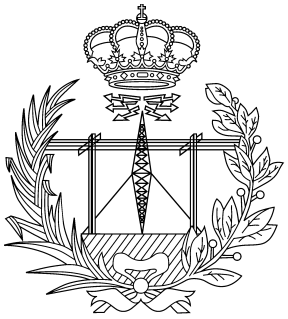
ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA



Trabajo Fin de Grado

IMPLEMENTACIÓN DE UN ENTORNO DE SIMULACIÓN DE REDES VRAN CON FRONTHAUL BASADO EN CONMUTACIÓN DE PAQUETES

(On the design and implementation of a simulation framework for vRAN and packet-switched fronthaul networks)

Para acceder al Título de

Graduado en
Ingeniería de Tecnologías de Telecomunicación

Autor: Claudia Abascal Gutiérrez

Septiembre- 2021

UNIVERSIDAD DE CANTABRIA

E.T.S. DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

CALIFICACIÓN DEL TRABAJO FIN DE GRADO

Realizado por: Claudia Abascal Gutiérrez
Director del TFG: Ramón Agüero Calvo, Luis Francisco Diez Fernández
Título: "IMPLEMENTACIÓN DE UN ENTORNO DE SIMULACIÓN DE REDES VRAN CON FRONTHAUL BASADO EN CONMUTACIÓN DE PAQUETES"
Title: "On the design and implementation of a simulation framework for vRAN and packet-switched fronthaul networks"

Presentado a examen el día: 28 de Septiembre del 2021

para acceder al Título de

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Composición del tribunal:
Presidente (Apellidos, Nombre): Crespo Fidalgo, José Luis
Secretario (Apellidos, Nombre): Diez Fernández, Luis Francisco
Vocal (Apellidos, Nombre): García Gutiérrez, Alberto Eloy

Este Tribunal ha resulto otorgar la calificación de: .....

Fdo.: El Presidente

Fdo.: El Secretario

Fdo.: El Vocal

Fdo.: El Director del TFG
(solo si es distinto del Secretario)

V^{o} B^{o} del Subdirector

Trabajo Fin de Grado N^{o}
(a asignar por Secretaría)

AGRADECIMIENTOS

A mi familia, en especial a mis padres, por apoyarme, animarme y estar siempre ahí en esta carrera de fondo. Y a ti Vero, por tu papel indiscutible como hermana mayor.

A Pablo, porque siempre has confiado y creído en mi, incluso ni cuando yo lo veía claro.

A Lara, por estos cuatro años juntas, tanto de carrera como de amistad.

A Andrea y Dani, por vuestro apoyo incondicional desde la barrera.

Y, por último, a mis tutores, Ramón y Luis, por la continua atención, disponibilidad y confianda que habéis depositado en mi durante todos estos meses de trabajo.

Resumen

Nos encontramos ante una realidad indiscutible cuando hablamos de la necesidad de realizar un cambio en la red de acceso radio que actualmente conocemos. Su ininterrumpido uso ha generado consigo un importante crecimiento del tráfico en las estaciones bases, provocando que estas, a su vez, se vean sobrecargadas. Por esta razón se estudian distintas soluciones, entre las cuales encontramos la idea de la virtualización. El objetivo de dicha virtualización es ejecutar y centralizar las funciones de banda base en centros de datos. Pero ¿es realmente conveniente realizar todas las funciones en los centros de datos? A dicha pregunta podríamos contestar sí, si hablamos de su bajo coste tanto económico como energético, su baja latencia y sus mayores capacidades, pero a su vez, implicaría una red fronthaul con altas prestaciones sin olvidar el inconveniente de la sobrecarga. Debido a ello, aparece la división funcional, donde se declina la idea de realizar el procesado en una única unidad centralizada para dar paso a una división de tareas entre las estaciones bases y el controlador.

Palabras clave: Red de Acceso Radio, estaciones bases, virtualización, funciones, centros de datos, red fronthaul, división funcional.

Abstract

We are faced with an indisputable truth when we talk about the need to make a change in the radio access network that we currently know. Their uninterrupted use has led to a significant increase in traffic at base stations, which in turn causes them to be overloaded. For this reason different solutions are studied, among which we find the idea of virtualization. The objective of such virtualization is to execute and centralize the database bandwidth functions in data centers. But is it really convenient to perform all functions in data centers? To this question we could answer yes, if we speak of its low cost both economic and energy, its low latency and its greater capacities, but in turn, it would imply a fronthaul network with high performance without forgetting the inconvenience of the overload. Due to this, the functional division appears, where the idea of processing in a single centralized unit is declined to give way to a division of tasks between the base stations and the controller.

Keywords: Radio access network, base stations, virtualization, functions, data centers, fronthaul network, functional division.

Índice general

Índice de figuras

Índice de tablas

Índice de acrónimos

Números

3GPP 3rd Generation Partnership Project.

4G Cuarta Generación.

5G Quinta Generación.

B

BBU Baseband Unit.

BP Backpressure.

BS estación base.

C

C-RAN Centralized Radio Access Networks.

CU Central Unit.

D

DU Distributed Unit.

E

eMBB enhanced mobile broadband.

ETSI Instituto Europeo de Normas de Telecomunicaciones.

G

GIT Grupo de Ingeniería Telemática.

H

HetSNets HetSNets.

L

LTE Long Term Evolution.

M

MEC Mobile Edge Computing.

MIMO Multiple InputMultiple Output.

mMTC massive machine-type communications.

N

NFV Network function virtualization.

P

PRBs Physical Resource Block.

Q

QoS Quality-of-Service.

R

RF Radio frequency.

RR Round Robin.

RRH Remote Radio Head Unit.

RRU Remote Radio Unit.

RU Remote Unit.

S

SDN Software-defined networking.

T

TIC Tecnologías de la información.

U

URLLC ultra-reliable low latency communications.

W

WFQ Weighted Fair Queuing.

Capítulo 1

Introducción

El inminente asentamiento de la Quinta Generación de las tecnologías de redes móviles (5G) trae consigo velocidades de conectividad más rápidas, latencia ultrabaja, mayor seguridad, conexión de multitud de dispositivos y un mayor ancho de banda[1]. Además de ello, la demanda de datos por los servicios de banda ancha móvil mejorados (embb), las comunicaciones de baja latencia ultra confiables (URLLC) y las comunicaciones masivas del tipo maquina (mMTC) implican un cambio, una reestructuración, en la red que hoy en día conocemos.[2]

El primer tipo de servicios incluidos en el IMT-2020 es el eMBB. Estos servicios se centran en la utilización de la red por parte del usuario para el acceso a contenido multimedia, realidad virtual, o juegos en red de forma ubicua.

Por su parte, los servicios URLLC, tiene requisitos muy estrictos en cuanto a capacidades tales como el caudal garantizado, la latencia y la disponibilidad. Dentro de este tipo de servicios se incluirían los relacionados con tele-medicina, o conducción autónoma.

Y, por último, en el caso de la utilización mMTC se caracteriza por un elevado número de dispositivos conectados que transmiten un volumen relativamente bajo de datos, y en general menos sensibles al retardo. Bajo esta definición se incluirían las redes de sensores, cuyos dispositivos suelen ser de bajo coste y con una prolongada duración de la batería.

A fin de proporcionar los requisitos demandados por este tipo de servicios, se tuvo que replantear la arquitectura de la red, y muy especialmente de la red de acceso que tradicionalmente es el segmento con mayores limitaciones. En primer lugar, se vio la necesidad de incrementar de manera notable el número de puntos de acceso y su coordinación, lo que suponía un coste elevado de la red. Para solventar este inconveniente se propuso cambiar la red de acceso de arquitecturas distribuidas, donde las funcionalidades (unidad de banda base) de las estaciones base se situaban junto con la antena, a arquitecturas centralizadas. En estas soluciones, haciendo uso de la capacidad de virtualización, las funcionalidades de las estaciones base se centralizan en un punto alejado de las antenas. Estas primeras soluciones reciben el nombre de Cloud Radio Access Network (C-RAN)[3].Esta tecnología ofrece una alternativa mejor organizada y más

eficiente, algo totalmente necesario ante el crecimiento de requisitos de latencia y tráfico [4]. Gracias a la tecnología de computación en la nube los objetivos de mayores capacidades con menores retardos se ven alcanzados.

Como se ha indicado, la arquitectura C-RAN consiste en transportar el procesado en banda base a un centro de procesado de datos en el que se encuentran varias Base Band Unit (BBU) virtualizadas y centralizadas. La red se encuentra dividida entre las BBUs y los cabezales de radio remotos, Remote Radio Head (RRH), unidas ambas partes por la red fronthaul que los conecta. Esta centralización, además de abaratar los RRH (que son elementos relativamente sencillos), permite una estrecha cooperación de las estaciones base, cuyas BBU están instanciadas en un mismo punto.

Sin embargo, las soluciones C-RAN imponen requisitos muy altos a la red de *fronthaul* que en la práctica requiere que todas las RRH posean un enlace de fibra óptica[5]. A su vez esto implica la realización de obra civil y un aumento de los costes. Por ello se propusieron soluciones intermedias entre las arquitecturas totalmente distribuidas tradicionales y las soluciones C-RAN.

Por todo ello y basándose en el funcionamiento de la C-RAN con su parte virtualizada y centralizada surge la idea de efectuar el reparto de las funcionalidades de la BBU a realizar en la entidad virtualizada y en las RRHs. A este nuevo planteamiento se le conoce como functional split o división funcional. Para evitar confusión en la nomenclatura, en este nuevo esquema la entidad virtualizada y centralizada se denomina Central Unit (CU), que implementa parte las funciones de la BBU. Por otro lado, el resto de funciones se instancian en la Distributed Unit (DU) que se encuentra cerca o en el mismo lugar que la RRH, que por uniformidad recibe el nombre de Radio Unit (RU). El funcionamiento ideal de esta nueva arquitectura sería concentrar el máximo número de funcionalidades en la CU y el resto en la DU para maximizar la tasa de datos, pero hay que tener en cuenta que debido a la red frontahul que conecta ambas entidades esto no podría ser realizable por los factores mencionados anteriormente. Por lo que aparece una nueva incógnita, la forma de realizar el reparto de funcionalidades más óptimo entre las CU y DU.

Con esta nueva incógnita aparecen nuevos métodos para afrontarlo. La solución actual, consiste en determinar el split de forma estática en base a una serie de estudios como puede ser el tráfico, el número de usuarios, las interferencias etc. En este caso el nivel de centralización se variará al reconfigurar la red para prestar nuevos servicios.

Por otro lado, útimamente se ha planteado la posibilidad de realizar el cambio de split de forma dinámica, dando lugar al dynamic functional split. Este, consiste en una selección del nivel de split en función del estado de la red en ese momento, así como de los requisitos de los servicios prestados. Este tipo de selección puede basarse en diferentes parámetros, como pueden ser: los retardos, la carga de la red, el consumo energético, la eficiencia, etc.

1.1. Objetivos

Este trabajo se centra en la última de las soluciones mencionadas *dynamic functional split*, y aborda los siguientes objetivos:

- Modelado de la red de acceso, incluyendo estaciones base y red fronthaul, como un sistema de colas.
- Implementación de un entorno de simulación desde cero, basado en el modelado anterior, que permita la evaluación de esquemas de selección de split de forma dinámica.
- Implementación de varias métricas de rendimiento independientes del esquema de selección para poder compararlos de forma justa.
- En este entorno se han implementado y validado varios algoritmos de selección de split. De ellos nos hemos centrado en una solución basada en *backpressure*, mientras que los otros algoritmos se han tomado como referencia de rendimiento.

1.2. Estructura del documento

El resto del documento se estructura de la siguiente manera. En el Capítulo 2 se presentan los conceptos utilizados relacionados con C-RAN, functional split y los algoritmos realizados. Seguidamente, los detalles del entorno de simulación se describen en el Capítulo 3. A continuación, en el Capítulo 4 se presentan los resultados obtenidos en varios escenarios para validar las implementaciones realizadas y comparar los algoritmos. Por último, en el Capítulo 5 se resumen las conclusiones del trabajo y se enumeran algunas líneas futuras.

Capítulo 2

Antecedentes

La llegada de la Quinta Generación (5G) y las futuras generaciones que le seguirán traen consigo una serie de requisitos muy estrictos que plantean una reestructuración de la red tal y como la conocemos para ofrecer de forma eficiente los servicios necesarios. El objetivo de las redes 5G es claro: proporcionar servicios heterogéneos con características y requisitos variables. [6]

Ante este reto, se realizaron numerosos esfuerzos para mejorar los sistemas de QoS ,*(Quality-of-Service)* o MEC *(Mobile Edge Computing)* pero a largo plazo, la solución definitiva pasaba por incrementar la capacidad física de la red, lo cual nos llevaría directamente a la fibra como solución del problema. El inconveniente de esto reside en el presupuesto que conllevaría desplegar dichas redes, además de los largos plazos de tiempo que implicaría dejando muchas zonas sin su instalación por su complicada localización. [7]

Uno de los avances más notables que se han alcanzado cuando hablamos de la reestructuración de la red se corresponde con la capacidad de virtualizar y reasignar funciones de red aprovechando las técnicas de *virtualización de funciones de red definidas por software (SDN) y funciones de red (NFV)*[6]. Esto implica que las funcionalidades de las estaciones base se dividen, pasando a ser algunas de ellas virtualizadas.

Inicialmente, junto a la primera y segunda generación de tecnologías de telefonía móvil todas las funcionalidades de banda base y radio estaban centralizadas en la estación base. [8]

Seguidamente, con la llegada de la tercera generación apareció la idea de separar la Estación Base (BS) en dos unidades. Por un lado, encontramos la división en unidad de radio, *Remote Radio Head (RRH)* o *Remote Radio Unit(RRU)* y en la unidad de procesado base, *Base Band Unit (BBU)*. El RRH contiene la antena y las funciones de radio mientras que el BBU contenía todas las funciones de procesado de banda base. Cada par RRH y BBU se conectaban usando un nuevo segmento de red a lo que se le denominó red *fronthaul*.

Con la implantación de las redes de cuarta generación, 4G, se introdujo el concepto *Cloud Radio Access Network*, C-RAN, donde las BBUs se centralizaron en una única ubicación, *BBU-pool*. Ello implicaba que todo el BBU se virtualizaba mientras que un cabezal de radio remoto (RRH) realizaba las funciones básicas de radio frecuencia (RF). Esta solución daba pie a una estrecha coordinación de los puntos de acceso, pero a su vez imponía una serie de requisitos muy exigentes en los enlaces fronthaul que conectaban el RRH y el BBU[5].

La red fronthaul define unos requisitos muy estrictos no solo en términos de velocidad de datos, sino también en términos de latencia, jitter, tasa de error de bits [9] y la limitación a la hora de hablar sobre la capacidad de centralización. Estas condiciones deberían ser abordadas por los proveedores de servicio que tendrían que hacer frente a los elevados costes que ello supone para alcanzar una red rentable con un funcionamiento eficiente y una estrategia de evolución. Por lo tanto, para mantener los costes de los operadores bajo control, mantener el aumento de tráfico y para minimizar el impacto de las *Tecnologías de la información, TIC* en el medio ambiente, se empezó a prestar mayor atención al hecho de limitar el consumo de energía en el aumento de la red.[10] [11]

Esta, no fue la única medida que se tomó, se empezó a investigar la posibilidad de reducir la tasa de bits en los enlaces manteniendo al mismo tiempo los beneficios que la C-RAN tradicional aportaba. El planteamiento que se alcanzó fue el de procesar la señal de forma más exhaustiva antes de que esta se trasmitiera incluyendo, también, más funciones localmente. Pero ello, apareció una nueva cuestión, ¿cuántas funciones han de permanecer localmente y cuántas han de ser enviadas a los centros de datos? Ante esta pregunta, los principales organismos de normalización como es el Instituto Europeo de Normas de Telecomunicaciones (ETSI), empezó a introducir el término virtualización como solución al problema y en concreto el término *división funcional* o *functional split.*[12][13][14]

La división funcional es capaz de determinar la cantidad de funciones que permanecen localmente en la antena y la cantidad de funciones centralizadas en el centro de procesado datos.

El uso de las divisiones funcionales conlleva la división del BBU en una unidad distribuida (DU) situada cerca del RRH, a lo que ahora conoceremos como unidad radio (RU), y una unidad central (CU) que se encontrará cerca del borde de la red de agregación. Cuantas más funciones se encuentren en la RU, más procesado se habrá hecho antes de que los datos atraviesen la red fronthaul y más liberada quedará esta. Esto permitirá mantener la mayoría de las ventajas de la centralización a la vez que se respetan los requisitos de la red.

En el reporte técnico [15] el GPP muestra la Figura 2.1 representando ocho posibles divisiones funcionales fijas entre la unidad central y la distribuida y aportando una breve descripción de cada una de ellas, así como sus casos de uso. Como se puede ver en la Figura 2.1, las opciones de split se encuentran en los límites de los protocolos presentes en la estación base. Asimismo, se han propuesto opciones intra-protocolo para la mayoría de ellos (RLC, MAC, y PHY).

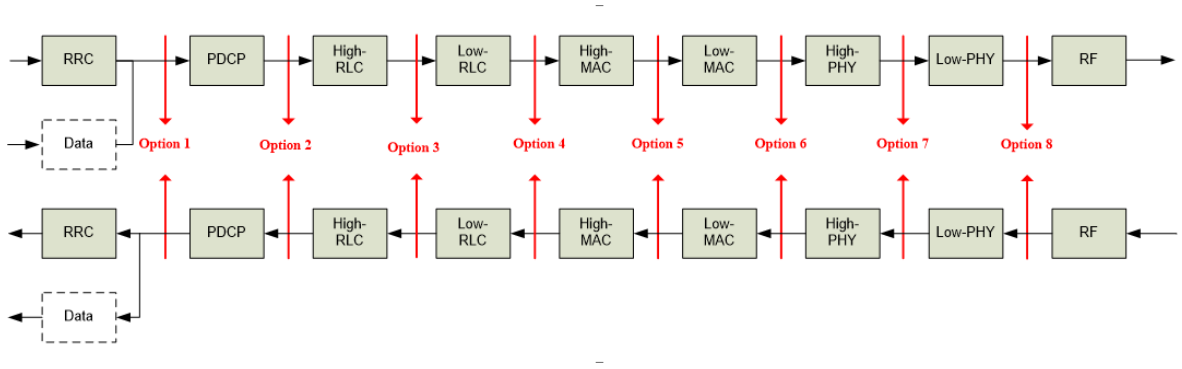


Figura 2.1: División Funcional entre la unidad central y la unidad distribuida [15]

Aunque en la actualidad la selección de split se contempla como una configuración de red, tanto la comunidad investigadora como iniciativas industriales, entre las que destacan Open Radio Access Network (O-RAN), están empezando a analizar la posibilidad de una división funcional flexible[16]. Esta es capaz de aportar soluciones escalables y rentables, permite la coordinación de características de rendimiento, gestión de carga, optimización de rendimiento en tiempo real y la posibilidad de adaptación en los cambios que presente la red.

En este trabajo, el sistema vRAN se ha modelado como un sistema de colas. Para ello se parte de la idea de que la cantidad de paquetes que la CU o DU pueden drenar (enviar hacia los usuarios) depende de la opción de *split*. Por ejemplo, una opción centralizada acumulará muchas funciones en la CU, por lo que una parte importante de la capacidad de cómputo se dedicará a hacer transitar los paquetes entre los diferentes protocolos (procesado, actualización de estado, escritura de cabeceras, etc.). Por el contrario, en una opción distribuida la CU estará más descargada de tareas, por lo que podrá dedicar su capacidad de cómputo únicamente a enviar paquetes. De esta manera, podemos pensar en la CU como varias colas en paralelo, cada una de ellas representado un *split*, conectadas a enlaces de diferentes capacidades en función del *split* que modelen. Un razonamiento similar se puede seguir para modelar la DU. En los capítulos posteriores se verá este modelado con ejemplos concretos.

Como se ha comentado, durante la realización del simulador se utilizarán una serie de algoritmos que permitirán la selección de split y se evaluarán mediante diferentes métricas de rendimiento. Por ello, a continuación, se hace una breve descripción de los algoritmos utilizados.

2.1. Algoritmos

En el desarrollo aparecen tres algoritmos diferentes. Encontraremos por una parte el algoritmo Round Robin y el Weighted Fair Queuing que poseen in comportamiento fijado independiente del estado de la red. Y, por otro lado, el algoritmo Max-weight backpressure cuyo comportamiento viene dado por el estado de la red en el momento de tomar la decisión. Este algoritmo se enmarca dentro de los conocidos como algoritmos oportunistas. En todos los casos se asume que el tiempo

está ranurado y que se toman decisiones al principio de cada ranura o *slot*. Además, como se ha mencionado anteriormente, el sistema se modela como un sistema de colas, por lo que los algoritmos deben decidir el tráfico que transita entre las colas. Antes de describir los algoritmos, en la Ecuación (2.1) se indica la actualización de las colas que se efectúa en cada *slot* y que es igual para todos los algoritmos.

$$Q_k(t+1) = \max\left[Q_k(t) - b_k(t), 0\right] + a_k(t) \tag{2.1}$$

En la Ecuación (2.1) la ocupación de la cola k en el *slot* t se indica como $Q_k(t)$. La llegada de paquetes en ese slot viene dada por $a_k(t)$ y la salida de paquetes se representa como $b_k(t)$. De este modo, los diferentes algoritmos tienen que decidir la cantidad de paquetes que abandona cada cola, y entra en las otras, en cada uno de los *slot*.

2.1.1. Round-Robin

El algoritmo Round Robin (RR) es uno de los algoritmos más antiguos, sencillos y equitativos en el reparto de la CPU entre los procesos. Esto implica que el algoritmo evita la monopolización del uso de la CPU y es muy válido en entornos de tiempo compartido.

Su funcionamiento se basa en la asignación de recursos de forma alternativa y ordenada entre las diferentes colas del sistema. Este algoritmo será el primero a implementar y realizará una repartición equitativa entre las colas de los diferentes escenarios a implementar.

A modo de ejemplo como podemos ver en la Figura 2.2, si nos encontramos ante un escenario con una única CPU y dos posibles colas, el algoritmo reparte dicha capacidad de procesado entre ambas colas de manera alternativa.

Slot	1	2	3	4	5	6
CPU	Cola 1	Cola 2	Cola 1	Cola 2	Cola 1	Cola 2

Figura 2.2: RR: Reparto Capacidad de Procesado

2.1.2. Weighted Fair Queuing (WFQ)

Por otro lado, el algoritmo Weighted Fair Queuing (WFQ) da un paso más e intenta buscar la posibilidad de realizar repartos no equitativos, es decir, introducir el termino de prioridad en las colas, cosa que el Round Robin no consideraba. Como anteriormente se explicó, el Round Robin asignaba recursos alternativamente entre sus colas sin ajustarse a ningún otro parámetro. En este caso, el WFQ debe tener en cuenta la prioridad de las colas, es decir, si la cola 1 tiene prioridad 2 frente a la cola 2 que tiene prioridad 1, dos tercios de la capacidad de procesado serán para la cola 1 mientras que la cola 2 obtendrá el tercio restante.

Para ilustrar esta situación tenemos la Figura 2.3, donde como hemos comentado podemos ver cómo la CPU destina dos de cada tres recursos a la cola 1 mientras que la cola 2 obtiene su tercio de la capacidad total.

Slot	1	2	3	4	5	6
CPU	Cola 1	Cola 1	Cola 2	Cola 1	Cola 1	Cola 2

Figura 2.3: WFQ: Reparto Capacidad de Procesado

2.1.3. Max-weight backpressure

Y, por último, nos encontramos ante el algoritmo Max-weight backpressure. A diferencia del algoritmo Round Robin y el WFQ, se trata de un algoritmo dinámico o también conocido como algoritmo oportunista, es decir, este en función de las condiciones que presente el escenario en cada slot tomará una decisión u otra. En concreto este algoritmo tiene como objetivo principal la estabilidad de las colas. Aunque se pueden incluir métricas adicionales para implementar políticas dentro de los regímenes de estabilidad, en este trabajo se ha utilizado la versión más sencilla.

Una vez detectado una entrada de tráfico al sistema el algoritmo hará un estudio de todos los posibles caminos disponibles en el escenario, descartando aquellos de capacidad de tráfico menor frente a los que presenten una capacidad mayor. Con este descarte, finalmente el algoritmo procederá a realizar los movimientos oportunos por el sistema.

El algoritmo atiende a las siguientes Ecuaciones (2.2) y (2.3) En cada slot *t* se selecciona la acción α dentro de un conjunto de acciones permitidas que maximizar la Ecuación (2.2) que se indica a continuación:

$$\sum_{i=1}^{N}\sum_{j=1}^{N} b_{i,j}(\alpha(t))W_{i,j}(t) \tag{2.2}$$

donde i y j representan las colas origen y destino de un enlace; $b_{i,j}$ es la capacidad del enlace; y $W_{i,j}$ se conoce como backlog diferencial y se define como se indica en la Ecuación (2.3).

$$W_{i,j}(t) = Q_i(t) - Q_j(t) \tag{2.3}$$

En este caso, Q_i representa el tamaño del buffer en la cola origen y Q_j el tamaño del buffer en la cola destino.

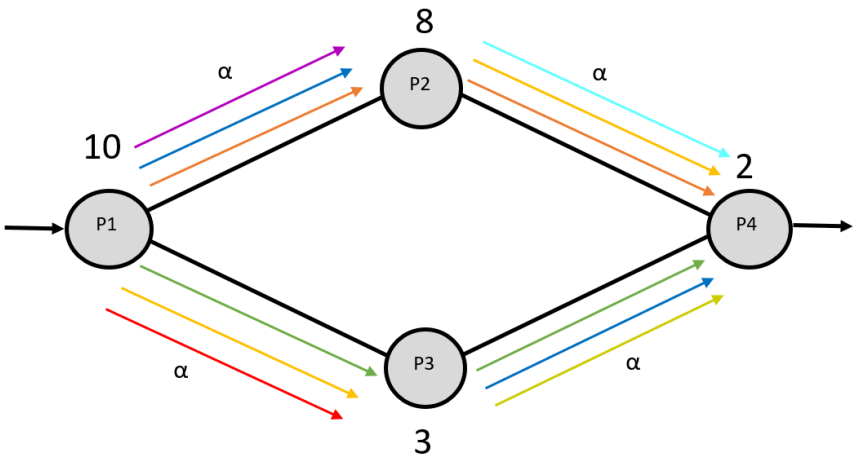


Figura 2.4: BackPressure: Posibles paths

Tabla 2.1: Leyenda: Posibles paths

Posibles paths	Color del path
$P1 \rightarrow P2 - P2 \rightarrow P4$	Naranja
$P1 \rightarrow P2 - P3 \rightarrow P4$	Azul Oscuro
$P1 \rightarrow P3 - P2 \rightarrow P4$	Amarillo
$P1 \rightarrow P3 - P3 \rightarrow P4$	Verde
$P1 \rightarrow P2$	Morado
$P1 \rightarrow P3$	Rojo
$P2 \rightarrow P4$	Azul celeste
$P3 \rightarrow P4$	Caqui
$None$	-

A modo de ejemplo, en la Figura 2.4 se muestra una red muy sencilla compuesta por 4 colas, donde el tráfico tiene que ir desde P1 a P4. Gracias a las flechas y sus diferentes colores, se observa que para este escenario sencillo encontramos un total de posibles decisiones, incluyendo aquel que no transmite ningún paquete. El detalle de cada decisión se muestra en la Tabla 2.1; cabe indicar que en el dibujo no se incluyen decisiones de no enviar.

A su vez, en la Figura 2.4 se representan el número de paquetes que contiene en ese momento cada elemento, así como la capacidad de los enlaces, α, que en este caso será genérica para todos ellos.

Una vez que se tienen todos los datos sobre el ejemplo, lo que hace el algoritmo del Backpressure es lo siguiente:

1. Realizados los cálculos y obtenido los resultados el algoritmo compara entre todos ellos y selecciona el resultado mayor, en este caso, el *path3* ($P1 \rightarrow P3 - P2 \rightarrow P4$) con valor 13α. El cálculo para cada alternativa se muestra en la Tabla 2.2. Siguiendo la definición

Tabla 2.2: Decisiones Backpressure (Figura 2.3)

Nº Path	Path	Operación	Resultado
1	$P1 \rightarrow P2 - P2 \rightarrow P4$	Path1=$(10-8)\alpha + (8-2)\alpha$	8α
2	$P1 \rightarrow P2 - P3 \rightarrow P4$	Path2=$(10-8)\alpha + (3-2)\alpha$	3α
3	$P1 \rightarrow P3 - P2 \rightarrow P4$	Path3=$(10-3)\alpha + (8-2)\alpha$	13α
4	$P1 \rightarrow P3 - P3 \rightarrow P4$	Path4=$(10-3)\alpha + (3-2)\alpha$	8α
5	$P1 \rightarrow P2$	Path5=(10-8)α	2α
6	$P1 \rightarrow P3$	Path6=(10-3)α	7α
7	$P2 \rightarrow P4$	Path7=$(8-2)\alpha$	6α
8	$P3 \rightarrow P4$	Path8=$(3-2)\alpha$	α
9	$None$	-	0

teórica se calcula el *differential backpressure* entre cada par de colas en el sistema que están conectadas. Después se evalúa la utilidad de cada posible decisión y se toma aquella con mayor utilidad.

2. Seguidamente se actualizará cada uno de los datos del escenario y para una nueva entrada de tráfico el algoritmo realizará una nueva comparación de las posibles decisiones, repitiendo en cada uno de los *slot*.

Capítulo 3

Entorno de simulación

El desarrollo del simulador se basa en la toma de decisiones en un tiempo ranurado en slots. Dependiendo del algoritmo deseado el sistema toma una u otra decisión basándose en sus requisitos y teniendo en cuenta siempre el número de paquetes que ha llegado al sistema en el slot anterior abordando así, la actualización de las colas.

La descripción del entorno de simulación se realizará usando un escenario sencillo. El mismo razonamiento que se expondrá para el escenario sencillo se ha seguido para los escenarios de mayor complejidad que se evaluarán en el siguiente capítulo. La razón por la cual se describirá la implementación a partir de un caso particular es que, debido a la complejidad de las topologías que se dan en los escenarios, no se ha podido realizar una implementación genérica y válida para cualquier escenario. Por el contrario, en la versión actual es necesario especializar la implementación de los algoritmos a la topología de red usada.

A lo largo de este capítulo se describirá la implementación del simulador, hablando de sus diferentes clases y métodos y haciendo una descripción de su funcionamiento. Aquellas clases que lo requieran irán acompañadas, también, de un pseudocódigo para facilitar y simplificar su entendimiento.
Inicialmente se explicarán las clases genéricas utilizadas, entre las que están un generador de variables aleatorias, paquetes, fichero de resultados y una clase que implemente una cola genérica. Cabe recordar que, como se ha mencionado anteriormente, la red vRAN se modela como un sistema de colas. Seguidamente se explicará la implementación de los algoritmos aplicados a sistemas de colas, haciendo uso de un escenario sencillo.

3.1. Generador de variables aleatorias

Dentro de los diferentes escenarios que se pueden dar o que se podrían implementar dentro del simulador no se puede dar por hecho que todo el tráfico de llegada será constante y controlado. Por ello y para explorar diferentes opciones a la hora de aportar resultados concluyentes, se crea

la clase *VarAleat*. Esta clase aglutina varias variables aleatorias existentes en la librería estándar de C++ y abstrae su uso.

- **Distribución de Bernoulli** De acuerdo con la función de probabilidad discreta, esta distribución produce valores booleanos aleatorios atendiendo a la Ecuación 3.1. [17]

$$P(b|p) = \left\{ \begin{pmatrix} (p) & \text{si} & b == true \\ (1-p) & \text{si} & b == false \end{pmatrix} \right\} \tag{3.1}$$

  Por ello, se crea el método *int Bernoulli(float p)* pasándole como parámetro *p*, la probabilidad de que el suceso sea cierto.

- **Distribución Binomial** A su vez, la distribución binomial produce números aleatorios no negativos acordes a la función de probabilidad discreta descrita en la Ecuación 3.2. [18]

$$P(i|t,p) = \binom{t}{i} \cdot p^i \cdot (1-p)^{t-i} \tag{3.2}$$

  Como resultado de la distribución se obtiene el número de éxitos de una secuencia de *n* experimentos independientes con una probabilidad fija *p* de ocurrencia de éxito. Por ello, nuestro simulador cuenta con el método *Binomial(int n, float p)*, donde fijamos, tanto el número de experimentos como la probabilidad de éxito de los mismos.

- **Distribución de Poisson** Por último, encontramos la distribución de Poisson, aquella que produce valores enteros aleatorios no negativos y que atiende a la función de probabilidad discreta descrita por la Ecuación 3.3. [19]

$$P(i|a) = \frac{e^{-a} \cdot a^i}{i!} \tag{3.3}$$

  El valor que nos devuelve es la probabilidad de que ocurran exactamente *i* sucesos de un evento aleatorio si el número medio esperado de esos sucesos es *a*. Para ello, se crea el método *Poisson(float a)*, donde como ya se ha comentado, se pasa el número medio de sucesos esperado.

Para la implementación de estos tres métodos es necesario el uso de las funciones de generación de números aleatorios *std::random_device* y *std::mt19937*. La primera de ellas genera números aleatorios enteros distribuidos uniformemente que producen números aleatorios no deterministas. Por su lado, la segunda de ellas se trata de un generador pseudoaleatorio Mersenne Twister de números de 32 bits.

3.2. Cola

La clase *Cola* es la encargada de representar cada cola en el sistema. Para ello se utilizara el contenedor *std:: queue*, un adaptador de contenedor que nos permitirá realizar las funciones necesarias, teniendo en cuenta que se trabajará con una estructura de datos FIFO (el primero en entrar, es el primero en salir).

La clase está compuesta por dos variable miembro:

- **std::queue<Paquete>m_buffer:** Contenedor de paquetes.
- **std::queue<Paquete>m_inputBuffer:** Contenedor de paquetes de entrada. Este almacenamiento adicional resulta necesario para implementar la actualización de las colas descrita en el Capítulo 2 para el algoritmo de Backpressure.

Y a su vez, por cuatro métodos públicos:

- **Void PushPacket(Paquete p):** Se trata de una función sencilla la cual se encarga de introducir en el contenedor m_inputBuffer el paquete pasado una vez ha sido llamada la función. En concreto este método inserta el paquete que se pasa como parámetro al final del *buffer*.
- **Paquete PopPacket(void):** Al contrario del método anterior este, se encarga de eliminar el primer paquete de la cola. La función, antes de eliminar cualquier paquete realiza una copia del mismo para poder retornar el valor una vez ha sido llamada la función. Con esto conseguimos relacionar e identificar los paquetes para futuros movimientos entre las colas.
- **Float BufferStatus():** Este método como su propio nombre indica, devuelve el número de paquetes que contiene la cola una vez es llamado.
- **UpdateQueue(void):** Y, por último, esta función se encarga de actualizar las colas del sistema. En ella se implementa un bucle que se realiza siempre y cuando el objeto m_inputBuffer sea mayor que cero. El contenido de este bucle introduce una copia del m_inputBuffer al m_buffer y vacía el primero de los objetos.

3.3. Paquete

Paquete se trata de una nueva clase de nuestro simulador. En ella se recogen todos los parámetros relacionados con los paquetes, como puede ser su slot de llegada y salida, su retardo o el identificador para conocer de qué paquete se trata. Para ello, lo primero a realizar es la asignación de un identificador único durante el tiempo de simulación en la construcción del paquete. Esto nos ayudará a tener un mayor control sobre los diferentes paquetes del sistema y su comportamiento en el mismo.
La clase define tres variables miembro:

- **int m_id:** Identificador del paquete.
- **int m_in:** Identificador del slot de llegada.
- **int m_out:** Identificador del slot de salida.

Y cuatro métodos muy sencillos:

- **void Llegada(int slot):** Almacena el slot en el objeto m_in.
- **void Salida(int slot):** Almacena el slot en el objeto m_out.
- **int Retardo(void):** El método devuelve el retardo sufrido por el paquete siempre y cuando el tiempo de salida se haya establecido.
- **int GetId():** Devuelve el identificador del paquete.

3.4. Fichero

Dentro de la implementación uno de los objetivos finales es la recolección de datos concluyentes e interesantes para el proyecto. Por tanto, para dicha recogida se ha implementado la clase *Fichero*, que se encarga de almacenar datos a lo largo de la simulación.
Para hacer una futura impresión de los resultados más ordenada y aclaratoria, en el constructor, definimos un prefijo, que nos indicará sobre que algoritmo estamos trabajando y una ruta donde serán guardados.
La clase, contiene tres variables miembro:

- **std::string m_prefix:** Prefijo del nombre de los ficheros de resultados.
- **std::string m_path:** Ruta donde se almacenarán los resultados.

- **std::map<std::string, std::map<int, float>> m_colas:** *m_colas* es un mapa de mapas que almacena la evolución de las colas. Los mapas son otro tipo de contenedor en el cual se pueden crear arrays asociativos. Este tipo de arrays se caracterizan porque sus elementos se asocian a una determinada clave[20]. En este caso, el objeto obtendrá como clave un *String* y su valor irá ligado a otro mapa con clave un número entero y su valor *float* correspondiente. Con ello se almacena, para cada cola determinada por su nombre en el *String*, su ocupación en cada uno de los *slots* que dura la simulación.

El paso de datos y la obtención de resultados se realiza a partir de los siguientes métodos:

- **void LogQueue(int i, float b, std::string q):** Este método obtiene como información, un entero con el número del slot en el que se encuentra el sistema, el número de paquetes que contiene el elemento del sistema y un string con el elemento asociativo deseado. Esta información se le proporciona al método con el fin de que este actualice la información de las colas. En el Algoritmo 1 se indica el pseudo-código utilizado en este método.

Algorithm 1 Código: LogQueue

```
if (m_colas.find(q) == m_colas.end()) then
    m_colas[q] = {};
end if
m_colas[q][i] = b;
```

- **void Escribir():** Una vez, la función anterior ha actualizado la información de todo el sistema toca representar esos datos en un fichero para más tarde poder analizarlo, por ello, se crea el método *Escribir*. El método consiste en un bucle anidado que permite adentrarse en todos y cada uno de los elementos del objeto *m_colas*. Gracias al primer bucle se escoge la cola de la cual se quiere representar sus datos y con el segundo, se representan. El Algoritmo 2 describe el pseudo-código implementado en este método.

Algorithm 2 Pseudocódigo Escribir()

```
for Recorrer el mapa m_colas do
    Abrir fichero çolas_- Nombre_Cola
    for (Recorrer el mapa interior de m_colas) do
        Escribir en el fichero el slot y el estado del buffer de la cola
    end for
    Cerrar fichero
end for
```

- **void LogDecisions(float u1, float u2, float u3, float u4):** Se trata de un método sencillo, encargado de recoger la probabilidad de selección de cada cola. El método además almacena esta información en su fichero correspondiente.

- **void Delays(std::vector<Paquete>&retardos):** Este método recibe como parámetro un vector de la clase *Paquete* y extrae a información de los paquetes. En concreto este método obtiene tanto el identificador del paquete con la función *Get.Id()* como el tiempo que permanece el paquete en el sistema, función *Retardo()* de la clase *Paquete* y los almacena en un fichero. En el Algoritmo 3 se muestra el pseudo-código relativo a este método.

Algorithm 3 Pseudocódigo Delays()

```
Abrir fichero f = Cola_ + Retardos
for Recorrer el vector retardos do
    Escribir identificador paquete + el delay del mismo
end for
Cerrar Fichero
```

3.5. Controladores

Como se ha comentado en la introducción del capítulo debido a las características de los diferentes escenarios y a la forma de abordar el código se ha optado por la realización de diferentes clases *Controladores* en función de la topología.
No obstante, esto no significa que dichas clases no tengan métodos y objetos en común. Las clases *Controladores* son las encargadas de realizar todas las órdenes del sistema, incluyendo, el encaminamiento de los paquetes, la actualización del sistema, la representación en los ficheros, etc.
Para ello, la clase se inicializa definiendo el número de iteraciones del sistema, es decir, el número de veces que este recibe tráfico.
Aunque la lógica implementada por cada controlador depende de la topología, las variables miembro que contienen son las mismas:

- **std::map<std::string, ClaseCola>m_colas**: El objeto *m_colas* consiste en un contenedor de tipo mapa donde el valor *ClaseCola* se asocia a la clave de tipo *String*, es decir, las colas van asociadas a un nombre.

- **std::vector<Paquete>m_queue:** A su vez *m_queue* es un contenedor de tipo vector donde se van almacenando los diferentes paquetes del sistema una vez que ha finalizado su procesado.

- **int m_pktId:** esta variable almacena el identificador que ha de asignarse al siguiente paquete generado.

- **int m_currentSlot:** indica el *slot* de simulación actual.

- **int m_numIters:** número total de iteraciones.

- **std::map<std::string, int>m_capacities:** Se trata de un mapa que asigna el valor de la capacidad de los enlaces entre las distintas colas del sistema.
- **VarAleat m_trafficGen:** Objeto relacional a las clase *VarAleat*.
- **int m_avgPktsPerSlot:** esta variable indica la tasa media de generación de tráfico.

Además de tener los siguientes métodos en común:

- **void Init():** Se trata de un método simple a la vez que imprescindible, ya que su tarea principal es inicializar el sistema con los datos necesarios. Para ello, inserta en los mapas *m_colas* y *m_capacities* tanto los elementos de la red como las capacidades de los enlaces, respectivamente, utilizando la función modificadora *Insert()* de este tipo de contenedor. Además, también inicializa el objeto *m_avgPktsPerSlot*.

Algorithm 4 Método Init()

```
m_colas.insert({ “Elemento1”, ClaseCola()});
m_colas.insert({ “Elemento2”, ClaseCola()});
m_colas.insert({ “Elemento3”, ClaseCola()});
...
m_capacities.insert({ “Enlace12”, x});
m_capacities.insert({ “Enlace13”, y});
m_capacities.insert({ “Enlace24”, z});
...
m_avgPktsPerSlot = w;
```

Los siguientes métodos son los encargados de administrar la red, realizando los movimientos de paquetes entre las colas del sistema de acuerdo al algoritmo aplicado. En el Algoritmo 4 se ejemplifica el funcionamiento de este método.

- **void Inyecta(int npkts, ClaseCola &dest):** Este método es el encargado de introducir los paquetes en la red por cada slot de tiempo. Como parámetros de entrada obtiene el número de paquetes y la cola destino donde se desean introducir. A lo largo del desarrollo observaremos que en nuestros escenarios el número de paquetes siempre se inyecta en la CU, pero gracias a la generalización del método se podrían introducir paquetes en cualquier cola del sistema. La lógica implementada por este método se muestra en el Algoritmo 5.

Algorithm 5 Método Inyecta()

```
for int i = 0; i <npkts; i++ do
    Paquete p(++m_pktId);                    ▷ Generación del paquete p
    p.Llegada(m_currentSlot);                ▷ Define el slot de llegada de p
    dest.PushPacket(p);                      ▷ Introduce p en la cola destino
end for
```

- **void Traspasar(int npkts, ClaseCola &orig, ClaseCola &dest):** El método *Transpasar*, como su propio nombre indica, transfiere un número determinado de paquetes, el cuál ha de coincidir con la capacidad del enlace, desde la cola origen a la cola destino, parámetros que se le pasan a la función. Un factor a tener en cuenta es la posibilidad de que la cola origen no tenga el suficiente número de paquetes que requiere en enlace, por tanto, el método hará una comparación entre el buffer de la cola origen y la capacidad del enlace, obteniendo el menor número entre ambos parámetros y transfiriendo dicha cantidad de paquetes. En el Algoritmo 6 se muestra la implementación del método.

Algorithm 6 Método Traspasar()

```
int pktsOrig = orig.BufferStatus();
int r = std::min(pktsOrig, npkts);       ▷ Selección del nº pkts a traspasar
for for (int i = 0; i <r; i++) do
    Paquete pkt = orig.PopPacket();      ▷ Elimina el pkt del origen
    dest.PushPacket(pkt);                ▷ Añade el pkt al destino
end for
```

- **void Drenar(int npkts, ClaseCola &orig):** Y, por último, el método *Drenar* consiste en ir vaciando y sacando un número determinado de paquetes que se le introduce como parámetro de entrada y la cola que se desea liberar. El método se desarrolla de forma genérica para que cualquier cola pueda ser drenada, pero en nuestras simulaciones siempre será la RU la protagonista del método. Como se puede ver en el Algoritmo 7, todos los paquetes drenados, y que por lo tanto han abandonado el sistema, se guardan en el contenedor *m_queue* para procesar su información al completar la simulación.

Algorithm 7 Método Drenar()

```
for (int i = 0; i <npkts; i++) do
    Paquete pkt = orig.PopPacket();      ▷ Elimina el pkt del origen
    pkt.Salida(m_currentSlot);           ▷ Define el slot de salida del pkt
    m_queue.push_back(pkt);              ▷ Empuja pkt al final de m_queue
end for
```

Y, finalmente, el último método común para las clases controlador:

- **void FinishQueueUpdates():**Este método es el encargado de recorrer el mapa m_colas e ir actualizando los valores nuevos que se han dado sobre las colas, esto lo hace con la función ya descrita en anteriores apartador *UpdateQueue()*, tal como se muestra en el Algoritmo 8.

Algorithm 8 Método FinishQueueUpdates()

```
for (auto &item : m_colas) do
    item.second.UpdateQueue();
end for
```

La diferencia principal que presentan los dos controladores implementados es el número de algoritmos a realizar y la complejidad de su implementación, basándose siempre en los métodos anteriormente descritos.
Al iniciar la propuesta del simulador lo que se buscaba en un principio era abordar un escenario sencillo que a través de los esquemas de scheduling tradicionales permitiera validar el correcto funcionamiento del simulador al completo y la capacidad de generar métricas de rendimiento de las diferentes entidades simuladas. Para ello, se hace uso del Escenario_0, un escenario sencillo compuesto por la Unidad Central(CU), la Unidad Radio (RU) y dos divisiones funcionales (S1,S2).
De la mano del Escenario_0, llega la implementación del *Controlador_0*, el cuál comprende todos los métodos ya mencionados anteriormente e implementa los tres algoritmos tradicionales descritos.

Una vez validado el correcto funcionamiento del primer escenario, se da un paso más allá y se busca uno más realista donde los resultados obtenidos sean más representativos. Para este propósito se hace uso del Escenario_1, compuesto por una Unidad Central(CU), dos switches, la Unidad Distribuida (DU) y la Unidad Radio (RU).
Este escenario se implementará en el *controlador_1*, el cual solo implementará el algoritmo del *BackPressure*, ya que al incluir la Unidad Distribuida (DU) utilizar cualquiera de los otros métodos no tendría mucho sentido.
A continuación, se detalla la lógica de cada uno de los algoritmos implementados en los controladores.

3.5.1. Round-Robin

La implementación del Round-Robin se trata de la más sencilla a realizar sobre nuestro escenario de validación (*Escenario_0*). El requisito a seguir por este scheduler consiste en la repartición de la CPU de forma alternativa y ordenada entre el número total de colas.
El algoritmo RR es un método de la clase *controlador_0*. Este consta de un bucle con un número de iteraciones determinadas, las cuales representan el número de slots de entrada al sistema. Utilizando la variable booleana *upPath* se controla la paridad de los slots, parámetro fundamental para la selección alternativa de las colas. Una vez el algoritmo se haya adentrado en el bucle, examinará el valor de la variable descrita anteriormente y según este tomará un camino u otro. Tras la primera decisión, el valor de la variable *upPath* es cambiado por su otra posibilidad haciendo posible la elección del siguiente camino en el siguiente *slot*.
La representación de las elecciones del algoritmo queda reflejada en la Figura 3.1 y clarificada en la Tabla 3.1.

Tabla 3.1: Posibles paths_RR_Controlador0

Posibles paths	upPath	Color Path
$CU \rightarrow S1 - S1 \rightarrow RU$	true	Azul
$CU \rightarrow S2 - S2 \rightarrow RU$	false	Verde

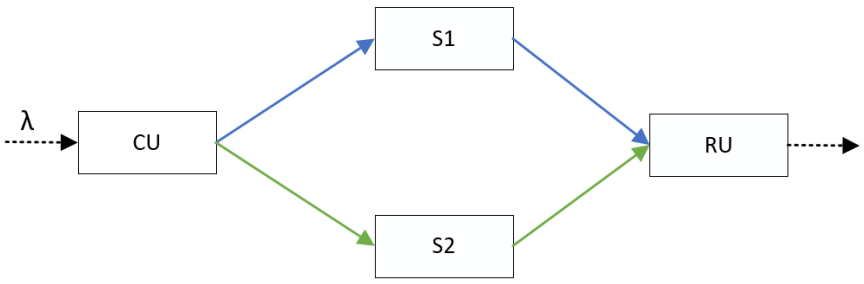


Figura 3.1: Controlador_0:Paths RR

En cada *slot*, y como se ha descrito en las secciones anteriores, la red es drenada, cargada con el tráfico definido y finalmente actualizada, cargando y escribiendo estos nuevos datos en los ficheros de resultados.
Por último, se calcula la utilización de los enlaces gracias a los contadores definidos, que incrementaran su valor en una unidad por cada vez que su enlace sea utilizado.
En el Algoritmo 9 se muestra el código relativo al método.

Algorithm 9 controlador_0: ROUND ROBIN

```
m_currentSlot = 0;
float u1 = 0, u2 = 0, u3 = 0, u4 = 0;                    ▷ Define la utilización de los enlaces.
Fichero logResults(“RR_", “resultsAux");                  ▷ Selección del fichero
bool upPath = true;                                       ▷ Variable de paridad

for (m_currentSlot = 0; m_currentSlot <m_numIters; m_currentSlot++) do
    if (upPath) then
        Traspasar(m_capacities[“CU_S1”], m_colas[“CU”], m_colas[“S1”]);
        Traspasar(m_capacities["S1_RU”], m_colas[“S1”], m_colas[“RU”]);
        ++u1;
        ++u3;
    else
        Traspasar(m_capacities[“CU_S2"], m_colas[“CU"], m_colas[“S2"]);
        Traspasar(m_capacities[“S2_RU"], m_colas[“S2"], m_colas[“RU"]);
        ++u2;
        ++u4;
    end if
    upPath = !upPath;                                     ▷ Cambio en la variable de paridad.
    Drenar(m_colas[“RU"].BufferStatus(), m_colas[“RU"]);  ▷ Vacio de RU
    Inyecta(tipo de tráfico, m_colas[“CU"]);              ▷ Entrada de tráfico al sistema
    FinishQueueUpdates();                                 ▷ Actualización de las colas
    /*Carga de los datos*/
    logResults.LogQueue(m_currentSlot, m_colas[“CU”].BufferStatus(), “CU");
    logResults.LogQueue(m_currentSlot, m_colas[“S1”].BufferStatus(), “S1");
    logResults.LogQueue(m_currentSlot, m_colas[“S2”].BufferStatus(), “S2");
    logResults.LogQueue(m_currentSlot, m_colas[“RU”].BufferStatus(), “RU");
end for
logResults.Escribir();

/*Calculo de la ocupación de los enlaces*/
u1 = (u1/m_numIters) · 100;
u2 = (u2/m_numIters) · 100;
u3 = (u3/m_numIters) · 100;
u4 = (u4/m_numIters) · 100;
logResults.LogDecisions(u1, u2, u3, u4);
logResults.Delays(m_queue);
```

3.5.2. Weighted Fair Queuing

Con el scheduler *Weighted Fair Queuing* se da un paso más allá que con el scheduler anterior y esta vez permite realizar un reparto de paquetes desigual por las colas del escenario atendiendo a la prioridad de estas.

El funcionamiento del algoritmo comienza definiendo la prioridad de las colas, esta prioridad se corresponde con la capacidad de cada enlace. Una vez definidas las prioridades, se genera un número aleatorio entre [0,1] por cada slot de entrada al sistema, que permitirá al algoritmo decidir por qué cola decantarse atendiendo siempre a su criterio de prioridad.

Al igual que en el RR, el WFQ cuenta con un bucle de iteraciones fijadas. En cuanto se adentra en él, comparará el número aleatorio con la prioridad de la primera cola. Si este número aleatorio es mayor o igual que cero y a la vez menor o igual que la prioridad del primer enlace, el algoritmo toma la decisión de ir por el camino superior de la red, es decir, pasando por *S1*. Si, por el contrario, el número aleatorio se sale del rango marcado, el camino seleccionado será el correspondiente a la ruta que pasa por *S2*.

- *p1*= 'prioridad cola 1'
- *p2*= 'prioridad cola 2'
- *RandNumber= 'Número aleatorio'*

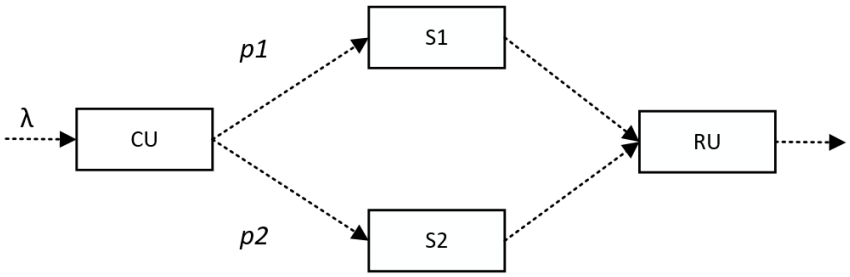


Figura 3.2: Controlador_0:Paths WFQ

Tabla 3.2: Posibles paths_WFQ_Controlador0

Decisión	Path
$0 \leq RandNumber \leq p1$	$CU \rightarrow S1 - S1 \rightarrow RU$
$p1 < RandNumber \leq 1$	$CU \rightarrow S2 - S2 \rightarrow RU$

Algorithm 10 controlador_0: WFQ

```
m_currentSlot = 0;
/*Definición de la prioridad de las colas*/
float prio1 = m_capacities[“CU_S1"];
float prio2 = m_capacities[“CU_S2"];
float p1 = prio1 / (prio1 + prio2);

float u1 = 0, u2 = 0, u3 = 0, u4 = 0;          ▷ Definición de la utilización de las colas
Fichero logResults(“WFQ_", “resultsAux");       ▷ Selección del fichero
srand(0);

for ((m_currentSlot = 0; m_currentSlot <m_numIters; m_currentSlot++)) do
    float RandNumber = (float)rand() / RAND_MAX;  ▷ Generación del número aleatorio

    /*Toma de decisiones*/
    if (RandNumber >= 0&&RandNumber <= p1) then
        Traspasar(m_capacities[“CU_S1"], m_colas[“CU"], m_colas[“S1"]);
        Traspasar(m_capacities[“S1_RU"], m_colas[“S1"], m_colas[“RU"]);
        u1++;
        u3++;
    else
        Traspasar(m_capacities[“CU_S2"], m_colas[“CU"], m_colas[“S2"]);
        Traspasar(m_capacities[“S2_RU"], m_colas[“S2"], m_colas[“RU"]);
        u2++;
        u4++;
    end if

    Drenar(m_colas[“RU"].BufferStatus(), m_colas[“RU"]);   ▷ Vacio de RU
    Inyecta(tipo de tráfico, m_colas[“CU"]);               ▷ Entrada de tráfico al sistema
    FinishQueueUpdates();                                  ▷ Actualización de las colas

    /*Carga de datos*/
    logResults.LogQueue(m_currentSlot, m_colas[“CU”].BufferStatus(), “CU");
    logResults.LogQueue(m_currentSlot, m_colas[“S1”].BufferStatus(), “S1");
    logResults.LogQueue(m_currentSlot, m_colas[“S2”].BufferStatus(), “S2");
    logResults.LogQueue(m_currentSlot, m_colas[“RU”].BufferStatus(), “RU");
end for
logResults.Escribir();

/*Calculo de la utilización de los enlaces*/
u1 = (u1/m_numIters) · 100;
u2 = (u2/m_numIters) · 100;
u3 = (u3/m_numIters) · 100;
u4 = (u4/m_numIters) · 100;
logResults.LogDecisions(u1, u2, u3, u4);   ▷ Escritura de la utilización de los enlaces
logResults.Delays(m_queue);                ▷ Escritura del retardo de los paquetes
```

3.5.3. Max-weight backpressure

Este último scheduler es el más completo dentro de toda la implementación. Se basa en encontrar, de entre todos los posibles caminos del escenario, aquel que permita transportar el máximo número de paquetes por slot.
Para ello, primeramente, tras adentrarse en el bucle, el algoritmo calcula la diferencia entre los *buffers* de las colas. Una vez realizada esta operación, calcula la cantidad real de tráfico que viajará de una a otra cola. Este cálculo se produce ya que, si la cola origen no tiene la cantidad de paquetes reclamadas por los enlaces, enviará todo lo que ella pueda y no lo que le demanden.
Con todos los cálculos realizados, se almacena en un mapa todos los posibles caminos con su correspondiente posibilidad de transmitir tráfico. Ese mapa será analizado por el algoritmo y seleccionará aquel camino, que se corresponda con el valor de tráfico a transmitir más alto.
Una vez, conocido el camino, el algoritmo hará una búsqueda de él entre las posibles combinaciones descritas y tras encontrarlo, realizará las correspondientes transiciones de tráfico.
Como en los casos anteriores, una vez realizada la selección, la red es drenada, inyectada y actualizada, dando paso a la representación de los resultados en los ficheros.
El Algoritmo 11, se trata de un algoritmo genérico ya que basándose en él y siguiendo su dinámica se implementará tanto en el *controlador_0* como en el *controlador_1*.

Algorithm 11 BackPressure

```
m_currentSlot = 0;
float u1 = 0, u2 = 0, ...                          ▷ Definición de la utilización de los enlaces
Fichero logResults(“BP_", “resultsAux");                                  ▷ Selección del Fichero

for ((m_currentSlot = 0; m_currentSlot <m_numIters; m_currentSlot++)) do
    // Calculando la acumulación de diferencias;*/
    int w_cu_s1 = m_colas[“CU”].BufferStatus() - m_colas[“S1”].BufferStatus();
    int w_cu_s2 = m_colas[“CU”].BufferStatus() - m_colas[“S2”].BufferStatus();
    ...
    /*Calculando la cantidad real de tráfico*/
    int b_cu_s1 = std::min(m_capacities[“CU_S1”], m_colas[“CU”].BufferStatus())
    int b_cu_s2 = std::min(m_capacities[“CU_S2”], m_colas[“CU”].BufferStatus());
    ...
    /*Almacenamiento de los posibles caminos + su capacidad de tx*/
    std::map<std::string, int>actions;
    actions.insert(“CU_S1-S1_RU", (w_cu_s1 * b_cu_s1) + (w_s1_ru * b_s1_ru));
    ...
    actions.insert(“NONE”, 0);
    /*Obtención del máximo path*/
    int maxValue = -1e9;
    std::string confMax = "NONE";
    for (auto &item : actions) do
        if (item.second >maxValue) then
            maxValue = item.second;
            confMax = item.first;
        end if
    end for
    /*Realización del encaminamiento de los pkts*/
    if confMax == “CU_S1-S1_RU” then
        Traspasar(m_capacities[ÇU_S1"], m_colas[ÇU"], m_colas["S1"]);
        Traspasar(m_capacities["S1_RU"], m_colas["S1"], m_colas[RU"]);
        u1++;
        u3++;
    else
        if confMax == “CU_S1-S2_RU” then
            Traspasar(m_capacities[ÇU_S1"], m_colas[ÇU"], m_colas["S1"]);
            Traspasar(m_capacities["S2_RU"], m_colas["S2"], m_colas[RU"]);
            u1++;
            u4++;
        end if
        ...
    end if
    Drenar(m_colas[“RU"].BufferStatus(), m_colas[“RU"]);          ▷ Vacio de RU
    Inyecta(tipo de tráfico, m_colas[“CU"]);          ▷ Entrada de tráfico al sistema
    FinishQueueUpdates();                              ▷ Actualización de las colas
    logResults.LogQueue(...)                           ▷ Carga los datos de las colas
end for
logResults.Escribir();
logResults.LogDecisions(u1, u2, u3, u4);     ▷ Escritura de la utilización de los enlaces
logResults.Delays(m_queue);                        ▷ Escritura del retardo de los paquetes
```

Capítulo 4

RESULTADOS

En este capítulo se representan los diferentes resultados obtenidos de las simulaciones realizadas para comprobar el buen funcionamiento de los algoritmos. Como se ha comentado se tienen dos escenarios diferentes. El primero de ellos, el *Escenario_0*, es un escenario sencillo utilizado para comprobar el correcto funcionamiento del entorno de simulación. Como parte de esta validación se comprobará el comportamiento del algoritmo *Backpressure* comparándolo con la otras soluciones implementadas.
Una vez, contrastados los primeros resultados de validación y observado que el funcionamiento del *Backpressure* es correcto, se analiza el segundo escenario del proyecto, *Escenario_1*, donde esta vez, se analizará el rendimiento de *Backpressure* utilizando una topología de red y configuración realistas.
Antes de presentar los resultados obtenidos en cada escenario se describen las métricas utilizadas para la evaluación.

4.1. Métricas

4.1.1. Estabilidad de las colas

La primera de las métricas que se analizará en el simulador es la estabilidad de las colas o *mean rate stability* que se define en la Ecuación 4.1. Esta métrica se consigue para cada una de las colas en los diferentes *slots* de la simulación.

$$\lim_{t\to\infty} \frac{E\{|Q(t)|\}}{t} = 0 \tag{4.1}$$

Para conseguir la estabilidad de las colas, lo que se debe observar es que el *mean rate* tienda a cero siempre que el número de *slots* tienda a infinito.

A modo de ejemplo, en la Figura 4.1 se muestra una simulación con dos colas durante 1000 slots. Como se puede ver la que la cola 1 presenta un comportamiento estable tendiendo a cero, al contrario que la cola 2, la cual presenta un comportamiento totalmente inestable.

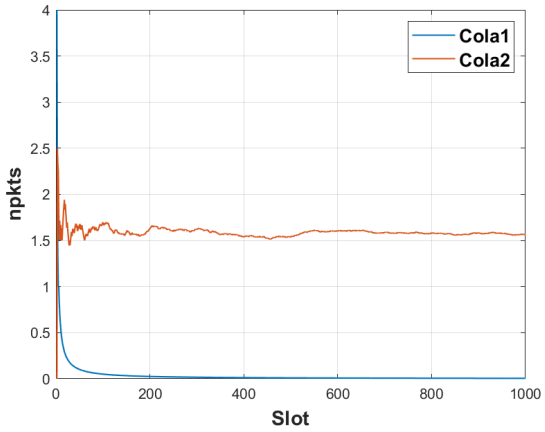


Figura 4.1: Mean rate stability

4.1.2. Retardos

Otra de las métricas que se analizará es el tiempo en *slots* que permanecen los paquetes desde que entran hasta que salen del sistema.
Su representación se realizará a través de la función de distribución acumulada (CDF), la cual es capaz de calcular la probabilidad acumulada del retardo medido en slots. En algunos casos también se mostrará la evolución de los retardos de los paquetes a medida que estos abandonan el sistema.

4.1.3. Selección de splits

La última métrica a analizar es la probabilidad con la que se selecciona cada cola y el reparto de los niveles de *split* sobre ellas. Esta métrica se representará mediante diagramas de barras para comparar las probabilidades de elegir los diferentes *splits*.

4.2. Escenario_0

En primer lugar, se encuentra el *Escenario_0*, es el encargado de realizar las comprobaciones de los tres algoritmos. Se trata de un escenario sencillo, Figura 4.2, compuesto por la Unidad Central (CU), la Unidad Radio (RU) y dos divisiones funcionales (S1,S2). Merece la pena recordar que se asume que los diferentes *splits* permiten enviar diferentes tasas de tráfico desde la CU. Al tener el tiempo ranurado con un valor fijo de *slot*, las diferentes tasas se traducen en diferentes capacidades de los enlaces.

Para llevar a cabo la validación de su funcionalidad, se presentan seis configuraciones, donde se variarán tanto la capacidad de los enlaces como el tráfico de entrada para poner a prueba a la red. En la Tabla 4.1 se indican las capacidades de los enlaces y tráfico utilizado en cada configuración.

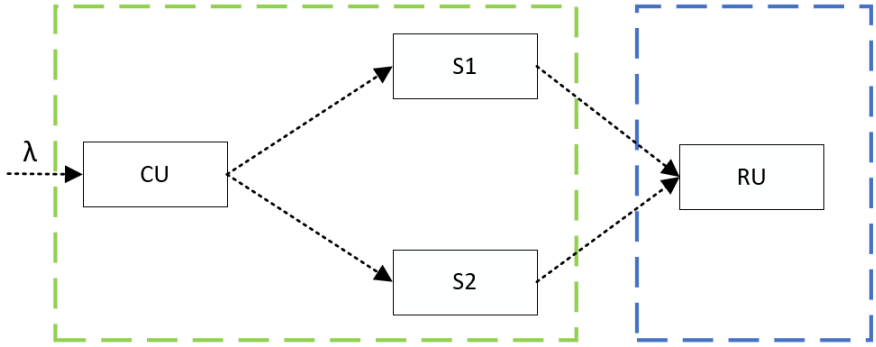


Figura 4.2: Escenario_0: Validación de la Implementación

Tabla 4.1: Escenario_0: Diferentes escenarios

-	CU-S1	CU-S2	S1-RU	S2-RU	Tráfico (λ)
Conf_0	10	10	10	10	Constante(5)
Conf_1	10	10	10	10	Poisson(5)
Conf_2	5	5	5	5	Poisson(5)
Conf_3	8	2	8	2	Poisson(5)
Conf_4	5	2	5	2	Poisson(5)
Conf_5	5	5	5	2	Poisson(5)

- **Conf_0:** La primera de las simulaciones se realiza frente a un escenario con tráfico constante y sobre-dimensionado, de tal modo que la capacidad de los enlaces es mucho mayor que el tráfico de entrada. Debido a la capacidad de los enlaces, se espera que todos los algoritmos presenten un comportamiento óptimo e igual.

  Como se puede observar en la Figura 4.3, se representan la estabilidad de las colas para los tres algoritmos implementados en el escenario y la gráfica correspondiente al retardo de los paquetes. Este patrón se llevará a cabo para la representación de las diferentes configuraciones. Con respecto a los resultados de la primera configuración, el comportamiento esperado se cumple y en todos los algoritmos se observa la estabilidad de las colas, así como un comportamiento óptimo con respecto al retardo de los paquetes.

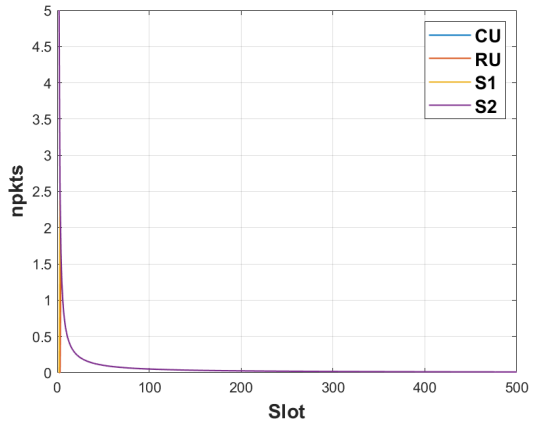
- **Conf_1:** La segunda simulación se realiza, también, sobre un escenario sobre-dimensionado pero esta vez con tráfico aleatorio siguiendo la distribución de Poisson. Gracias a esta situación se puede observar como en la Figura 4.4 el algoritmo del *Backpressure* puede adaptarse a los picos de tráfico, aunque debido al sobredimensionado de la capacidad también permite que los otros algoritmos se comporten de forma similar, mostrando la estabilidad de sus colas y un retardo óptimo.

- **Conf_2:** Un nuevo escenario de tráfico aleatorio de Poisson, pero este se diferencia del escenario anterior en que las capacidades ponen al escenario al borde de la saturación. Como se indica en la Tabla 4.1, tanto la capacidad de los enlaces como la tasa de tráfico es de 5 paquetes por *slot*. En la figura 4.5 se aprecian los resultados de este escenario y cómo el *BackPressure* es capaz de adaptarse a los picos de tráfico y consigue, a su vez, una mejor estabilidad mientras que los otros dos algoritmos empiezan a demostrar signos de inestabilidad en sus colas. Por otro lado, en la gráfica referida al retardo se puede ver que tanto el algoritmo de Round Robin como WFQ provocan retardos mucho mayores. Esto se debe a que el algoritmo de *Backpressure* se adapta de manera dinámica a las fluctuaciones de tráfico aleatorio de manera oportunista, mientras que los otros tienen un comportamiento estático y predefinido.

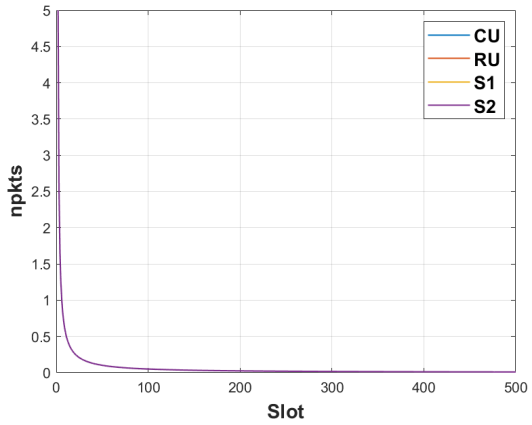
- **Conf_3:** Nuevo escenario, nueva metodología. Se estudia ahora la posibilidad de un escenario de capacidades no uniformes, pero sobredimensionadas. En concreto, en este escenario se tiene una capacidad mucho mayor por el camino superior (eligiendo el *split* 1), que por el inferior. Con este escenario se observa en la Figura 4.6 como el *Round-Robin* no está diseñado para este tipo de escenarios, mientras que el *Weighted Fair Queuing*, aprovechando las probabilidades asignadas, es capaz de tener un comportamiento estadístico mejor incluso que el *BackPressure*, como se puede observar en la Figura 4.6d, donde el retardo del WFQ es menor al del propio BP.

- **Conf_4:** La quinta simulación presenta un escenario con capacidades no uniformes y en saturación. En este caso, como se puede observar en la Figura 4.7 solo el *BackPressure* es capaz de conseguir la estabilidad. El *Weighted Fair Queuing*, incluso con las probabilidades específicamente definidas para este escenario se ve condicionado por el tráfico aleatorio. Además de en la estabilidad de las colas, se puede observar, cómo en la Figura 4.7d, la función de distribución acumulada para al *RR* y el *WFQ* se aleja mucho de tener un comportamiento óptimo.

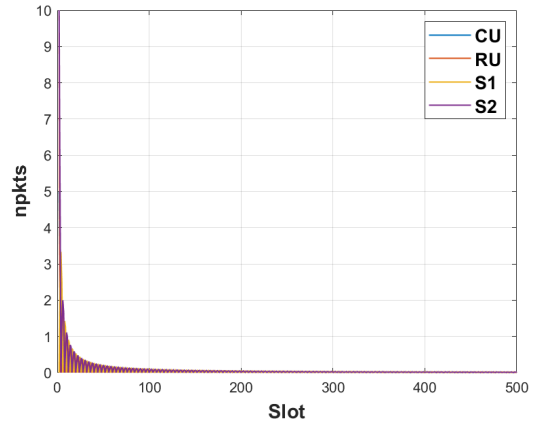
- **Conf_5:** Y, por último, ante un escenario no uniforme y con un camino actuando como cuello de botella, solamente el *Backpressure* es capaz de adaptarse a la heterogeneidad de la topología de la red, mostrando la estabilidad de las colas y un CDF óptimo (Figura 4.8) mientras que por su parte el RR y el WFQ presentan colas muy inestables, además de retardos mucho mayores.



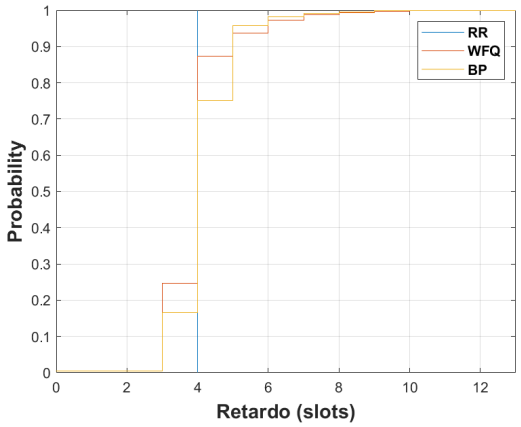
(a) Estabilidad RR



(b) Estabilidad WFQ

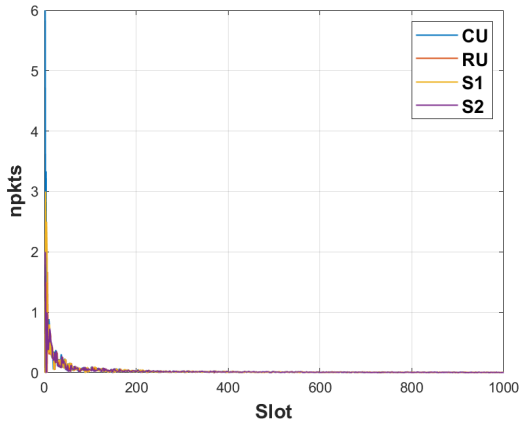


(c) Estabilidad BP

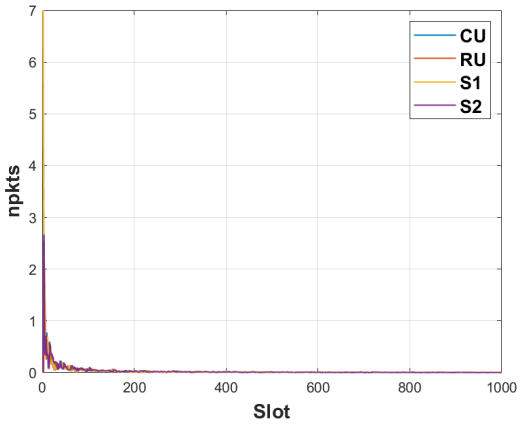


(d) CDF del retardo

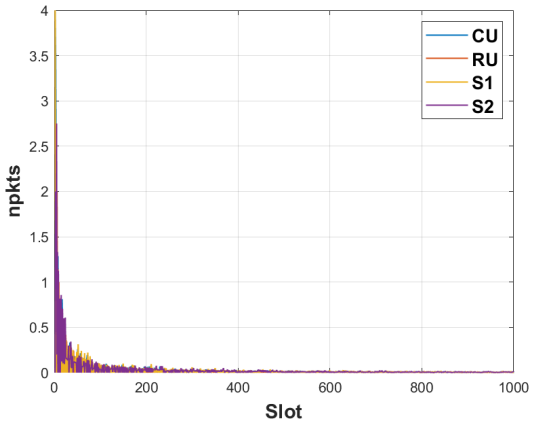
Figura 4.3: Resultados_Escenario0_conf0



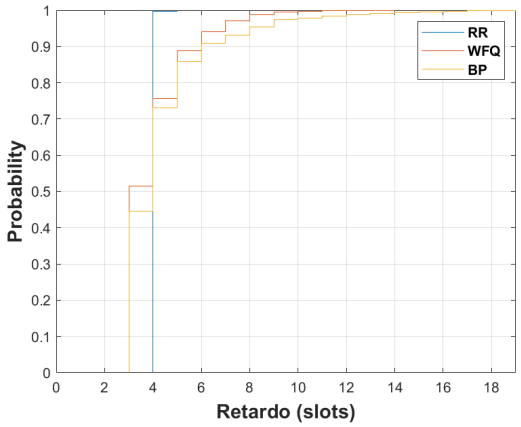
(a) Estabilidad RR



(b) Estabilidad WFQ

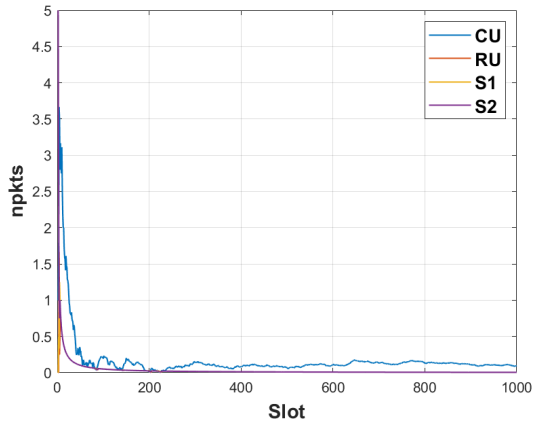


(c) Estabilidad BP

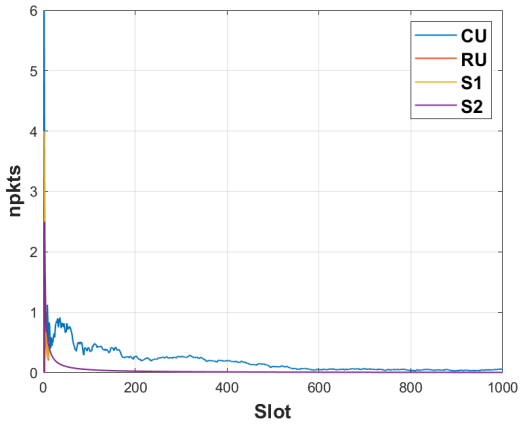


(d) CDF del retardo

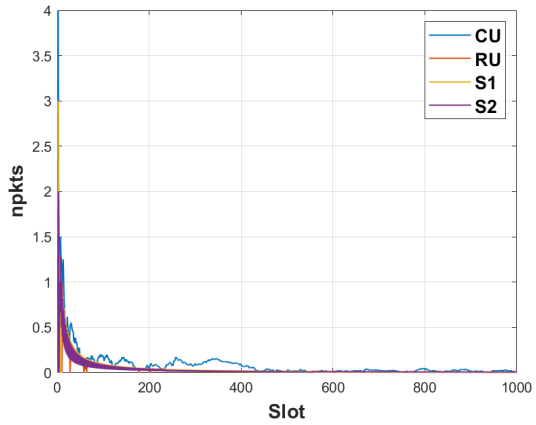
Figura 4.4: Resultados_Escenario0_conf1



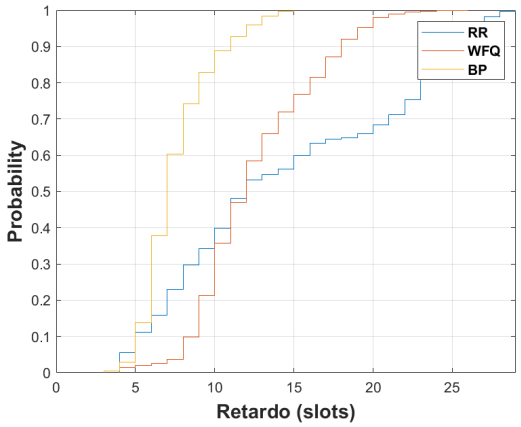
(a) Estabilidad RR



(b) Estabilidad WFQ

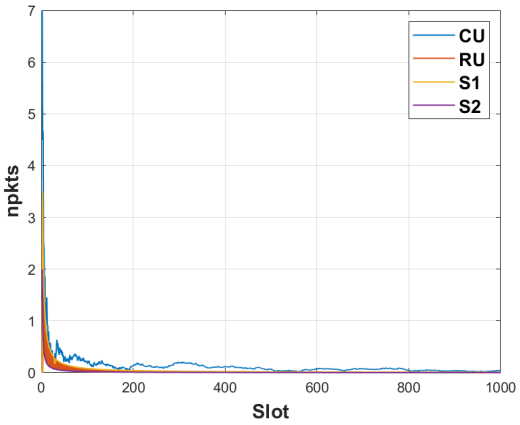


(c) Estabilidad BP

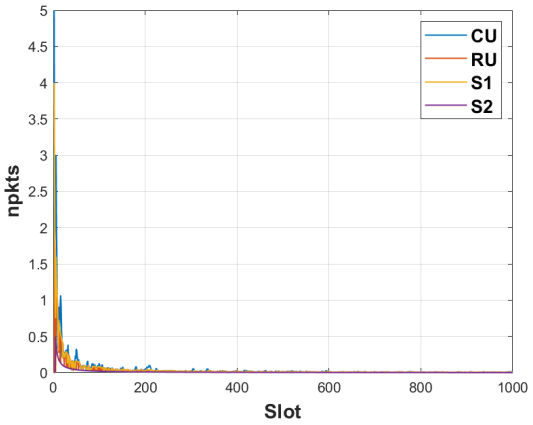


(d) CDF del retardo

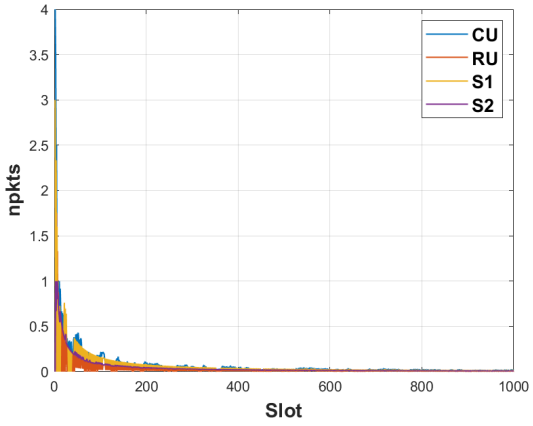
Figura 4.5: Resultados_Escenario0_conf2



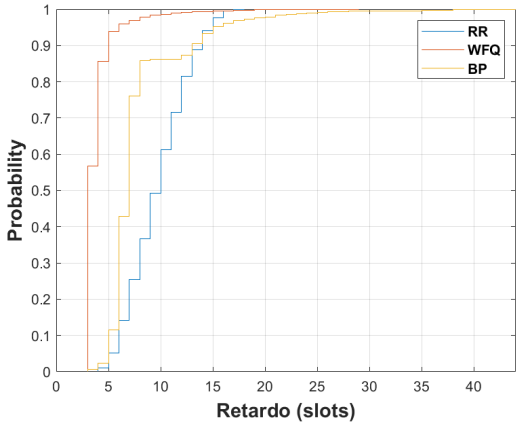
(a) Estabilidad RR



(b) Estabilidad WFQ

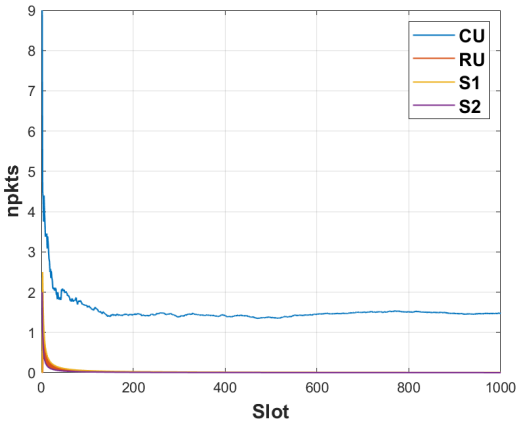


(c) Estabilidad BP

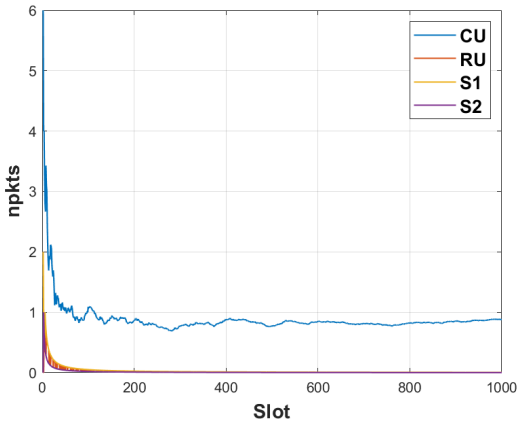


(d) CDF del retardo

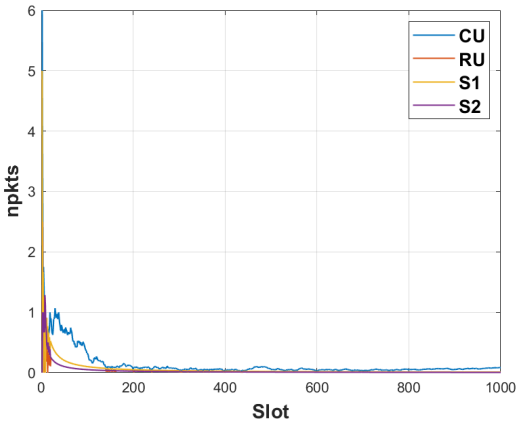
Figura 4.6: Resultados_Escenario0_conf3



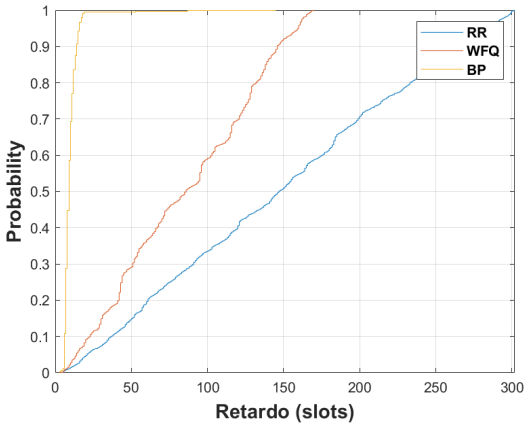
(a) Estabilidad RR



(b) Estabilidad WFQ

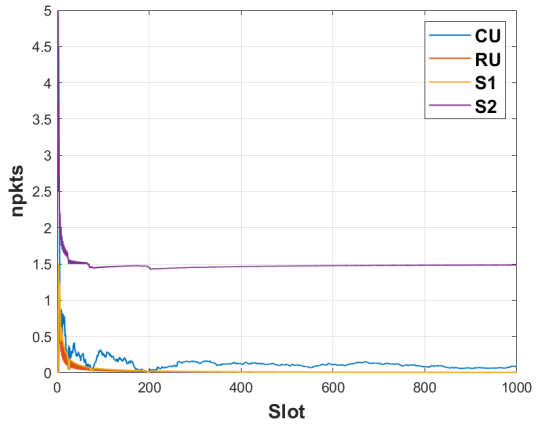


(c) Estabilidad BP

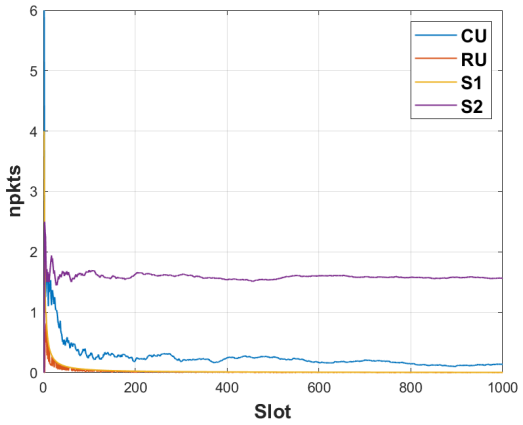


(d) CDF del retardo

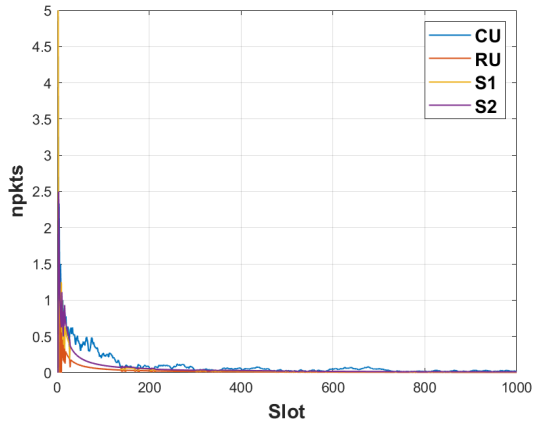
Figura 4.7: Resultados_Escenario0_conf4



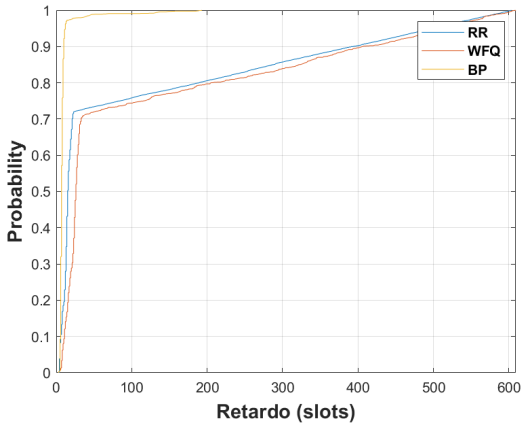
(a) Estabilidad RR



(b) Estabilidad WFQ



(c) Estabilidad BP



(d) CDF del retardo

Figura 4.8: Resultados_Escenario0_conf5

4.3. Escenario_1

El siguiente escenario es el *Escenario_1*. Como podemos observar en la Figura 4.9 se trata de un escenario más realista y complejo compuesto por una Unidad Central (CU), una Unidad Distribuida (DU), una Unidad Radio (RU) y dos switches. A su vez, cada switch se subdivide en dos colas con la finalidad de separar los caminos asociados a cada *split*, por lo que cada switch tiene tantas colas como splits hay en el sistema. Esta división de los *switches* se realiza para asegurar que un paquete asignado a un *split* en la CU usa el mismo *split* en la DU.

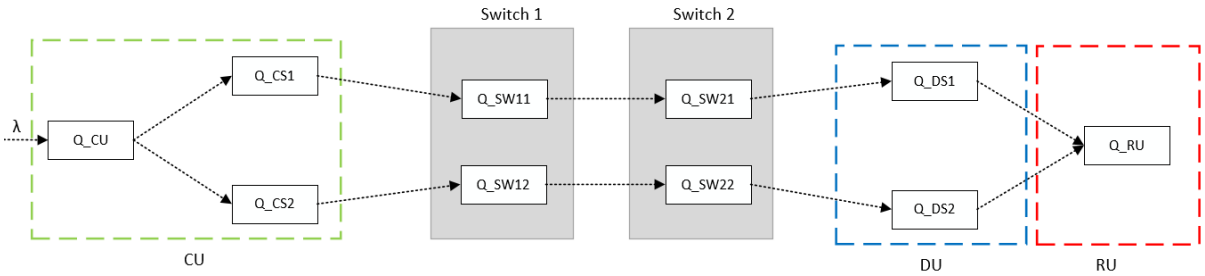


Figura 4.9: Escenario_1: Escenario realista

Para este escenario se utilizará el algoritmo *Backpressure* con el fin de realizar la selección de *split* más adecuada para cada caso. Debido a que nos encontramos en un nivel mayor de complejidad, sólo se estudiaran aquellos caminos completos (no se contemplan soluciones de no envío), teniendo en cuenta que el tráfico que entra por un split tiene que salir por el mismo. A modo de ejemplo, incluso para un escenario sencillo como el mostrado en la Figura 4.9, parecen 32 posibles soluciones de caminos completos, lo que aumentaría de manera exponencial si se considerar todas las combinaciones de no enviar en alguno de los segmentos.

Para este escenario, se presentan dos configuraciones diferenciadas por el nivel de split a utilizar y el tráfico de entrada al sistema.

Datos comunes:

- ***slot*= 1 ms.**
- **Capacidad enlaces Unidad Central**: Infinitos, (10000 pkts/s)
- **Capacidad enlaces entre switches**: 83.3 pkts/s (1Gbps asumiendo paquetes de 1500 Bytes)

A partir de los datos comunes se han realizado dos configuraciones. En ambos casos se tienen 2 splits, cuya capacidad varía en cada configuración. La primera representa un configuración en la que se pueda variar entre los split C-RAN y MAC/PHY, mientras que en la segunda se puede variar entre RLC/MAC y PDCP/RLC. Las capacidades de los enlaces en cada split se obtienen tomando como base la tasa media de tráfico de una estación base.

Computational Complexity (GOPS) (macro 20 MHz)			Traffic rate (pkt/s) Packet size = 1500B					
	Ratio relative to C-RAN		6PRBs	15 PRBs	25 PRBs	50 PRBs	75 PRBs	100 PRBs
C-RAN	20,9	0,96	639,23	1580,86	2618,18	5240,19	7804,78	12021,69
Intra-PHY	16	1,25	835,00	2065,00	3420,00	6845,00	10195,00	15703,33
MAC/PHY	10,7	1,87	1248,60	3087,85	5114,02	10235,51	15244,86	23481,62
Intra-MAC	8,7	2,30	1535,63	3797,70	6289,66	12588,51	18749,43	28879,69
RLC/MAC	6,7	2,99	1994,03	4931,34	8167,16	16346,27	24346,27	37500,50
Intra-RLC	5,7	3,51	2343,86	5796,49	9600,00	19214,04	28617,54	44079,53
PDCP/RLC	4,7	4,26	2842,55	7029,79	11642,55	23302,13	34706,38	53458,16
RRC/PDCP	2,7	7,41	4948,15	12237,04	20266,67	40562,96	60414,81	93056,79

(a) Datos CU

Computational Complexity (GOPS) (macro 20 MHz)			Traffic rate (pkt/s) Packet size = 1500B					
	Ratio relative to C-RAN		6PRBs	15 PRBs	25 PRBs	50 PRBs	75 PRBs	100 PRBs
C-RAN	0,1	50,00	33400,00	82600,00	136800,00	273800,00	407800,00	628133,33
Intra-PHY	5	1,00	668,00	1652,00	2736,00	5476,00	8156,00	12562,67
MAC/PHY	10,3	0,49	324,27	801,94	1328,16	2658,25	3959,22	6098,38
Intra-MAC	12,3	0,41	271,54	671,54	1112,20	2226,02	3315,45	5106,78
RLC/MAC	14,3	0,35	233,57	577,62	956,64	1914,69	2851,75	4392,54
Intra-RLC	15,3	0,33	218,30	539,87	894,12	1789,54	2665,36	4105,45
PDCP/RLC	16,3	0,31	204,91	506,75	839,26	1679,75	2501,84	3853,58
RRC/PDCP	18,3	0,27	182,51	451,37	747,54	1496,17	2228,42	3432,42

(b) Datos DU

Figura 4.10: Capacidades de los enlaces en las dos configuraciones del Escenario 1. En rojo se resaltan los valores para la configuración 1 y en amarillo para la configuración 2. La capacidad de cómputo de la CU y de la DU se han fijado en 20 y 5 GOPS

A continuación, esta tasa se escala como el ratio de la complejidad computacional de cada split relativa a la capacidad de cómputo de cada entidad CU o DU. En la Figura 4.10 se muestran los datos de complejidad computacional de cada split, así como las tasas (en paquetes por segundo) asociadas a ellos, para diferentes tamaños de estaciones base. En concreto, para este análisis se ha seleccionado una estación base de 100 Physical Resource Block (PRBs) , lo que equivale a un ancho de banda de 20 MHz en tecnología LTE.

Los datos obtenidos en la Tablas 4.2 y 4.3 son la consecuencia de realizar el producto del máximo número de paquetes por *slot* en cada una de las configuraciones, que es el valor que finalmente se utiliza en la simulación.

Al primer subescenario se le conocerá como *Escenario_11*. Con este escenario se realizarán tres simulaciones atendiendo a un tráfico de Poisson 5, 10 y 15 paquetes por slot. Como se ha mencionado, el escenario admite dos posibles divisiones funcionales, en este caso , se estudiarán la C-RAN (spli1) y la MAC/PHY (split2). De acuerdo con el 3GPP[15], el split MAC/PHY, se corresponde con la opción 6 de las posibles divisiones funcionales entre la CU y la DU(ver Figura 2.1). En este split, la capa física y la RF se encuentran en la unidad distribuida mientras que las capas superiores residen en la unidad central.

En la Figura 4.11 se muestra la utilización de cada una de las colas, lo que indica la probabilidad de seleccionar uno u otro split. Como se puede observar que para un tráfico de 5 paquetes por slot la utilización de los enlaces es prácticamente equitativa ya que todos son capaces de soportar el tráfico a la entrada por el sobre-dimensionado del escenario.

Tabla 4.2: Tasas de los enlaces entre colas en el Escenario_11 medido en paquetes por slot.

Escenario_11	CS1-SW11	CS2-SW12	DS1-RU	DS2-RU
C-RAN	12.02	-	628.13	-
MAC/PHY	-	23.5	-	6.09

Al aumentar el tráfico a 10 paquetes por slot se empieza a observar cómo en los enlaces de la CU se reparten la carga de trabajo equitativamente, pero una vez llegados a DU la selección del segundo split es mucho más probable. Esto se debe a que el primer split tiene mucha capacidad, por lo que no es necesario seleccionarlo de manera frecuente, mientras el segundo se debe seleccionar más veces para asegurar la estabilidad.

Al seguir aumentando el tráfico empieza a hacerse más notable la diferencia en la carga de trabajo entre la Unidad Central y la Unidad Distribuida y se aprecia cómo el algoritmo se decanta por la utilización del *split 2*, en este caso, el *split* MAC-PHY.

De este gráfico se comprende, que cuanto menor sea el tráfico, más funcionalidades se realizarán en la Unidad Central y menos en la Unidad distribuida y viceversa. A su vez, en la Figura 4.12a se puede observar cómo a medida que el tráfico aumenta, la estabilidad de la cola DS2 comienza a perderse hasta ser totalmente inestable.

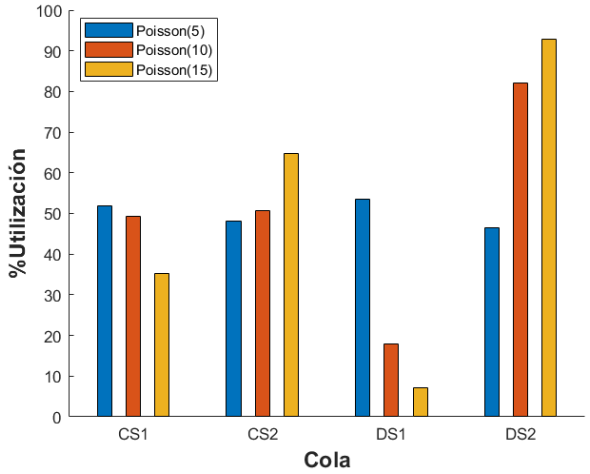
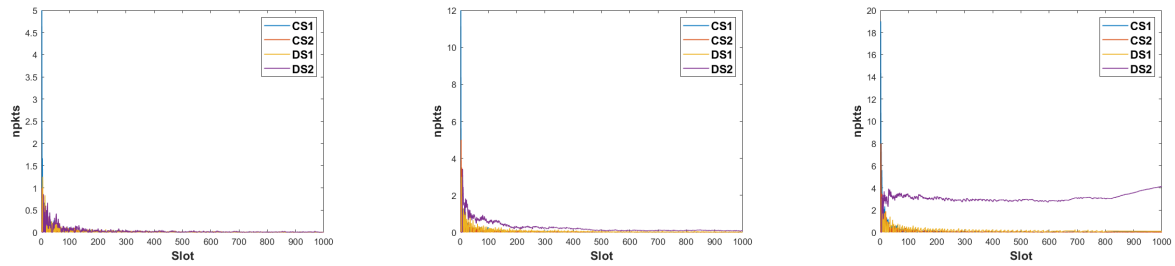


Figura 4.11: Escenario_11: Utilización de las colas



(a) Estabilidad BP_Poisson(5) (b) Estabilidad BP_Poisson(10) (c) Estabilidad BP_Poisson(15)

Figura 4.12: Resultados Escenario 1_1

El segundo subescenario se corresponde con el *Escenario_12*. Al igual que el anterior subescenario, realizara tres simulaciones con tráfico de Poisson con tasa 1, 3 y 5 paquetes por slot haciendo uso de las divisiones funcionales RLC/MAC y PDCP/RLC.
Según el 3GPP [15], el *split* RLC/MAC agrupa el RRC, DPCP y RLC en la unidad central mientras que en la unidad distribuida se encuentran la capa física, MAC y RF. En referencia al *PDCP/RLC*, agrupa RRC y PDCP en la unidad central mientras RLC, MAC, capa física y RF se quedan la unidad distribuida.

Tabla 4.3: Tasas de los enlaces entre colas en el Escenario_12 medido en paquetes por slot.

Escenario_12	CS1-SW11	CS2-SW12	DS1-RU	DS2-RU
RLC/MAC	37.5	-	4.39	-
PDCP/RLC	-	53.45	-	3.85

En la tabla 4.3 se observa que, en este caso, los enlaces de la unidad central presentan sobre-dimensionado, actuando como cuello de botella aquellos pertenecientes a la unidad distribuida.

Una vez obtenidos los resultados representados en la Figura 4.13 podemos observar que para un tráfico de 3 paquetes por slot el comportamiento de la red es equitativo ya que la capacidad de

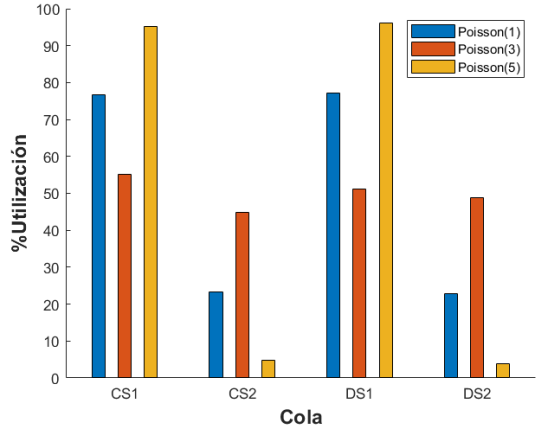
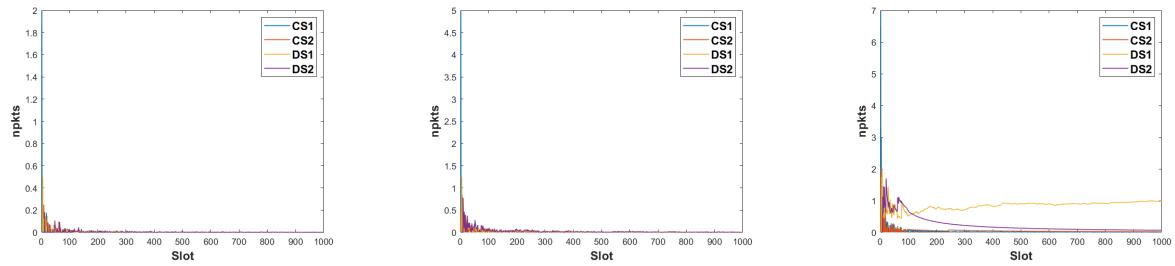


Figura 4.13: Escenario_12: Utilización de las colas



(a) Estabilidad BP_Poisson(1) (b) Estabilidad BP_Poisson(3) (c) Estabilidad BP_Poisson(5)

Figura 4.14: Resultados Escenario 1_2

los enlaces de la unidad distribuida se encuentran entorno a la entrada de tráfico. Por el contrario, en los casos extremos se observa cómo la red se decanta por la utilización del split RLC/MAC.

A su vez, analizando la Figura 4.14 se observa cómo para las dos primera situaciones la red presenta un comportamiento estable, siendo en la tercera situación (con tasa de tráfico de 5 paquetes por slot) donde las colas pertenecientes a la unidad distribuida presentan inestabilidad, la cual DS2 es capaz de corregir pero DS1 no. De estos resultados podemos observar como a mayor tráfico en la entrada, la red apuesta por realizar más funcionalidades en la Unidad Central.

A la vista de los resultados se puede concluir que la estabilidad del sistema no se encuentra siguiendo una política concreta de centralización o descentralización. Como se ha observado, en el primer escenario a medida que se saturaba el sistema la búsqueda estabilidad lleva a una mayor descentralización, mientras que en la segunda configuración ocurre al revés. Esto refuerza la idea de aplicar políticas dinámicas (como la ofrecida por *Backpressure*) que tengan en cuenta el estado del sistema en cada momento.

Capítulo 5

CONCLUSIONES

Finalmente, gracias a este trabajo se han introducido y profundizado más los conceptos C-RAN y la división funcional capaz de tomar decisiones basándose en el comportamiento de la red en el momento oportuno. Además de esto, se ha podido observar un desarrollo detallado de la implementación desde algoritmos más tradicionales y sencillos hasta aquellos capaces de asignar los recursos de procesado a las colas y de tomar decisiones de nivel de split.

También, se han podido comparar el comportamiento de este algoritmo más desarrollado con aquellos que no son capaces de aplicar esa flexibilidad a sus decisiones y gracias a los resultados obtenidos se han podido resaltar sus desigualdades. Además de esto, bajo diferentes escenarios, se ha podido observar el comportamiento del algoritmo y analizar los diferentes resultados que provocaban en él los diferentes cambios de escenarios.

Como se ha mencionado en capítulos anteriores, la evaluación en escenarios realistas evidencia que la política de split a aplicar ante cargas de tráfico elevadas no es única. La evaluación a reflejado la importancia de la capacidad de envío en cada uno de los split, que a su vez dependerá del tamaño de la estación base, así como de las capacidades de cómputo. Estos resultados refuerzan la idea de usar soluciones adaptativas y oportunistas para la selección de split.

A partir de este trabajo se plantean varias líneas futuras. En primer lugar, el trabajo da como uno de sus principales resultados el desarrollo de un entorno de simulación. Esta herramienta podrá ser utilizada en el futuro para evaluar diferentes escenarios en los que se tengan en cuenta más split y varias estaciones base que compartan recursos computacionales de la CU.

Por otro lado, se podría abordar la resolución del problema de optimización que se obtiene en el algoritmo de *Backpressure* mediante técnicas heurísticas.

Bibliografía

[1] *Telefonía 5G: qué es, ventajas y cobertura en España – comparativa 4G vs 5G*. https://www.adslzone.net/reportajes/telefonia/5g/.

[2] *IMT Vision – Framework and overall objectives of the future development of IMT for 2020 and beyond*. https://www.itu.int/dms_pubrec/itu-r/rec/m/R-REC-M.2083-0-201509-I!!PDF-E.pdf. Accessed: 2021-08-20.

[3] A. Checko, H. L. Christiansen, Y. Yan, L. Scolari, G. Kardaras, M. S. Berger y L. Dittmann. "Cloud RAN for Mobile Networks—A Technology Overview". En: *IEEE Communications Surveys Tutorials* 17.1 (2015), págs. 405-426. DOI: 10.1109/COMST.2014.2355255.

[4] *Cisco Annual Internet Report (2018–2023) White Paper*. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html. Accessed: 2021-08-20.

[5] G. O. Pérez, J. A. Hernández y D. Larrabeiti. "Fronthaul network modeling and dimensioning meeting ultra-low latency requirements for 5G". En: *IEEE/OSA Journal of Optical Communications and Networking* 10.6 (2018), págs. 573-581. DOI: 10.1364/JOCN.10.000573.

[6] C. C. Erazo-Agredo, M. Garza-Fabre, R. Agüero, L. Diez, J. Serrat y J. Rubio-Loyola. "Joint Route Selection and Split Level Management for 5G C-RAN". En: *IEEE Transactions on Network and Service Management* (2021), págs. 1-1. DOI: 10.1109/TNSM.2021.3091543.

[7] *Las redes cRAN (Cloud Radio Access Network)*. http://micm.es/noticias/las-redes-cran-cloud-radio-access-network/.

[8] *Evolución de la red de comunicación móvil, del 1G al 5G*. https://www.universidadviu.com/es/actualidad/nuestros-expertos/evolucion-de-la-red-de-comunicacion-movil-del-1g-al-5g.

[9] A. Pizzinat, P. Chanclou, F. Saliou y T. Diallo. “Things You Should Know About Fronthaul”. En: *Journal of Lightwave Technology* 33.5 (2015), págs. 1077-1083. DOI: 10.1109/JLT.2014.2382872.

[10] *El fronthaul y su importante papel en el avance de la tecnología 5G*. https://www.viavisolutions.com/es-es/fronthaul. Accessed: 2021-08-12.

[11] A. D. La Oliva, X. C. Perez, A. Azcorra, A. D. Giglio, F. Cavaliere, D. Tiegelbekkers, J. Lessmann, T. Haustein, A. Mourad y P. Iovanna. “Xhaul: toward an integrated fronthaul/backhaul architecture in 5G networks”. En: *IEEE Wireless Communications* 22.5 (2015), págs. 32-40. DOI: 10.1109/MWC.2015.7306535.

[12] C.-l. I, Y. Yuan, J. Huang, S. Ma, C. Cui y R. Duan. “Rethink fronthaul for soft RAN”. En: *IEEE Communications Magazine* 53.9 (2015), págs. 82-88. DOI: 10.1109/MCOM.2015.7263350.

[13] L. M. P. Larsen, A. Checko y H. L. Christiansen. “A Survey of the Functional Splits Proposed for 5G Mobile Crosshaul Networks”. En: *IEEE Communications Surveys Tutorials* 21.1 (2019), págs. 146-172. DOI: 10.1109/COMST.2018.2868805.

[14] B. Debaillie, C. Desset y F. Louagie. “A Flexible and Future-Proof Power Model for Cellular Base Stations”. En: *2015 IEEE 81st Vehicular Technology Conference (VTC Spring)*. 2015, págs. 1-7. DOI: 10.1109/VTCSpring.2015.7145603.

[15] 3. G. P. P. (3GPP). *Study on new radio accesstechnology: Radio access architecture and interfaces*. 2017. URL: https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3056.

[16] P. Arnold, N. Bayer, J. Belschner y G. Zimmermann. “5G radio access network architecture based on flexible functional control / user plane splits”. En: *2017 European Conference on Networks and Communications (EuCNC)*. 2017, págs. 1-5. DOI: 10.1109/EuCNC.2017.7980777.

[17] *cppreference.comstd::bernoulli_distribution*. https://en.cppreference.com/w/cpp/numeric/random/bernoulli_distribution.

[18] *cppreference.comstd::binomial_distribution*. https://en.cppreference.com/w/cpp/numeric/random/binomial_distribution.

[19] *cppreference.comstd::poisson_distribution*. https://en.cppreference.com/w/cpp/numeric/random/poisson_distribution.

[20] *cppreference.comstd::map*. https://en.cppreference.com/w/cpp/container/map.