice> wifi_dev = DynamicCast<WifiNetDevice>(dev);
Ptr<WifiMac> mac = wifi_dev->GetMac();
PointerValue ptr;

Ptr<EdcaTxopN> edca;

mac->GetAttribute("BK_EdcaTxopN", ptr);
edca = ptr.Get<EdcaTxopN>();
      edca->GetMinCw();
      edca->GetMaxCw();

mac->GetAttribute("BE_EdcaTxopN", ptr);
      edca = ptr.Get<EdcaTxopN>();
      edca->GetMinCw();
      edca->GetMaxCw();

mac->GetAttribute("VI_EdcaTxopN", ptr);
      edca = ptr.Get<EdcaTxopN>();
      edca->GetMinCw();
      edca->GetMaxCw();


mac->GetAttribute("VO_EdcaTxopN", ptr);
      edca = ptr.Get<EdcaTxopN>();
      edca->GetMinCw();
      edca->GetMaxCw();
```

Como podemos ver, es muy sencillo obtener los valores de los parámetros de las colas de cada AP, así como editarlos, como veremos en el siguiente capítulo, donde vamos a ir paso a paso definiendo el escenario a utilizar.

A.3 Creación del escenario en ns-3

Tras analizar las diferentes clases existentes en ns-3, vamos a ir explicando cómo crear un escenario completo, comenzando por crear los diferentes elementos de red y configurarlos para asignarles todos los parámetros necesarios para su funcionamiento. Posteriormente, debemos crear los flujos de tráfico de las diferentes estaciones asociadas al AP o del AP a las estaciones.

Finalmente, veremos cómo crear funciones para realizar los pasos necesarios por el algoritmo, como por ejemplo establecer los nuevos valores de los parámetros de las colas ***AC_BK*** u obtener los valores en ciertos instantes de tiempos del parámetro *CW*.

Vamos a poner como ejemplo un escenario sencillo, en el que exista un único AP con diferentes estaciones conectadas a él, en los que aparezca, por un lado, tráfico TCP de background, tanto descendente como ascendente y por otro lado tráfico VoIP descendente y ascendente (llamadas de voz).

En primer lugar, debemos crear los elementos necesarios (un AP y el número escogido de estaciones):

```
NodeContainer apNode1;
apNode1.Create (1);

int numNodos = ¿?;

NodeContainer wifiStaNodes;
wifiStaNodes.Create(numNodos);

NodeContainer wifiStaNodes1;

for (int i = 0; i< numNodos; i++){
      wifiStaNodes1.Add (wifiStaNodes.Get (i));
}

NodeContainer wifiApNode1 = apNode1.Get (0);
```

Ya tenemos creados los nodos que funcionarán como AP y como STA. Ahora debemos instalar el protocolo WiFi en todos ellos.

```
YansWifiChannelHelper channel = YansWifiChannelHelper::Default ();
YansWifiPhyHelper phy = YansWifiPhyHelper::Default ();

phy.Set("ChannelNumber",UintegerValue(1));
phy.SetChannel (channel.Create ());

Ssid ssid1 = Ssid ("primero");
WifiHelper wifi1 = WifiHelper::Default ();
QosWifiMacHelper mac_ap1 = QosWifiMacHelper::Default ();
QosWifiMacHelper mac_sta1 = QosWifiMacHelper::Default ();
```

```
wifi1.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
"DataMode", StringValue ("OfdmRate6Mbps"),  "ControlMode", StringValue
("OfdmRate6Mbps"));

mac_ap1.SetType ("ns3::ApWifiMac",  "Ssid", SsidValue (ssid1),
"BeaconGeneration", BooleanValue (true),  "BeaconInterval", TimeValue
(Seconds (1)));

mac_sta1.SetType ("ns3::StaWifiMac", "Ssid", SsidValue (ssid1),
"ActiveProbing", BooleanValue (false));

NetDeviceContainer staDevices1 = wifi1.Install (phy, mac_sta1,
wifiStaNodes1);
NetDeviceContainer apDevices1 =wifi1.Install(phy,mac_ap1, wifiApNode1);
```

Antes de instalar WiFi hemos tenido que crear la capa física y la capa MAC, como hemos explicado en el apartado A.2. Ahora vamos a establecer en qué posición del escenario los colocamos:

```
MobilityHelper mobility;

mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
                               "MinX", DoubleValue (47.5),
                               "MinY", DoubleValue (199.05),
                               "DeltaX", DoubleValue (1.0),
                               "DeltaY", DoubleValue (1.0),
                               "GridWidth", UintegerValue (6),
                               "LayoutType", StringValue ("RowFirst"));

mobility.Install (wifiStaNodes1);

mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
                               "MinX", DoubleValue (50),
                               "MinY", DoubleValue (201.55),
                               "DeltaX", DoubleValue (0),
                               "DeltaY", DoubleValue (0.0),
                               "GridWidth", UintegerValue (500),
                               "LayoutType", StringValue ("RowFirst"));

mobility.Install (wifiApNode1);
```

Ya los hemos posicionado. En este punto es cuando debemos asignarles el protocolo de internet y las direcciones IP necesarias:

```
InternetStackHelper stack;
stack.Install (wifiApNode1);
stack.Install (wifiStaNodes1);
Ipv4AddressHelper address;

address.SetBase ("10.1.4.0", "255.255.255.0");
Ipv4InterfaceContainer apInterfaces, staInterfaces;
staInterfaces = address.Assign (staDevices1);
apInterfaces = address.Assign (apDevices1);
```

Con esto finalizamos la configuración de todos los nodos existentes en la red. Ahora debemos crear el tráfico que genera o recibe cada uno de ellos:

Ejemplo de generación de tráfico TCP downlink (AP → STA)

```
  // TCP receiver

PacketSinkHelper
sink("ns3::TcpSocketFactory",InetSocketAddress("10.1.4.13", port));
ApplicationContainer apps_rec= sink.Install (wifiStaNodes2.Get(12));

apps_rec.Start(Seconds(1.0));
apps_rec.Stop(Seconds(100.0));


  // TCP Sender

OnOffHelper onoff("ns3::TcpSocketFactory",InetSocketAddress("10.1.4.1",
port));
onoff.SetAttribute("OnTime",
StringValue("ns3::ConstantRandomVariable[Constant=1]"));
onoff.SetAttribute("OffTime",
StringValue("ns3::ConstantRandomVariable[Constant=0]"));
onoff.SetAttribute("DataRate", StringValue("12Mbps"));
onoff.SetAttribute("PacketSize", StringValue("1500"));

ApplicationContainer apps_sen = onoff2a.Install(wifiApNode1);
Ptr<OnOffApplication> onoffappTCP;
onoffappTCP = DynamicCast<OnOffApplication>(apps_sen.Get(0));
onoffappTCP->TraceConnectWithoutContext("Tx",MakeBoundCallback
(&TagMarker, AC_BK));

apps_sen.Start(Seconds(1.0));
apps_sen.Stop(Seconds(100.0));
```

Para generar el tráfico, debemos crear un receptor y un emisor. Creamos el receptor en la estación que deseemos (en este caso en 10.1.4.13) y el emisor en el AP. Al OnOffHelper debemos decirle que es tráfico TCP mediante lo marcado en negrita. Tampoco debemos olvidarnos de marcar el tráfico como perteneciente a la cola ***AC_BK***, ya que esto es lo que lo marca como tráfico TCP de background.

Ejemplo de generación de tráfico TCP uplink (STA → AP)

```
// TCP Sender

OnOffHelper onoff1("ns3::TcpSocketFactory",InetSocketAddress
("10.1.4.30", port));
onoff1.SetAttribute("OnTime",
StringValue("ns3::ConstantRandomVariable[Constant=1]"));
onoff1.SetAttribute("OffTime",
StringValue("ns3::ConstantRandomVariable[Constant=0]"));
onoff1.SetAttribute("DataRate", StringValue("12Mbps"));
onoff1.SetAttribute("PacketSize", StringValue("1500"));
ApplicationContainer apps_sen1 =
onoff1.Install(wifiStaNodes1.Get(20));
Ptr<OnOffApplication> onoffappTCP1;
onoffappTCP1 = DynamicCast<OnOffApplication>(apps_sen1.Get(0));
onoffappTCP1->TraceConnectWithoutContext("Tx",  MakeBoundCallback
(&TagMarker, AC_BK));

apps_sen1.Start(Seconds(1.0));
apps_sen1.Stop(Seconds(100.0));

// TCP receiver

PacketSinkHelper
sink1("ns3::TcpSocketFactory",InetSocketAddress("10.1.4.30", port));
ApplicationContainer apps_rec1 = sink1.Install(wifiApNode1);

apps_rec1.Start(Seconds(1.0));
apps_rec1.Stop(Seconds(100.0));
```

Este caso es opuesto al anterior; debemos crear el emisor en la estación elegida y el receptor en el AP. También marcamos el tráfico como perteneciente a la cola ***AC_BK***.

Ejemplo de generación de tráfico UDP uplink y downlink (llamada VoIP):

```
// VoIP Sender Uplink
OnOffHelper senderVoIP_1up("ns3::UdpSocketFactory",
InetSocketAddress("10.1.4.30", 4000));
senderVoIP_1up.SetConstantRate(DataRate("64Kbps"), 160);
senderVoIP_1up.SetAttribute("OnTime",StringValue
("ns3::ConstantRandomVariable [Constant=1]"));
senderVoIP_1up.SetAttribute("OffTime",StringValue
("ns3::ConstantRandomVariable [Constant=0]"));
senderappVoIP_1up = senderVoIP_1up.Install(wifiStaNodes1.Get(0));

// VoIP Sender Downlink
OnOffHelper senderVoIP_1down("ns3::UdpSocketFactory",
InetSocketAddress("10.1.4.1", 4000)
senderVoIP_1down.SetConstantRate( DataRate("64Kbps") , 160);
senderVoIP_1down.SetAttribute("OnTime",StringValue
("ns3::ConstantRandomVariable [Constant=1]"));
senderVoIP_1down.SetAttribute("OffTime",StringValue
("ns3::ConstantRandomVariable [Constant=0]"));
senderappVoIP_1down=senderVoIP_1down.Install(apNode1.Get(0));

//VoIP Receiver Uplink
PacketSinkHelper receiverVoIP_1up("ns3::UdpSocketFactory",
InetSocketAddress ("10.1.4.30", 4000));
ApplicationContainer receiverappVoIP_1up=receiverVoIP_1up.Install
(apNode1.Get(0));

//VoIP Receiver Downlink
PacketSinkHelper receiverVoIP_1down("ns3::UdpSocketFactory",
InetSocketAddress("10.1.4.1", 4000));
ApplicationContainer receiverappVoIP_1down
=receiverVoIP_1down.Install(wifiStaNodes1.Get (0));


Ptr<OnOffApplication> down, up;
down = DynamicCast<OnOffApplication>(senderappVoIP_1down.Get(0));
up = DynamicCast<OnOffApplication>(senderappVoIP_1up.Get(0));

down->TraceConnectWithoutContext("Tx",MakeBoundCallback(&TagMarker,
AC_VO));
up->TraceConnectWithoutContext("Tx",MakeBoundCallback(&TagMarker,
AC_VO));

senderappVoIP_1down.Start(Seconds(1.0));
senderappVoIP_1down.Stop(Seconds(100.0));
senderappVoIP_1up.Start(Seconds(1.0));
senderappVoIP_1up.Stop(Seconds(100.0));
```

```
receiverappVoIP_1down.Start(Seconds(1.0));
receiverappVoIP_1down.Stop(Seconds(100.0));
receiverappVoIP_1up.Start(Seconds(1.0));
receiverappVoIP_1up.Stop(Seconds(100.0));
```

En este caso, debemos generar tráfico UDP (VoIP). Además, añadimos todo lo necesario para crear el tráfico en ambos sentidos en este código. Le asignamos la tasa de transmisión y el tamaño de paquete mediante el comando "SetConstantRate"

Ahora que ya tenemos el tráfico generado, debemos ser capaces de monitorizar los datos de todos los nodos de la red para poder pasárselos al algoritmo y que él decida lo que hacer. Para ello, instalamos en primer lugar un monitor de flujos:

```
FlowMonitorHelper fmHelper;
Ptr<FlowMonitor> allMon = fmHelper.InstallAll();
```

Ejecutamos el algoritmo y lo programamos para que se ejecute cada segundo desde el segundo 2:

```
Simulator::Schedule(Seconds(2), &Algoritmo, &fmHelper, allMon, apNodes,
algoritmoActivado);
```

El funcionamiento del algoritmo ha sido tratado en detalle en la memoria, por lo que aquí vamos a resumir el funcionamiento de las funciones que dicho algoritmo utiliza para obtener los datos o modificarlos:

En primer lugar, para obtener el throughput, el retardo medio de los paquetes y el porcentaje de paquetes perdidos de todas las colas realiza cálculos con los datos que obtiene de cada flujo:

Como queremos los datos cada segundo, hay que hacer ciertas operaciones para obtener los datos del intervalo de tiempo que queremos y no del total, que es lo que nos devuelve ns-3 por defecto.

```
throughput=(stats->second.rxBytes-BytesAntiguos) * 8.0 /
(intervalo_scheduler)/1024/1024;

delay=(stats->second.delaySum.GetSeconds()-DelayAntiguo)/
(stats->second.rxPackets-PaquetesAntiguos);

packet_loss=((stats->second.txPackets)-(stats->second.rxPackets))-
packet_loss_antiguo;
```

Cuando tenemos todos los datos necesarios, el algoritmo decide si debe modificar los parámetros de la cola ***AC_BK*** de cada AP. Si ha decidido que debe hacerlo, hay que acceder a dichos parámetros y modificarlos. Esto se hace de forma similar a lo que hemos visto en el apartado anterior para obtener los parámetros, pero utilizando funciones Set en lugar de Get.

Para esta labor se creó la siguiente función llamada SetCwMin:

```
void SetCwMin(Ptr<Node> node, int parametros[12])
{
	Ptr<NetDevice> dev = node ->GetDevice(1);
	Ptr<WifiNetDevice> wifi_dev = DynamicCast<WifiNetDevice>(dev);

	Ptr<WifiMac> mac = wifi_dev->GetMac();
	PointerValue ptr;

	Ptr<EdcaTxopN> edca;

	mac->GetAttribute("BK_EdcaTxopN", ptr);
	edca = ptr.Get<EdcaTxopN>();
	edca->SetMinCw(parametros[0]);
	edca->SetMaxCw(parametros[4]);

	mac->GetAttribute("BE_EdcaTxopN", ptr);
	edca = ptr.Get<EdcaTxopN>();
	edca->SetMinCw(parametros[1]);
	edca->SetMaxCw(parametros[5]);

	mac->GetAttribute("VI_EdcaTxopN", ptr);
	edca = ptr.Get<EdcaTxopN>();
	edca->SetMinCw(parametros[2]);
	edca->SetMaxCw(parametros[6]);

	mac->GetAttribute("VO_EdcaTxopN", ptr);
	edca = ptr.Get<EdcaTxopN>();
	edca->SetMinCw(parametros[3]);
	edca->SetMaxCw(parametros[7]);

}
```

La variable parametros[12] ya fue explicada en el Capítulo 3, y como recordamos contiene los parámetros que queremos modificar, ordenados por colas. Su estructura es la siguiente:

$$[CW_{minBK}, CW_{maxBK}, AIFSN_{BK}, CW_{minBE}, CW_{maxBE}, AIFSN_{BE},$$

$$CW_{minVI}, CW_{maxVI}, AIFSN_{VI}, CW_{minVO}, CW_{maxVO}, AIFSN_{VO}]$$

Finalmente, también creamos una función para obtener los valores de CW de la cola $\boldsymbol{AC\_BK}$ para luego poder mostrarlos en la gráfica. En este caso tuvimos además que editar el código fuente de ns-3, ya que no estaba creada ninguna manera de obtener dicho dato.

Ésta es la función desarrollada:

```
void GetCwGlobal(Ptr<Node> node_important, int numAP)
{
      Ptr<NetDevice> dev_important = node_important->GetDevice(1);
      Ptr<WifiNetDevice> wifi_dev = DynamicCast<WifiNetDevice>
      (dev_important);

      Ptr<WifiMac> mac = wifi_dev->GetMac();
      PointerValue ptr;

      Ptr<EdcaTxopN> edca;

      mac->GetAttribute("BK_EdcaTxopN", ptr);
      edca = ptr.Get<EdcaTxopN>();
      uint32_t CW = edca->GetCw();
}
```

Y éstos son los cambios que hubo que añadir en el código fuente para crear la función GetCw():

Archivo dcf-manager.cc: Añadimos las siguientes líneas:

línea 123

```
uint32_t
DcfState::GetCw (void) const
{
 return m_cw;
}
```

Archivo edca-txop-n.cc: Añadimos las siguientes líneas:

línea 408

```
uint32_t
EdcaTxopN::GetCw (void) const
{
  NS_LOG_FUNCTION (this);
  return m_dcf->GetCw ();
}
```

Archivo edca-txop-n.h: Añadimos las siguientes líneas:

línea 152

```
virtual uint32_t GetCw (void) const;
```