ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Máster

ANÁLISIS DE SOLUCIONES COMBINADAS DE SCHEDULING Y SELECCIÓN DE FUNCTIONAL SPLIT EN ARQUITECTURAS C-RAN

(On the analysis of joint scheduling and functional split selection over C-RAN architectures)

Para acceder al Titulo de

Máster Universitario en Ingeniería de Telecomunicación

Autor: Borja Macho Pisano

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

MASTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

CALIFICACIÓN DEL TRABAJO FIN DE MASTER

Realizado por: Borja Macho Pisano
Directores del TFM: Luis Francisco Díez Fernández y Ramón Agüero Calvo
Título: "Análisis de soluciones combinadas de scheduling y selección de functional split en arquitecturas C-RAN"
Title: "On the analysis of joint scheduling and functional split selection over C-RAN architectures"

Presentado a examen el día: 19 de octubre de 2020

para acceder al Título de

MASTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

Composición del Tribunal:
Presidente (Apellidos, Nombre): Cobo García, Adolfo
Secretario (Apellidos, Nombre): Conde Portilla, Olga Mª
Vocal (Apellidos, Nombre): García Arranz, Marta

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

Fdo.: El Presidente	Fdo.: El Secretario
Fdo.: El Vocal	Fdo.: El Director del TFM (sólo si es distinto del Secretario)
Vº Bº del Subdirector	Trabajo Fin de Máster Nº (a asignar por Secretaría)

ÍNDICE

ÍNDICE DE FIGURAS

ACRÓNIMOS

3GPP 3rd Generation Partnership Project. 12

BBU Base Band Unit. 4, 9–13, 19, 27, 47

C-RAN Cloud Radio Access Network. 7–13, 64
CAPEX Capital Expenditure. 9
CoMP Coordinated Multi-Point. 9, 13
CPRI Common Public Radio Interface. 13
CPU Central Processing Unit. 4, 14–16, 19, 20, 23, 25–32, 34, 36, 39–41, 46, 47, 49–62, 64, 65
CSIT Channel State Information at the Transmitter. 8
CU Central Unit. 11

D-RAN Distributed Radio Access Network. 13
DU Data Unit. 11
DU Distributed Unit. 11

eMBB enhanced Mobile BroadBand. 8, 11

FFT Fast Fourier Transform. 9
FIFO First In, First Out. 23

GPP General-Purpose Processor. 9

HetSNets Heterogeneous and Small cell Networks. 8, 13

ICIC InterCell Interference Coordination. 9
IMT International Mobile Telecommunications. 8
ITU-R Radiocommunication Sector of the International Telecommunication Union. 8

LTE Long Term Evolution. 8, 13

MAC Medium Access Control. 12, 14
MCD Máximo Común Divisor. 36, 37
MDPP Min Drift-Plus-Penalty. 14, 16, 19, 34, 42–46, 51–54, 65
mMIMO massive MIMO. 8, 13
mMTC massive Machine-Type Communications. 8, 11
MQ Max Cola. 14, 15, 34, 35, 65
MU-MIMO Multi-User Multiple Input Multiple Output. 8

NFV Network Function Virtualization. 11, 12
NGFI Next Generation Fronthaul Interface. 13

OPEX Operating Expenditure. 9

PDCP Packet Data Convergence Protocol. 12, 14
PHY Physical Layer. 12, 14

QoE Quality of Experience. 8
QoS Quality of Service. 12, 13

RANaaS Radio Access Network as a Service. 13
RF Radiofrecuencia. 8, 12, 13
RLC Radio Link Control. 12, 14
RR Round Robin. 14–16, 32–34, 36, 65
RRC Radio Resource Control. 12
RRH Remote Radio Head. 4, 9–13, 18–20, 23–32, 34, 36, 39, 41, 46, 47, 49–62, 64, 65, 67–70
RRU Remote Radio Unit. 11
RU Radio Unit. 11

SDN Software Defined Networking. 11, 12
SDR Software Defined Radio. 64

TD-LTE Time-Division Long Term Evolution. 11

URLLC Ultra-Reliable and Low Latency Communications. 8, 10, 11

WFQ Weighted Fair Queuing. 14, 15, 23, 36–38, 65

ABSTRACT

The incessant growth of the traffic that current mobile networks must carry brings a change of the radio access network architecture really necessary. Cloud-RAN is one of the proposals to achieve higher capacities, better latencies, energy improvements, and greater coordination capacity between base stations, due to the benefits of virtualizing and centralizing baseband processing in data centers. However, it has one main drawback, the need for very high-performance links in the fronthaul network. In order to reduce the requirements imposed by these links, recent works propose to divide the tasks between the controller and the base stations, so that not all the processing is carried out in the centralized entity, in what is known as a functional split. In this way, it is possible to maintain some advantages of C-RAN and reduce the cost of the fronthaul network. There is also the possibility of choosing the functional split level based on the particular network status, which adds greater flexibility to the system and improvements when adapting to different traffic patterns. In this project we implement a scheduler capable of exploiting these advantages.

RESUMEN

El incesante crecimiento del tráfico que deben cursar las redes móviles actuales hace cada vez más necesaria un cambio en la arquitectura de la red de acceso radio. La Cloud-RAN es una de las propuestas para lograr mayores capacidades, mejores latencias, mejoras energéticas y mayor capacidad de coordinación entre estaciones base, gracias a la premisa de virtualizar y centralizar el procesado en banda base en centros de datos. Sin embargo, presenta un principal inconveniente, la necesidad de enlaces de muy altas prestaciones en la red fronthaul. Para lograr reducir los requisitos impuestos sobre estos enlaces se propone dividir las tareas entre el controlador y las estaciones base, de forma que no todo el procesado se realice en la entidad centralizada, en lo que se conoce como división funcional. De este modo, se logra mantener parte de las ventajas de C-RAN, y reducir el coste de la red fronthaul. Existe también la posibilidad de elegir el nivel de división funcional en base al estado de la red, lo que añade una mayor flexibilidad al sistema, y mejoras en el momento de adaptarse a los diferentes patrones de tráfico. En este trabajo se implementará un scheduler capaz de explotar dichas ventajas.

1. INTRODUCCIÓN

El camino hacia las redes móviles de quinta generación, 5G, está dirigido por varias motivaciones. En [1], publicado en 2015, la ITU-R definió lo que serían los objetivos del desarrollo de *International Mobile Telecommunications*, IMT-2020, siguiendo las nuevas demandas previstas para el futuro: capacidad para soportar mayor tráfico, dar servicio a una cantidad de dispositivos más variada y mucho mayor, mejorar la calidad de experiencia del usuario (QoE) y lograr una reducción en los costes que conllevaría una mayor rentabilidad para los operadores.

Siguiendo lo descrito en esa recomendación, IMT-2020 soporta tres escenarios de uso inalámbricos: enhanced Mobile BroadBand (eMBB), massive Machine-Type Communications (mMTC) y Ultra-Reliable and Low Latency Communications (URLLC).

El primero de los escenarios, eMBB, se centra en la utilización de la red por los usuarios humanos, siendo sus prioridades el acceso a contenido multimedia, los datos y los servicios. Se diferencian dos casos: la cobertura de gran área, donde interesa tener movilidad y velocidades de datos mucho mayores de los que había hasta el momento; y los *hotspots*, zonas de cobertura más pequeñas donde se espera una movilidad reducida y mayores tasas de datos.

El segundo escenario, mMTC, tiene por objetivo el uso de la red por parte de dispositivos de bajo coste y de los que se desea una larga duración de la batería. Se espera tener un gran número de "cosas" conectadas y que transmitan información no sensible al retraso a baja velocidad.

El tercero, URLLC, tiene requisitos muy estrictos en cuanto al retardo, el *throughput* y la disponibilidad de la red. Está pensado para su uso en el entorno industrial, operaciones médicas remotas, redes de distribución eléctrica inteligentes o conducción autónoma, entre otros. Sectores que requieren de una fiabilidad extremadamente alta al tratarse de elementos muy críticos y donde la aparición de retardos podría resultar altamente problemática.

La aparición de estos casos de uso, con las mejoras que conllevan, hace necesaria una renovación en la arquitectura de la red de acceso radio. Una de las propuestas capaz de proveer mayores capacidades y menores retardos, además de lograr mejor eficiencia, tanto energética como monetaria y de recursos para los operadores de red es la *Cloud Radio Access Network (C-RAN)*.

El concepto de C-RAN, propuesto por primera vez en [2], surge del incremento que se lleva viendo en el tráfico de las redes móviles durante los últimos años y la previsión de que este seguirá creciendo [3]. Para hacer frente a dicho tráfico se requiere un aumento en la capacidad de la red que, en el caso de C-RAN, se obtiene aprovechando las ventajas de las tecnologías de computación en la nube.

El estándar LTE se encuentra cerca del límite teórico de eficiencia espectral determinado por Shannon para un enlace punto a punto, esto hace necesarias nuevas técnicas para incrementar la capacidad [4]. Un método para lograrlo consiste en aumentar la densidad de la red, es decir, el número de antenas por unidad de área. Se tienen dos enfoques que persiguen este propósito: el uso de *arrays* de antenas masivos en las macro estaciones base o la división de la red en células más pequeñas. El primero de los planteamientos se conoce como MIMO masivo, *massive MIMO (mMIMO)* [5]. Permite focalizar con mucha precisión la potencia transmitida, permitiendo que esta sea muy reducida, lo que conlleva mejoras en la capacidad del sistema y la eficiencia energética, pero implica mayores costes de despliegue por el tamaño de las antenas y la necesidad de un mayor número de cadenas de RF. El segundo planteamiento consiste en reducir el área de las estaciones base mediante el despliegue de pequeñas células heterogéneas de muy alta densidad: *Heterogeneous and Small cell Networks (HetSNets)* [6], con mayor un coste e interferencia. Existe también la posibilidad de hacer uso de técnicas *Multiple Input Multiple Output* multiusuario, MU-MIMO [7], capaces de reducir la interferencia y mejorar el rendimiento al precio de requerir *Channel State Information at the Transmitter (CSIT)*.

Además de la necesidad de aumentar la capacidad de la red, a los operadores de las redes móviles se les presenta un problema económico cada vez mayor: deben hacer frente a unos gastos en constante crecimiento, mientras que las ganancias que obtienen de cada usuario se mantienen en valores similares o incluso se ven reducidas. Este problema hace que la eficiencia de la red en términos económicos tenga cada vez una mayor relevancia.

Los costes totales de los operadores se dividen en CAPEX y OPEX, del inglés *CAPital EXpenditure* y *OPerating EXpenditure*, respectivamente. Los primeros, los gastos de capital, son los relacionados con el despliegue de la red, desde la planificación hasta la compra e instalación de los equipos y sistemas para que estos funcionen; estos gastos incrementan rápidamente con el número de estaciones base, al tratarse de uno de los elementos más caros de la red inalámbrica. Los segundos, los gastos de operación, son los asociados al funcionamiento de la red, tales como alquiler de espacios, electricidad, mantenimiento y mejoras de los equipos; estos también aumentan con el número de estaciones base, ya que consumen gran cantidad de energía. Así, China Mobile estima que el 72 % del consumo energético en su red se debe a las estaciones base, tanto por el funcionamiento de los propios equipos como por su refrigeración [8].

La arquitectura C-RAN tiene un fundamento sencillo: llevar el procesado en banda base a un centro de procesado de datos, lugar donde se encuentran virtualizadas y centralizadas varias *Base Band Unit (BBU)* que comparten los recursos físicos. Estos recursos pueden ser procesadores de propósito general, *General-Purpose Processor (GPP)*, o componentes de aceleración por hardware para las tareas más exigentes computacionalmente, como codificación y decodificación, FFT, decodificadores MIMO, etc. Estos componentes hardware pueden ser compartidos por diferentes BBUs gracias a la virtualización de entrada/salida [9].

Mediante esta centralización, en lo que se conoce como *BBU Pools*, se logra utilizar los recursos de la red de una forma más eficiente. Además conlleva varias ventajas: facilita la coordinación entre estaciones base, permitiendo reducir el retardo de los *handovers* entre células dentro de un mismo *BBU Pool*; favorece el empleo de técnicas como *Coordinated Multi-Point (CoMP)* o *InterCell Interference Coordination (ICIC)*, que aumentan la capacidad de la red y la eficiencia espectral; o el balanceo de carga, que permite hacer frente a patrones de tráfico cambiante, como el conocido *tidal effect* entre las zonas residenciales y las laborales, encontrándose infrautilizadas las primeras y con mucha carga las segundas durante las horas de trabajo, o al contrario fuera de ellas, esto faculta la optimización de los recursos entre las estaciones base con mucha y poca carga. La arquitectura C-RAN se ve como una de las técnicas para hacer del 5G una tecnología más ventajosa para el medio ambiente [10] ya que, mediante la centralización, se produce una ganancia de multiplexado que permite tener un menor número de BBUs, reduciendo el consumo energético y los gastos en alquiler.

En la arquitectura C-RAN, la red se encuentra dividida entre las BBUs virtualizadas en la *BBU Pool* y los cabezales de radio remotos, *Remote Radio Head (RRH)*. Las señales radio son generadas y transmitidas por estos últimos, que a su vez se encargan de recibir las señales de radiofrecuencia de los usuarios y reenviar los datos al centro de procesado a través de los enlaces de una red de transmisión óptica. Esta parte de la red que conecta los RRHs con la *BBU Pool* se conoce como *fronthaul*.

De esta división de la red en dos partes, una centralizada formada por las BBUs y otra distribuida formada por las RRHs, surge el planteamiento de como efectuar el reparto de las tareas que deben ser realizadas en cada una, en lo que se conoce como *functional split* o división funcional. La literatura insta a emplear la opción de *split* más centralizada, es decir, con una mayor cantidad de tareas en la BBU y las menos posibles en el RRH a fin de maximizar las tasas de datos. Esta alternativa, en el momento de aplicarse, podría no ser realizable ya que el número de funciones que pueden ser centralizadas se ve limitado por la capacidad de los enlaces de la red *fronthaul* que conecta los RRHs con las BBUs, y por la capacidad del centro de procesado [11]. Por lo tanto el conjunto de funciones que pueden ser

centralizadas será una parte del total, apareciendo así el problema de determinar el reparto óptimo de las mismas entre RRH y BBU.

Un método para tomar una decisión del nivel de *split* en el que se va a trabajar, consiste en realizar estudios estadísticos del tráfico, número de usuarios, interferencia, etc. y determinar el *split* de forma estática en base a dichos estudios. Esto supone principalmente dos problemas: el primero es que al basarse en estimaciones, que pueden ser imprecisas, establecería configuraciones no del todo óptimas; el segundo es que cualquier desviación temporal respecto a los datos de las estimaciones lleva a un mal uso de los recursos.

Otro método consiste en realizar la selección de nivel de *split* de forma dinámica, en base al estado de la red, en lo que se conoce como *flexible* o *dynamic functional split*. Esto se puede hacer teniendo en cuenta gran cantidad de criterios: el *throughput*, el retardo, el consumo energético, la carga de la red, etc.

El objetivo de este trabajo es el desarrollo de un *scheduler* capaz de tomar decisiones de asignación de colas a recursos de procesado y selección de niveles de *split*, tratando de optimizar el retardo de los paquetes que atraviesan el sistema sin sacrificar la estabilidad de las colas para el posterior análisis de su rendimiento bajo diferentes escenarios. Este *scheduler* está pensado para el caso de uso URLLC ya que la finalidad del mismo es la de mantener el retardo lo más reducido posible, balanceando la carga entre la BBU y el RRH por medio de los niveles de *split*.

El resto del documento se encuentra estructurado de la siguiente manera: En la sección 2 se tiene información sobre la arquitectura C-RAN, trabajos relacionados y los *schedulers* analizados en este trabajo. En la sección 3 se muestran los detalles de la implementación desarrollada. En la sección 4 se muestran los resultados obtenidos para los diferentes escenarios analizados. Finalmente, en la sección 5 se tienen las conclusiones y líneas de trabajo futuro.

2. ESTADO DEL ARTE

En los últimos años los temas de investigación ligados al C-RAN y al *dynamic functional split* han ido ganando peso ya que son tecnologías que prometen mejorar las tasas de datos, las latencias, la eficiencia y la flexibilidad de la red. Esto las hace idóneas para la quinta generación de redes móviles, donde se dará servicio a gran variedad de dispositivos con diferentes requisitos según los tres casos de uso planteados para el 5G: *enhanced Mobile BroadBand*, *massive Machine-Type Communications* y *Ultra-Reliable and Low Latency Communications*.

La aparición del C-RAN sigue los pasos de la evolución vista en las generaciones previas. En un comienzo, con la primera y segunda generación de tecnologías de telefonía móvil, todas las funciones de procesado en banda base y radio estaban centralizadas en la estación base. Con la llegada de la tercera generación se produjo una división en unidad de radio, *Remote Radio Head (RRH)* o *Remote Radio Unit (RRU)*, y unidad de procesado banda base, *Base Band Unit (BBU)* o *Data Unit (DU)*. Ambas partes pueden estar separadas hasta 40 Km y conectadas mediante fibra óptica o enlaces microondas, además, un único BBU puede dar servicio a varias RRHs.

El siguiente avance se produce con la centralización de las BBUs en BBU Pools, constituidas por procesadores de propósito general y hardware dedicado para las funciones más pesadas computacionalmente, donde las funciones clásicas de la red de acceso radio pueden ser virtualizadas mediante técnicas de *Network Function Virtualization (NFV)* [12] y *Software Defined Networking (SDN)*, dando origen a la arquitectura C-RAN. Esta primera propuesta se basa en la centralización total y permite mayor flexibilidad de cara a las futuras mejoras de la red, reducción en los consumos energéticos, adaptabilidad a los patrones de tráfico no uniformes, mejoras en la capacidad de la red y las latencias, facilidad en la coordinación entre BBUs y, además, reduce la capacidad computacional necesaria en las estaciones base, con la consiguiente reducción en los costes de despliegue y operación [4].

La red *fronthaul*, que conecta la unidad banda base centralizada con la cabecera de radio remota, marca una limitación a la capacidad de centralización. Los enlaces conectando ambas partes deben ser de muy alta capacidad, muy alta fiabilidad y muy baja latencia. Estos requisitos tan elevados se deben a que, en caso de llevarse todas las funciones a la unidad central, la red *fronthaul* deberá soportar la transmisión de las muestras en bruto de fase y cuadratura (I/Q) de las RRHs y aumentan altamente el coste de los enlaces, siendo la implementación más común la fibra dedicada. Por ejemplo, una estación con 8 antenas TD-LTE con 20 MHz de ancho de banda necesita una tasa de transmisión de 10 Gbps.

A fin de reducir los requisitos de la red *fronthaul* aparece la división funcional, también conocida como *functional split*, donde una parte de las funciones banda base son centralizadas y el resto permanecen en los emplazamientos remotos [13]. Las tareas se reparten entre la unidad central o *Central Unit (CU)* y la unidad distribuida o *Distributed Unit (DU)* mientras que la RRH pasa a conocerse como unidad radio o *Radio Unit (RU)*. Esto permite mantener la mayoría de las ventajas de la centralización a la par que relajar los requisitos de la red *fronthaul*. Cada una de las distintas opciones de reparto de tareas se conoce como *split* funcional y tiene unas necesidades de capacidad, latencia y *jitter* en la red *fronthaul*. Surge también el concepto de división funcional en cascada, esto es, la posibilidad de tener unidades centrales o distribuidas entre el par CU-DU.

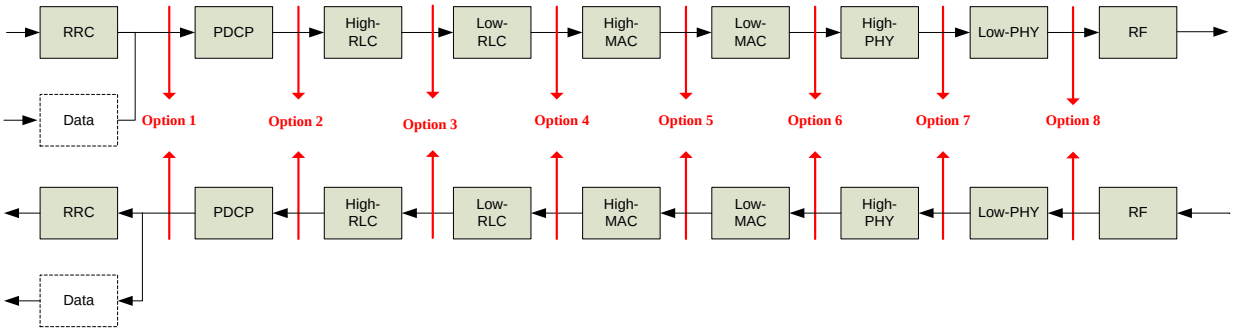


Figura 2.1: Opciones de división funcional entre BBU y RRH. Fuente [14]

En [14] el 3GPP prevé ocho posibles niveles de *split*, cinco de ellos entre las capas del *stack* de protocolos y tres *splits* internos dentro de las mismas. Estas ocho divisiones se encuentran ilustradas en la Figura 2.1 y son las siguientes:

- Opción 1 RRC-PDCP: RRC en la unidad central. PDCP, RLC, MAC, capa física y RF en la unidad distribuida.
- Opción 2 PDCP-RLC: RRC, PDCP en la unidad central. RLC, MAC, capa física y RF en la unidad distribuida.
- Opción 3 (intra RLC): RRC, PDCP y high RLC (función parcial de RLC) en la unidad central. Low RLC (función parcial de RLC), MAC, capa física y RF en la unidad distribuida.
- Opción 4 RLC-MAC: RRC, PDCP y RLC en la unidad central. MAC, capa física y RF en la unidad distribuida.
- Opción 5 (intra MAC): Ciertas partes de la capa MAC se llevan a la unidad distribuida junto con la capa física y RF, el resto de la capa MAC y las capas superiores en la unidad central.
- Opción 6 MAC-PHY: RRC, PDCP, RLC y MAC en la unidad central capa física y RF en la unidad distribuida.
- Opción 7 (intra PHY): Ciertas partes de la capa PHY se llevan a la unidad distribuida junto con la RF, el resto de la capa PHY y las capas superiores en la unidad central.
- Opción 8 PHY-RF: Las funciones de RF se llevan a la unidad distribuida, las capas superiores en la unidad central.

El 3GPP también contempla el empleo de la división funcional flexible, de donde se destacan las ventajas de la gestión de la carga, la optimización del rendimiento en tiempo real, adaptabilidad a diferentes casos de uso y la habilitación de *Network Function Virtualization* y *Software Defined Networking*.

Además, la elección de un nivel de división funcional fijo no es una solución eficiente ya que, entre otras cosas, no tiene en cuenta las variaciones en el tráfico a lo largo del día. Realizar esta selección de forma óptima es una tarea complicada y dependerá del caso de uso que se busque. Se deberá atender a diversos factores, por ejemplo: limitaciones de la red radio, interferencia inter-celular, tipos de servicios soportados, necesidad de soportar QoS por servicio, necesidad de soportar cierta densidad de usuarios y carga en una zona geográfica o capacidad para funcionar con redes de transporte de rendimientos diferentes.

Existen multitud de trabajos que tratan algunos de los aspectos contemplados a lo largo de este trabajo. En los siguientes párrafos se ejemplifican algunos estudios relacionados. Se puede encontrar información sobre la arquitectura C-RAN así como sus ventajas en [4, 6, 8-10, 15, 16]. En [4] se realiza un análisis del estado del arte del C-RAN, prestando especial atención a destacar las ventajas de esta arquitectura

y a los desafíos que en esta se presentan. Un estudio sobre las *Hyper-Dense Heterogeneous and Small cell Networks* se puede encontrar en [6], donde además se realiza una comparación de las mismas con *massive MIMO* y C-RAN. En [8] se muestra a la C-RAN como una opción para solventar el problema de los gastos crecientes de los operadores de red al mismo tiempo que para mejorar el rendimiento, pone gran interés en las diferentes fuentes de gastos de los operadores y como C-RAN puede ayudar a reducirlos, destaca además las ventajas de C-RAN y sus desafíos técnicos. Una estructura lógica de tres capas para mejorar la eficiencia del procesamiento centralizado, un algoritmo de emparejamiento de usuarios en recursos radio y una implementación de precodificación lineal óptima para gestionar la interferencia en C-RAN capaz de explotar las ventajas computacionales de la nube en paralelo son propuestos en [9], cuyos resultados muestran mejoras en la eficiencia espectral. Un estudio sobre la eficiencia energética y espectral, donde se presenta nuevamente a la C-RAN como una opción de peso para lograr mejoras energéticas se puede encontrar publicado en [10]. Una comparación de la cantidad de recursos necesarios en C-RAN frente a la arquitectura tradicional distribuida D-RAN, realizada en [15], muestra que los recursos necesarios son cuatro veces menores con la estructura centralizada. En [16] se tienen los resultados de una implementación de *uplink CoMP* en C-RAN, logrando mejoras del 20-50 % en zonas de buena cobertura y del 50-100 % en los bordes de la célula.

Se tienen investigaciones cubriendo la división funcional en [14, 17-21]. En la referencia [17] se hace un análisis de diferentes estudios que se han realizado sobre cada uno de los niveles de *split* demarcados en [14], dividiéndolos entre referencias teóricas, referencias de simulaciones y referencias prácticas; destaca que la mayoría de los trabajos encontrados son para los *splits* de capa física. Los autores de [18] analizan las desventajas de la interfaz CPRI, usada en la actualidad para conectar BBUs con RRHs y cuya especificación se puede encontrar en [22], centrándose en su poca viabilidad para el uso en C-RAN. Además, proponen una nueva interfaz *Next Generation Fronthaul Interface (NGFI)*, capaz de soportar diferentes niveles de *split* entre BBU y RRH en base a las características del enlace y capaz de desvincular el número de antenas de la capacidad necesaria en la red *fronthaul*. En [19] se tiene una comparación mediante métodos matemáticos y de simulación de los diferentes *splits* funcionales en términos de QoS y eficiencia energética y de coste, de donde concluyen que cuantas más funciones se centralicen mayor será la ganancia de multiplexado alcanzable por lo que el *split* 8 sería el más recomendable, especialmente para operadores con acceso a redes *fronthaul* de bajo coste. Las simulaciones de [20] muestran nuevamente que el *split* funcional 8, el más centralizado ya que solo deja en el RRH las funciones de RF, es el más eficiente energéticamente pero también el que presenta requisitos más altos en el *fronthaul*. Los autores de [21] proponen dos divisiones de las funciones de capa física que logran en las simulaciones reducciones del 30-40 % en el ancho de banda necesario en la conexión entre BBU y RRH respecto a la centralización total.

Análisis acerca de la división funcional flexible se pueden localizar en [23-29]. La referencia [23] se centra en la evolución de la arquitectura LTE para llevar a cabo el concepto de *Radio Access Network as a Service (RANaaS)*, donde se centralizan de forma parcial las funcionalidades en base a las necesidades y las características de la red. En [24] se propone un algoritmo para seleccionar el *split* funcional para cada célula, de tal forma que se minimice la interferencia inter–celular y el ancho de banda necesario en la red *fronthaul*. Un estudio que contrasta varias tecnologías, tanto cableadas como inalámbricas, que pueden ser implementadas en la red *fronthaul*, y muestra los requisitos de diferentes *splits* funcionales en términos de latencia y ancho de banda se puede encontrar en [25]. En [26], Koutsopoulos presenta un análisis teórico del problema de la selección de división funcional y *scheduling* con el objetivo de minimizar las latencias y propone soluciones al mismo. Una implementación de las soluciones descritas por Koutsopoulos y su evaluación en diferentes escenarios se puede encontrar en [27], donde se demuestra que, para los escenarios analizados, es preferible fijar la política de envío y optimizar el nivel de *split*. En [28] se tiene una propuesta de estructura para C-RAN que permite el uso de división funcional y Ethernet en la red *fronthaul*. Una implementación de un sistema funcional capaz de cambiar entre dos niveles de

split, PDCP-RLC y MAC-PHY, se propone y analiza en [29], donde afirman que es posible introducir cambios de *split* en tiempo de ejecución pero a costa de pérdidas de paquetes o mayores retardos.

A la vista de la literatura analizada, se puede asegurar que la selección dinámica de *split* es un tema de gran interés que aun se encuentra en fase de investigación.

Como se ha comentado anteriormente, en este trabajo se aborda la selección dinámica de *split* conjuntamente con el *scheduling* de tráfico. A continuación se describen las características principales de los *scheduler* implementados.

2.1. SCHEDULERS IMPLEMENTADOS

En el simulador se han implementado cuatro schedulers: *Round Robin*, *Max Cola*, *Weighted Fair Queuing* y *Min Drift-Plus-Penalty*. Los tres primeros son soluciones clásicas que no son capaces de aprovechar las ventajas del uso de diferentes niveles de *split*, sino que solo planifican qué cola tiene acceso a qué CPU. El *scheduler Min Drift-Plus-Penalty*, sin embargo gestiona tanto el orden de envío como la selección de *split*. A lo largo de esta sección se analizarán más en detalle las diferentes opciones introducidas.

2.1.1. ROUND ROBIN

El primer *scheduler* que se ha implementado para validar el correcto funcionamiento del simulador es el *Round Robin (RR)*. Su funcionamiento es el más simple de todos, se basa en asignar los recursos de procesado en cada *slot* temporal a cada una de las colas de forma ordenada sin atender a ningún parámetro de su configuración o estado. Por ejemplo en un sistema con un recurso de procesado y dos colas, dicho recurso se repartiría entre la primera y la segunda cola alternativamente, cambiando en cada *slot* como se puede apreciar en la Figura 2.2.

Slot	1	2	3	4	5
CPU 0:	Cola 1	Cola 2	Cola 1	Cola 2	Cola 1

Figura 2.2: Reparto de capacidad de procesado: 1 CPU, 2 Colas

En caso de tener un mayor número de recursos de procesado el reparto de los mismo se realizará también de forma ordenada (primer recurso para la primera cola, segundo para la segunda, etc.) manteniendo la posición de la última cola que obtuvo acceso al procesador entre *slots* consecutivos. A modo de ejemplo de un caso con este comportamiento se tiene el reparto de las CPUs de la Figura 2.2, para 2 CPUs y 3 colas.

Slot	1	2	3	4	5
CPU 0:	Cola 1	Cola 3	Cola 2	Cola 1	Cola 3
CPU 1:	Cola 2	Cola 1	Cola 3	Cola 2	Cola 1

Figura 2.3: Reparto de capacidad de procesado: 2 CPUs, 3 Colas

Mediante el uso de este *scheduler* se consigue un reparto equitativo de la capacidad de procesado entre las colas de un modo sencillo, por lo que se trata de un *scheduler* ideal para emplear en escenarios

simétricos, donde todas las colas tengan las mismas características. Como veremos en las posteriores secciones su comportamiento en escenarios asimétricos es muy precario ya que al asignar la misma capacidad de procesado a colas con características diferentes se suele dar el caso de que las colas de mayor tasa de entrada se saturan mientras que las de menor tasa ven sobredimensionados sus recursos. Otro problema de este *scheduler*, y que comparte con el WFQ, es que, al realizarse un reparto estático de los recursos, estos quedan sin utilizarse en caso de que la cola asignada se encuentre vacía.

2.1.2. Max Cola

El *scheduler Max Cola* tiene un funcionamiento muy simple: en cada *slot* asigna a la CPU la cola que tenga la mayor cantidad de bytes. El uso de este *scheduler* ayuda a lograr estabilidad en las colas, ya que al ceder la capacidad de procesado a las colas más llenas logra que las longitudes de las mismas tiendan a ser similares y lo más reducidas posible.

2.1.3. Weighted Fair Queuing

El *scheduler Weighted Fair Queuing (WFQ)* busca solventar uno de los problemas del *scheduler Round Robin*, la imposibilidad de realizar repartos de capacidad de procesado no equitativos entre las colas. Para ello aparece una magnitud que se ha denotado como "prioridad" y es la encargada de indicar al *scheduler* cómo realizar el reparto de capacidad entre las colas.

Como se ha comentado, en el *Round Robin* las colas se asignan alternativamente al recurso de procesado en cada *slot* temporal en un ratio 1:1, es decir en un escenario de dos colas se asignaba tantas veces la cola 1 como la cola 2. La prioridad se emplea para variar ese ratio. Si la prioridad de la cola 1 es de 2 y la de la cola 2 es de 1, se asignarán dos *slots* temporales a la cola 1 por cada uno que se asigne a la cola 2, de modo que, a efectos prácticos, la cola 1 obtendrá dos tercios de la capacidad de procesado y la cola 2 obtendrá un tercio.

El reparto de la capacidad se hará, por lo tanto, mediante la asignación a cada cola de un número de *slots* igual a la prioridad de la cola una vez eliminados todos los factores comunes con las prioridades de otras colas, es decir, tener dos colas con prioridades 4 y 2 es, a efectos del reparto de capacidad, lo mismo que tenerlas con prioridades 2 y 1. Por lo tanto las prioridades efectivas de las colas serán números primos entre si.

En la Figura 2.4 se puede ver, a modo de ejemplo, un reparto de una CPU entre dos colas de prioridades 4 y 2, respectivamente. Se aprecia que por cada dos *slots* en los que se concede acceso a la CPU a la cola 1 hay un *slot* en el que se concede acceso a la cola 2, siguiendo el orden mostrado en la figura.

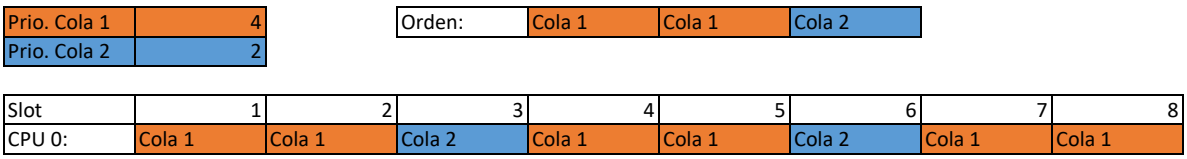


Figura 2.4: Reparto de capacidad de procesado: 1 CPU, 2 Colas

En la Figura 2.5 se tiene un ejemplo con dos CPUs y tres colas de prioridades 4, 2 y 1, respectivamente. Al no existir ningún factor común entre las tres prioridades no se pueden reducir, por lo tanto la cola 1 tendrá acceso a las CPUs en cuatro ocasiones por cada dos veces que tenga acceso la cola 2 y por cada vez que tenga acceso la cola 3, según el orden también presente en la figura.

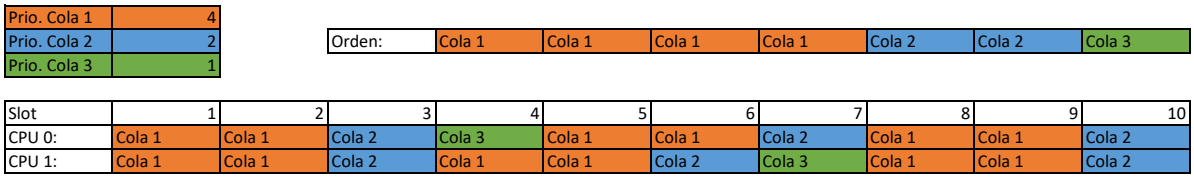


Figura 2.5: Reparto de capacidad de procesado: 2 CPUs, 3 Colas

Al igual que el *scheduler Round Robin* el reparto de los recursos se realiza de forma estática, en caso de dar con una cola vacía en el momento en el que se le asigna la CPU, los recursos quedarían sin utilizarse.

2.1.4. MIN DRIFT-PLUS-PENALTY

El *Min Drift-Plus-Penalty (MDPP)* se trata del más completo de los *schedulers*, ya que permite planificar no solo la asignación de colas a las CPUs sino también el nivel de *split* con el que se procesan estas y permite, además, tener en cuenta otras métricas como será en nuestro caso el retardo de los paquetes.

Para ello se emplea la técnica de optimización "Drift-plus-penalty" de Lyapunov, una poderosa herramienta que, mientras mantiene la estabilidad de colas reales o virtuales, permite la optimización de funciones objetivo [30].

Con esta técnica y al contar con un número reducido de combinaciones, se comprueban todos los posibles repartos de CPUs, *splits* y colas. Sin embargo, dicho número de combinaciones puede crecer muy rápidamente, al estar determinado por la siguiente expresión:

$$\#Combinaciones = (\#Colas + 1)^{\#CPUs} \cdot (\#NivelesSplit)^{\#CPUs}$$

Esta expresión se obtiene como sigue. El número de opciones de reparto de colas y CPUs se puede expresar como:

$$\#Opciones = (\#Colas + 1)^{\#CPUs}$$

A su vez, cada una de ellas se verá multiplicada la opción de tener un número de combinaciones de *splits* de:

$$\#Splits = (\#NivelesSplit)^{\#CPUs}$$

Por lo que el número total de combinaciones será:

$$\#Combinaciones = \#Opciones \cdot \#Splits = (\#Colas + 1)^{\#CPUs} \cdot (\#NivelesSplit)^{\#CPUs}$$

De esta expresión se puede deducir que el número de combinaciones seguirá un crecimiento exponencial con el número de CPUs y potencial con el número de colas y niveles de *split*. Por ejemplo para dos colas, dos CPUs y dos niveles de *split* el número de combinaciones será: $(2+1)^2 \cdot (2)^2 = 3^2 \cdot 2^2 = 9 \cdot 4 = 36$.

Para cada una de dichas combinaciones se evalúa la expresión mostrada en la Expresión 2.1 eligiendo la configuración con peso mínimo.

$$Vy_0(t) + \sum_{k=1}^{K} Q_k(t) \cdot [a_k(t) - b_k(t)] \tag{2.1}$$

La expresión anterior está compuesta por dos sumandos; el primero, $Vy_0(t)$, será el encargado de representar el peso del retardo en las decisiones; el segundo, $\sum_{k=1}^{K} Q_k(t) \cdot [a_k(t) - b_k(t)]$, se trata de la componente que trata de estabilizar las colas. Por lo tanto a la hora de tomar las decisiones el *scheduler* intentará compensar ambos términos eligiendo la opción que reduzca el retardo de los paquetes sin perjudicar la estabilidad de las colas.

3. IMPLEMENTACIÓN

El funcionamiento del sistema está basado en la toma de decisiones en un tiempo ranurado en *slots*. Al comienzo de cada *slot* se calcula el número de paquetes que han llegado durante el *slot* anterior a cada una de las colas y su tamaño, tras ello el *scheduler* toma una decisión en base a dicha llegada y el estado del sistema. Por último, se procede a procesar los paquetes de las colas conforme a la decisión tomada por el *scheduler* y se estima el retardo que tendrán dichos paquetes en el RRH correspondiente.

Las simulaciones, en las que se considera solo el sentido descendente de la comunicación, se realizan bajo el supuesto de que el cambio de *split* es instantáneo, sin cortes en el servicio, pérdidas ni incrementos en el retardo similar al propuesto en [29]. Al final de cada simulación se obtienen cinco ficheros de salida con los resultados que se indican a continuación:

- Fichero *in*: Almacena los datos de llegada de paquetes a cada una de las colas en cada *slot* de la simulación. Se escribe una fila del fichero por *slot* y por cola, cada una conteniendo la siguiente información en sus columnas:
  1. Slot
  2. Número de cola
  3. Número de paquetes que han entrado
  4. Número de bytes que han entrado
- Fichero *out*: Almacena en cada *slot* el estado de las colas, se tiene una primera columna con el *slot* y cinco columnas adicionales por cada cola en el sistema:
  1. Slot
  2. Número de paquetes perdidos cola 0
  3. Número de paquetes enviados cola 0
  4. Número de bytes enviados cola 0
  5. Número de paquetes en espera cola 0
  6. Número de bytes en espera cola 0
  7. Número de paquetes perdidos cola 1
  8. Número de paquetes enviados cola 1
  9. Número de bytes enviados cola 1
  10. Número de paquetes en espera cola 1
  11. Número de bytes en espera cola 1
- Fichero *delays*: Almacena los datos de todos los paquetes que salen del sistema. Por cada paquete que sale se añade una fila con las siguientes once columnas:
  1. ID del paquete
  2. Número de la CPU que lo ha procesado
  3. Número de la cola del paquete
  4. Slot de entrada
  5. Slot de salida
  6. Tamaño del paquete
  7. Retardo total

8. Retardo debido a la espera en la cola
9. Retardo debido al procesado en la CPU
10. Retardo debido al enlace entre BBU y RRH
11. Retardo debido al procesado y espera en el RRH

- Fichero *cpu*: Para cada *slot* almacena el uso de las CPUs del sistema, tiene tres columnas:
  1. Slot
  2. Número de la CPU
  3. Uso de la CPU
- Fichero *decision*: En el caso de hacer uso del *scheduler Min Drift-Plus-Penalty* se almacenan en este fichero cada una de las decisiones tomadas para cada *slot*, tiene tres columnas:
  1. Slot
  2. Decisión de reparto de CPUs
  3. Decisión de split

3.1. SISTEMA

El sistema está formado por diversos componentes, implementados cada uno en su propia clase de C++. Según el tipo de componente puede haber una única instancia o varias:

- 1 Generador de números aleatorios
- K Colas, cada una con sus parámetros de llegada de paquetes, tamaño, prioridad y RRH asociada.
- N CPUs, cada una con su capacidad.
- M RRHs, cada uno con su capacidad y su enlace con la BBU.
- 1 Scheduler

Se tiene, además de los componentes, la propia función *sistema* cuyo cometido es el de generar y englobar todas las clases anteriores.

El sistema contiene los parámetros básicos de configuración, algunos están fijados en el propio código como es el caso del tiempo de simulación y de *slot*, así como el nivel de *split* que se encuentra fijado de forma predeterminada para los *schedulers* que no tienen capacidad de elegirlo, el número y peso de los *splits*; y otros se cargan desde el fichero de inicialización, siendo este el caso del número de colas, CPUs y RRHs y sus características, así como el peso de las componentes del retardo. Seguidamente se indican los parámetros concretos utilizados:

- **# define TIEMPO_SIMULACION 10000:** Número de *slots* que durará la simulación.
- **# define TIEMPO_SLOT 1:** Duración en milisegundos de un *slot*.
- **extern int NUMERO_SPLITS:** Número de niveles de *split* contemplados en la simulación.
- **extern int PESOS_SPLITS[]:** Peso computacional de cada una de las tareas a repartir entre la CPU y el RRH.
- **extern int FIXED_SPLIT:** Nivel de *split* que se encuentra activo de forma predeterminada en los *schedulers* que no son capaces de tomar decisiones de reparto de *splits*.
- **extern int NUMERO_COLAS:** Número de colas en el sistema, el valor se carga desde el fichero de inicialización.

- **extern int NUMERO_CPUS:** Número de CPUs en el sistema, el valor se carga desde el fichero de inicialización.
- **extern int NUMERO_RRHS:** Número de RRHs en el sistema, el valor se carga desde el fichero de inicialización.
- **extern float peso_delay[]:** Pesos de cada una de las componentes del retardo, los valores se cargan desde el fichero de inicialización.

En el archivo de cabecera *sistema.h* se tiene la definición de la estructura que tendrán los paquetes a lo largo de la simulación. Cada paquete tendrá asociado su tamaño, el *slot* temporal en el que entra al sistema, un ID generado aleatoriamente al crear el paquete y un puntero a otro paquete que permitirá crear las listas enlazadas de las colas. Aparece también la declaración de la función *sistema*, cuyo único parámetro de entrada es el directorio donde se encuentra el fichero de inicialización y donde se generan los ficheros de salida tras la simulación:

- **int sistema(std::string directorio)**

La función *sistema* es llamada desde el flujo principal de ejecución y encapsula toda la simulación. En los Algoritmos 1 y 2 se tienen los pseudocódigos de la declaración de los componentes del sistema y la simulación, respectivamente. Es importante señalar que, a fin de que todos los paquetes puedan ser procesados como máximo en la duración de un *slot*, se limita el tamaño de los mismos a la máxima capacidad de la CPU. Esto es relevante para la generación del tamaño de los paquetes según la distribución exponencial negativa, donde dependiendo del valor medio que se tenga como parámetro podría ser, de forma más o menos probable, mayor que la capacidad de la CPU y por lo tanto no podría ser procesado en un único *slot*.

Algorithm 1: Función *sistema*: Declaración de componentes

```
/* Ficheros de salida                                                    */
open(infile);
open(outfile);
open(delayfile);
open(decisionfile);
open(CPUfile);
open(configFile);
srand(time(NULL));
// Semilla del generador de números aleatorios
/* Declaración de las componentes del sistema                            */
generador generador1;
std::vector<cpu>cpus;
// Vector con las CPUs
std::vector<enlace>enlaces;
// Vector con los enlaces
std::vector<rrh>rrhs;
// Vector con los RRHs
std::vector<cola>colas;
// Vector con las colas
generador1.configSistema(configFile);
/* Selección del scheduler                                               */
if ver_sched == 0 then
    scheduler = new rr_scheduler();
else if ver_sched == 1 then
    scheduler = new max_cola_scheduler();
else if ver_sched == 2 then
    scheduler = new wfq_scheduler();
else if ver_sched == 3 then
    scheduler = new mdpp_scheduler();
    read(configFile,&peso_delay[0],&peso_delay[1],&peso_delay[2]);
end
```

Algorithm 2: Función *sistema*: Simulación

```
/* Inicio simulación                                              */
for slot = 1 ; slot <= TIEMPO_SIMULACION ; slot++ do
    /* Generación del tráfico de entrada, a_k                     */
    for n_cola = 0 ; n_cola <NUMERO_COLAS ; n_cola++ do
        /* Distribución de Poisson a la entrada                   */
        generador1.getNpktsPoisson(n_cola.getTasa(),TIEMPO_SLOT);
        num_bytes_in = 0;
        for i = 0 ; i <num_pkts_in ; i++ do
            paquete = (pkt_t*)malloc(sizeof(pkt_t));
            paquete->id = rand();
            paquete->tiempo = slot;
            /* Tamaño del paquete exponencial negativo            */
            paquete->size = generador1.getSizeExp(n_cola.getTamanioPkts());
            if paquete->size >capacidad_CPU then
                paquete->size = capacidad_CPU;
            end
            paquete->next = NULL;
            num_bytes_in += paquete->size;
            /* Introducir paquete en la cola                      */
            retnum = n_cola.meterPkt(paquete);
            if retnum == -1 then
                numlost[n_cola]++;
                free(paquete);
            end
        end
        write(infile);
    end
    /* Toma de decisión del scheduler                             */
    scheduler->decision();
    /* Guardar información del slot en el fichero de salida       */
    write(outfile);
    for i = 0 ; i <NUMERO_COLAS ; i++ do
        write(outfile);
    end
end
return 0;
```

3.2. GENERADOR DE NÚMEROS ALEATORIOS

La clase *generador* se emplea para englobar varias funciones de generación de variables aleatorias y la función de configuración del sistema desde el fichero de inicialización (Anexo A).

- **int configSistema(FILE *configFile, int *NUMERO_COLAS, int *NUMERO_CPUS, int *NUMERO_RRHS, std::vector<cola>& colas_,std::vector<cpu>& cpus_, std::vector<enlace>& enlaces_,std::vector<rrh>& rrhs_):** Recibe como parámetro el fichero de inicialización y de-

vuelve el número de colas, CPUs y RRHs del sistema, así como los vectores que contienen los objetos ya inicializados con los valores especificados en el fichero.

Las variables aleatorias mencionadas anteriormente son las empleadas para obtener las realizaciones de los datos de entrada a las colas y se obtienen en cinco métodos de la clase:

- **double r2():** Devuelve una realización de una variable aleatoria uniforme entre 0 y 1.
- **int getSize():** Devuelve el tamaño de los paquetes siguiendo la distribución de los tamaños de trama Ethernet [31].
- **int getSizeExp(int media):** Devuelve el tamaño de los paquetes siguiendo la distribución exponencial negativa con la media que se pase a la función como parámetro.
- **int getNpktsUniforme(int max):** Devuelve una realización de una variable aleatoria uniforme entre 0 y el número máximo que se pase como parámetro.
- **int getNpktsPoisson(float tasa, float tiempo_slot):** Devuelve una realización de una variable aleatoria siguiendo una distribución de Poisson con tasa introducida como primer parámetro en un tiempo introducido como segundo parámetro.

3.3. COLAS

La clase *cola* es la encargada de almacenar los paquetes a la espera de ser procesados por las CPUs para posteriormente ser enviados a las RRHs. Se trata de colas FIFO, donde los primeros paquetes que llegan son también los primeros que salen.

Tienen cinco parámetros de configuración que se cargan desde el fichero de inicialización:

- Tamaño de la cola: Número máximo de paquetes que la cola puede almacenar. Si el valor es -1 la cola tendrá tamaño ilimitado.
- Tasa de entrada: Tasa de entrada de paquetes, medida en paquetes por *slot*, a la cola. Puede ser empleado tanto un valor fijo, como parámetro de una función de distribución de entrada de paquetes uniforme o de un proceso de Poisson.
- Prioridad: Peso de la cola para su uso en el reparto de capacidad del *scheduler* WFQ.
- Tamaño medio paquetes: Tamaño medio de los paquetes. Puede ser empleado tanto como valor fijo o como parámetro para la distribución de tamaño exponencial negativa.
- RRH asocidado: Puntero a la RRH asociada a la cola, para saber a cuál de las RRHs se deben enviar los paquetes una vez procesados y estimar su retardo.

La clase cuenta con ocho variables miembro, cinco de ellas para almacenar la configuración de la cola y otras tres para almacenar el estado:

- **int tamanio:** Almacena el valor de tamaño máximo de la cola, se emplea el valor -1 para crear una cola de tamaño ilimitado.
- **int numPaquetes:** Almacena el número de paquetes que se encuentran en ese momento en la cola.
- **int numBytes:** Almacena el número de bytes que se encuentran en ese momento en la cola.
- **int prioridad:** Almacena el valor de prioridad de la cola para su uso en el *scheduler* WFQ.
- **int tamanioPkts:** Almacena el tamaño medio de los paquetes, tanto para su uso como valor único, como para pasarlo como parámetro para la distribución de tamaño exponencial negativa.
- **float tasa:** Almacena el valor de la tasa de paquetes que llegan a la cola, tanto si es un número fijo o como el parámetro de la distribución de paquetes uniforme o proceso de Poisson.

- **pkt_t *inicioCola:** Almacena el puntero al primer paquete de la cola, el resto de paquetes están guardados a modo de lista enlazada de forma dinámica.
- **rrh *RRH_Cola:** Almacena el puntero a la RRH a la que se encuentra asociada la cola.

Dentro de la clase *cola* se tienen dieciséis métodos: tres de ellos, dos constructores y un destructor, para la creación o destrucción de instancias de la clase. La implementación de dos constructores permite tanto pasar los parámetros de configuración, como tener valores por defecto. A continuación se muestran los prototipos de estas funciones:

- **cola(int tamanio_, float tasa_, int prioridad_, int tamanioPkts_, rrh *RRH_Cola_) : tamanio(tamanio_), tasa(tasa_), prioridad(prioridad_), tamanioPkts(tamanioPkts_), RRH_Cola(RRH_Co numPaquetes(0), numBytes(0), inicioCola(NULL):** Construye una instancia de la clase cola con los valores que se le pasan a la función.
- **cola() : tamanio(0), tasa(0), RRH_Cola(), numPaquetes(0), numBytes(0), prioridad(0), tamanioPkts(0), inicioCola(NULL):** Construye una instancia de la clase cola vacía.
- $\sim$ **cola():** Elimina el objeto y libera la memoria asignada a los paquetes de forma dinámica.

Diez métodos que se emplean para leer o escribir las variables internas de la clase:

- **pkt_t *getPktPtr():** Devuelve el puntero al primer paquete de la cola, sin sacar este de la misma.
- **int getTamanio():** Devuelve el número máximo de paquetes que puede almacenar la cola.
- **float getTasa():** Devuelve la tasa de entrada de paquetes a la cola.
- **float setTasa(float tasa_):** Establece la tasa de entrada de paquetes a la cola.
- **int getBytesCola():** Devuelve el total de bytes de los paquetes en la cola.
- **int getNumPaquetes():** Devuelve el número de paquetes en la cola.
- **int getPrioridad():** Devuelve la prioridad de la cola.
- **int setPrioridad(int prioridad_):** Establece la prioridad de la cola.
- **int getTamanioPkts():** Devuelve el tamaño medio de los paquetes de la cola.
- **rrh *getRRHCola():** Devuelve un puntero a la RRH asociada a la cola.

Y por último, se tienen tres métodos específicos para el funcionamiento de la cola: uno para insertar paquetes, un segundo para sacarlos y el tercero es una modificación del empleado para obtener el número de bytes en la cola que tiene en cuenta el último *slot*:

- **pkt_t *cogerPkt():** Devuelve el puntero al primer paquete de la cola y saca el paquete de la misma, es la operación a realizar previa al procesado de los paquetes.
- **int meterPkt(pkt_t *paquete):** Introduce el paquete a la cola, si queda espacio. Tiene tres posibles valores de retorno: -1 si la cola se encuentra en su máximo tamaño y no ha podido ser añadido, 1 si el paquete ha podido ser añadido correctamente y queda espacio en la cola y 0 si el paquete se ha podido añadir correctamente pero la cola ha alcanzado su tamaño máximo.
- **int getBytesCola(int slot):** Devuelve el total de bytes de los paquetes en la cola que han llegado en el *slot* introducido como parámetro.

3.4. Recursos de Procesado

La clase *CPU* se emplea para representar las unidades de procesamiento del sistema. Tiene un único parámetro: la capacidad computacional, medida en bytes por segundo. Con este parámetro y el tamaño del paquete que se desea procesar podemos saber que el tiempo que tardará el paquete en ser procesado será su tamaño dividido entre la capacidad de la CPU, tal como se indica en la ecuación 3.1.

$$t_{CPU} = \frac{Tam.Pkt}{Capacidad_{CPU}} \tag{3.1}$$

Para realizar la implementación de los niveles de *split* se emplea una capacidad equivalente que varía para cada nivel de *split* a la hora de calcular el tiempo de procesado, tanto en las CPUs como en los RRHs. El cálculo de dicha capacidad se realiza de la siguiente manera:

Se han supuesto cinco tareas que deben ser repartidas entre la CPU y el RRH correspondiente. Como mínimo tanto CPU como RRH deben realizar una tarea, por lo que el número de niveles de *split* posibles será cuatro. Cada una de las tareas tiene un peso asignado, que será la capacidad computacional que supone realizar la propia tarea, es decir, si el peso es 1 equivale a procesar el paquete una vez, si el peso es 2 equivale a procesar el paquete dos veces. Por simplicidad se han supuesto todos los pesos iguales y de valor 1, tal y como se muestra en la Figura 3.1.

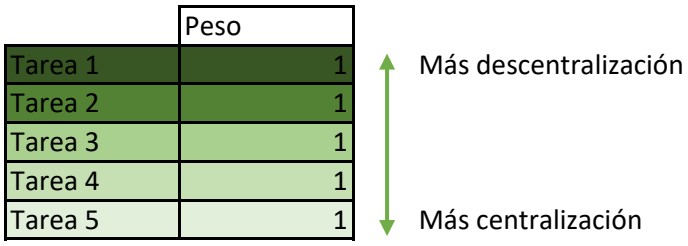


Figura 3.1: Pesos de las tareas

El nivel de *split* denota qué tareas se realizan en la CPU y cuales en el RRH, siendo el *split* 0 el más descentralizado al tener el mínimo número de tareas en la CPU y el máximo en el RRH, y el *split* 3 el más centralizado al tener la máxima cantidad de tareas en la CPU y la mínima en el RRH.

Para obtener la capacidad equivalente de la CPU se divide la capacidad nominal de esta entre la suma de los pesos de las tareas que deben realizarse en la misma para el nivel de *split* de interés, como se indica en la ecuación 3.2.

$$CapacidadEquivalenteCPU = \frac{CapacidadNominalCPU}{\sum_{i=1}^{NivelSplit} Peso_i} \tag{3.2}$$

En la Figura 3.2 se observan los cuatro niveles de *split* posibles y sus respectivas divisiones de tareas. Se puede apreciar que conforme aumenta el nivel de *split* se reduce el número de tareas en el RRH y aumenta en la CPU.

A modo de ejemplo se ha tomado una CPU con capacidad 20 000 y un RRH con capacidad 10 000 y se muestran en la Figura 3.3 los diferentes valores de su capacidad equivalente para cada *split*. Se puede ver que la capacidad suma presenta dos picos en los niveles de *split* de máxima centralización y máxima descentralización, siendo el mayor de ellos para el nivel que lleva más funciones al elemento con menor capacidad nominal.

		Peso
CPU	Tarea 1	1
RRH	Tarea 2	1
	Tarea 3	1
	Tarea 4	1
	Tarea 5	1

(a) Split 0

		Peso
	Tarea 1	1
CPU	Tarea 2	1
RRH	Tarea 3	1
	Tarea 4	1
	Tarea 5	1

(b) Split 1

		Peso
	Tarea 1	1
	Tarea 2	1
CPU	Tarea 3	1
RRH	Tarea 4	1
	Tarea 5	1

(c) Split 2

		Peso
	Tarea 1	1
	Tarea 2	1
	Tarea 3	1
CPU	Tarea 4	1
RRH	Tarea 5	1

(d) Split 3

Figura 3.2: Reparto de las tareas en los diferentes *splits*

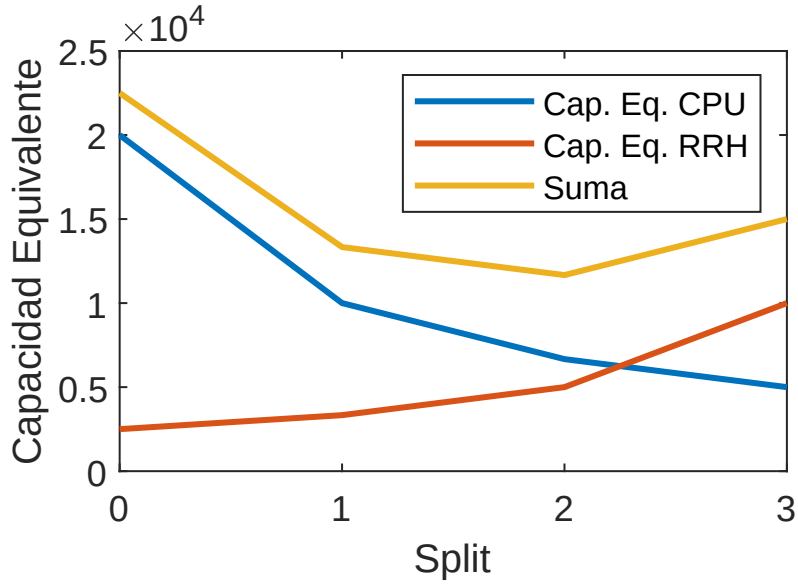


Figura 3.3: Capacidad de la CPU y la RRH para diferentes *splits*

El uso de CPU se calcula desde los *schedulers*, para ello se tiene la variable *t_cpu*, que se usa para mantener la cuenta del tiempo que la CPU ha estado ocupada en el *slot*. Cada vez que un paquete es procesado por la CPU el valor de *t_cpu* se incrementa con el valor del tiempo de procesado del paquete. El *scheduler* deja entrar paquetes a la CPU siempre que la suma del tiempo acumulado de uso de CPU, *t_cpu*, y el tiempo de procesado del siguiente paquete sea inferior a la duración del *slot*.

En la Figura 3.4 se tiene una representación gráfica de una CPU y dos colas, donde la cola 1 tiene acceso a la CPU. Cada uno de los colores representa el tiempo que tarda en procesarse un paquete, de forma que cuanto mayor es la franja de un color mayor es el tiempo de procesado del paquete en cuestión

debido a una mayor longitud del mismo. El tamaño de la celda debajo de la CPU 0 representa el tiempo de *slot*, y por lo tanto el valor máximo de *t_cpu*. Se aprecia que el último paquete de la cola 1 no llega a ser procesado en el *slot* representado al no tener tiempo suficiente.

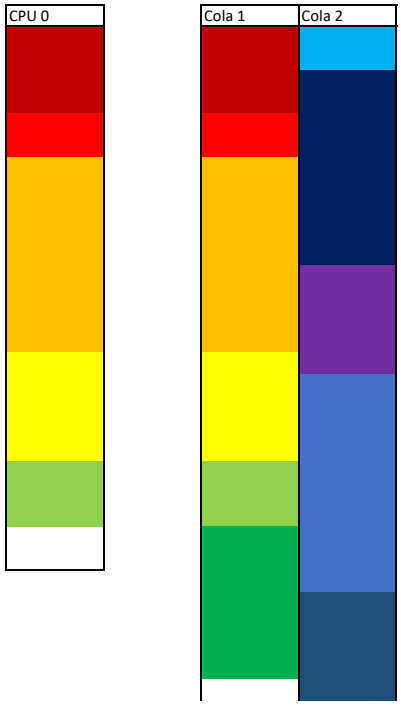


Figura 3.4: Uso de la CPU

La clase *CPU* cuenta con una única variable de clase:

- **float capacidad:** Almacena el valor de la capacidad nominal de la CPU.

Además, esta clase tiene tres métodos, uno para obtener el valor de la variable de clase y los otros dos para construir la clase, tanto inicializándola con un valor dado como sin inicializarla:

- **cpu(float capacidad_):capacidad(capacidad_):** Constructor de la clase que asigna a la variable capacidad el valor pasado como parámetro.
- **cpu():capacidad(0):** Constructor de la clase vacío que inicializa la capacidad a 0.
- **float getCapacidad():** Devuelve el valor de la capacidad de la CPU.

3.5. ENLACE

Para caracterizar los enlaces por los que se transmiten los paquetes entre la BBU y la RRH se ha creado la clase *enlace*. En el modelo empleado solo se tiene en cuenta un retardo fijo para todos los paquetes que atraviesan el enlace.

Se tiene una única variable de clase, para guardar el valor de dicho retardo:

- **float t_basico:** Tiempo básico de propagación de los paquetes a través del enlace.

Además se han implementado tres métodos, el primero para obtener el valor de la variable de clase y los dos siguientes para construir la clase, uno inicializándola a un valor indicado como parámetro y otro sin inicializarla:

- **enlace(float t_basico_):t_basico(t_basico_):** Constructor de la clase que asigna a la variable tiempo de propagación el valor pasado como parámetro.
- **enlace():t_basico(0):** Constructor de la clase vacío que inicializa el tiempo de propagación a 0.
- **float getPropagacion():** Devuelve el valor del tiempo de propagación del enlace.

3.6. REMOTE RADIO HEADS

En la clase *RRH* se incluye todo lo relativo a las Remote Radio Heads. Para estimar los retardos de los paquetes en el conjunto del RRH y su cola, estos se han modelado como un nodo M/M/1 siguiendo la notación de Kendall [32], un sistema con un único recurso y una cola de espera infinita.

Como sabemos de la teoría de la cola M/M/1, el tiempo de espera total en el sistema, es decir, la suma del tiempo de espera en la cola más el tiempo de procesado se calcula con la siguiente fórmula:

$$\overline{W_T} = \frac{1}{\mu - \lambda} \tag{3.3}$$

Siendo μ la tasa de muerte del sistema e igual al inverso de la media del tiempo de servicio, que se calcula como el tamaño medio de los paquetes de la cola entre la capacidad equivalente de la RRH:

$$\mu = \frac{1}{\overline{T_S}} = \frac{1}{\frac{\overline{Tam.Paquete}}{\overline{Cap.EquivalenteRRH}}} = \frac{Cap.EquivalenteRRH}{\overline{Tam.Paquete}} \tag{3.4}$$

Por otro lado, λ representa la tasa de llegada de paquetes al sistema de la RRH, siendo igual a la tasa de paquetes que salen de la CPU y, en media, igual a la tasa de entrada de la cola de la CPU, dado que el sistema, tal y como está configurado a efectos de este trabajo, no contempla la opción de que aparezcan pérdidas.

Con intención de tener en cuenta la carga de la RRH en *slots* anteriores se hace uso de una media móvil. En la configuración empleada para las simulaciones se tienen en cuenta los últimos tres *slots* para la capacidad y para la tasa se consideran los últimos cinco *slots*, en ambos casos se incluye el *slot* actual. Los valores a los que se inicializan los *arrays* empleados para almacenar los datos de los *slots* previos para calcular la media móvil son la tasa de la cola y la capacidad nominal de la RRH.

Para el cálculo de la capacidad equivalente de la RRH se han tomado los pesos 0.7 para ponderar la capacidad equivalente en el *slot* actual, 0.2 para el *slot* anterior y 0.1 para el *slot* previo a este. La Ecuación 3.5 se emplea para el cálculo de la capacidad equivalente que se usa posteriormente para obtener el retardo de la RRH:

$$\begin{aligned} Cap.EquivalenteRRH = 0{,}7 \cdot Cap.EquivalenteRRH(t) + 0{,}2 \cdot Cap.EquivalenteRRH(t-1) \\ + 0{,}1 \cdot Cap.EquivalenteRRH(t-2) \end{aligned} \tag{3.5}$$

Cada una de las capacidades equivalentes de la RRH se obtienen en su correspondiente *slot* de un modo análogo a como se hace con la capacidad de la CPU para los distintos *splits*, dividiendo la capacidad nominal de la RRH entre la suma de los pesos asociados a las tareas que corresponden a la RRH como se muestra en la Ecuación 3.6:

$$CapacidadEquivalenteRRH = \frac{CapacidadNominalRRH}{\sum_{i=NivelSplit+1}^{NumeroTareas} Peso_i} \tag{3.6}$$

Del mismo modo que con la capacidad, la tasa empleada para calcular el retardo se obtiene ponderando las tasas de cierto número de *slots* previos. En este caso se han tenido en cuenta cinco *slots*, con pesos iguales a su posición en el *array*, dividido entre la suma de todos los pesos para normalizarlo a 1, como se muestra en la Ecuación 3.7:

$$\lambda = \frac{5 \cdot \lambda(t)}{5+4+3+2+1} + \frac{4 \cdot \lambda(t-1)}{15} + \frac{3 \cdot \lambda(t-2)}{15} + \frac{2 \cdot \lambda(t-3)}{15} + \frac{1 \cdot \lambda(t-4)}{15} \tag{3.7}$$

Una vez obtenidos los valores de μ y λ se calcula el retardo debido a la RRH con la Ecuación 3.3. Se puede observar que en caso de que λ sea mayor que μ, es decir, que la tasa de llegada sea mayor que la tasa de salida, se tendría un retardo negativo. Si se da este caso el valor retornado será infinito.

La actualización de los *arrays* empleados para calcular la media móvil se realiza al final de cada *slot* temporal en el *scheduler*, que lleva la cuenta del número de paquetes y el número de bytes que son procesados en el *slot* para cada cola y la capacidad equivalente de la RRH para cada uso por las CPUs.

Se tienen, por lo tanto, dos matrices para almacenar los resultados durante la toma de decisión y que deben ser pasadas a la RRH al final del *slot*:

- **float tasaRRH[NUMERO_COLAS][2]:** Almacena, para cada cola, en la primera columna el número de paquetes que han sido procesados y en la segunda el número de bytes.
- **float capRRH[NUMERO_COLAS][NUMERO_CPUS]:** Almacena la capacidad equivalente de la RRH para cada cola y CPU.

En caso de que se conceda acceso a la misma cola en dos CPUs distintas con distinto nivel de *split* y por tanto distinta capacidad equivalente de RRH, se pasa al *array* de la RRH la media de las capacidades, tal y como se ven en el Algoritmo 3. Si la cola asignada a la RRH no ha tenido acceso a ninguna CPU en el *slot*, todos los valores de capacidad equivalente serán 0, y por lo tanto no se actualizará el *array* de capacidades de la RRH; sin embargo la tasa también será 0 y si será actualizada en el *array*.

Algorithm 3: Actualización de los *arrays* para la media del RRH

```
for n_cola = 0 ; n_cola <NUMERO_COLAS ; n_cola++ do
    capRRHmed = 0;
    cuentaMed = 0;
    for n_cpu = 0 ; n_cpu <NUMERO_CPUS ; n_cpu++ do
        if capRRH[n_cola][n_cpu] != 0 then
            capRRHmed += capRRH[n_cola][n_cpu];
            cuentaMed++;
        end
    end
    n_cola.getRRH()->addTasa(tasaRRH[n_cola]);
    if cuentaMed != 0 then
        n_cola.getRRH()->addCap(capRRHmed/cuentaMed);
    end
end
```

La clase *RRH* cuenta con cinco variables de clase, tres de ellas *arrays* para almacenar los valores empleados para el cálculo de las medias ponderadas, y una definición de una macro:

- **# define ventanaMedia 4:** Especifica el tamaño de los *arrays* que van a almacenar los valores de tasa y capacidad en los *slots* previos. El tamaño real de la ventana será ventanaMedia + 1, ya que el valor actual no se almacena hasta que se han realizado las operaciones.
- **float capacidad;** Variable que almacena el valor de la capacidad nominal de la RRH.
- **float tasaMedia[ventanaMedia]:** *Array* que almacena los valores previos de tasa de entrada al RRH.
- **float tamMedio[ventanaMedia]:** *Array* que almacena los valores previos de tamaño medio de los paquetes que llegan a la RRH.
- **float capMedia[ventanaMedia]:** *Array* que almacena los valores previos de capacidad equivalente de la RRH.
- **enlace enlace_rrh:** Variable que guarda el enlace asociado a la RRH.

Se tienen, además, ocho métodos de clase:

- **rrh(float capacidad_, enlace enlace_rrh_) : capacidad(capacidad_), enlace_rrh(enlace_rrh_), tasaMedia({0}), tamMedio({0}), capMedia({0}):** Constructor de la clase que inicializa la capacidad y en enlace a los valores que recibe como parámetro, dejando a inicializados a 0 los *arrays* de las medias.
- **rrh():capacidad(0),enlace_rrh(NULL),tasaMedia({0}),tamMedio({0}),capMedia({0}):** Constructor por defecto de la clase que inicializa todas las variables de clase a 0.
- **float calculaDelay(float tamMedioPkt, float tasa, float capEquivalente, float tamPkt):** Devuelve el retardo debido a la RRH siguiendo el método explicado anteriormente. Los parámetros *tasa* y *capEquivalente* son los empleados en el *slot* actual.
- **void addTasa(float *tasaRRH):** Actualiza los *arrays tasaMedia* y *tamMedio* añadiendo los elementos que recibe como parámetro y eliminando los más antiguos del *array*.
- **void addCap(float CapacidadEq):** Actualiza el *array capMedia* añadiendo el elemento que recibe como parámetro y eliminando el más antiguo.
- **void initMedia(float tasa, float capEq):** Inicializa todas las posiciones de los *arrays tasaMedia* y *capMedia* con los valores que se pasan como parámetro a la función.
- **float getCapacidad():** Retorna el valor de la variable capacidad.
- **float getTpropagacion():** Llama a la función *getPropagación()* del enlace asociado a la RRH y devuelve su valor de retorno.

3.7. SCHEDULERS

Todos los *schedulers* heredan de una clase base llamada *super_scheduler*, donde se tienen las variables y métodos de clase que son comunes a todos ellos. Se tienen tres variables de clase, que son punteros a los vectores con los componentes del sistema:

- **std::vector<cola>& colas:** Puntero al vector con las colas que forman parte del sistema.
- **std::vector<cpu>& cpus:** Puntero al vector con las CPUs que forman parte del sistema.
- **std::vector<rrh>& rrhs:** Puntero al vector con los RRHs que forman parte del sistema.

En el caso del *super_scheduler* se tienen siete métodos de clase, que, exceptuando el método *decision*, serán iguales para los cuatro *schedulers* derivados de esta clase:

- **super_scheduler(std::vector<cola>& colas_,std::vector<cpu>& cpus_,std::vector<rrh>& rrhs_) : colas(colas_),cpus(cpus_),rrhs(rrhs_):** Constructor de la clase, toma como parámetros los punteros a los vectores con los componentes del sistema.
- **std::vector<cola>& getColas():** Devuelve el puntero al vector con las colas del sistema.
- **std::vector<cpu>& getCPUs():** Devuelve el puntero al vector con las CPUs del sistema.
- **std::vector<rrh>& getRRHs():** Devuelve el puntero al vector con las RRHs del sistema.
- **float capEqSplitCPU(int split,float capacidad):** Función que toma como parámetros el nivel de *split* y la capacidad nominal de la CPU y devuelve la capacidad equivalente de la CPU para dicho *split*.
- **float capEqSplitRRH(int split,float capacidad):** Función que toma como parámetros el nivel de *split* y la capacidad nominal de la RRH y devuelve la capacidad equivalente de la RRH para dicho *split*.
- **virtual void decision(std::string directorio, FILE *delays, int slot, int *numout, float *bytesout):** Se trata de la función más importante del *scheduler* y variará dependiendo del tipo de *scheduler* que herede la clase. Es la función encargada de ver el estado del sistema, tomar las decisiones de reparto de CPUs y niveles de *split* y procesar los paquetes.

A continuación se detallarán las especializaciones de la clase *super_scheduler* que implementan los algoritmos concretos que se han evaluado.

3.7.1. ROUND ROBIN

Se trata del más sencillo de los *schedulers*, cuenta con una única variable de clase que se emplea para saber cuál es la cola que va a recibir acceso a la CPU, además de las heredadas del *super_scheduler*:

- **int cola_actual:** Almacena el número de la cola a la que la CPU está dando servicio.

Se tienen dos métodos de clase, el constructor de la clase heredada que llama a su vez al constructor de la clase base y la propia implementación de la función *decision()* para el *scheduler Round Robin*:

- **rr_scheduler(std::vector<cola>& colas_,std::vector<cpu>& cpus_,std::vector<rrh>& rrhs_) : super_scheduler(colas_,cpus_,rrhs_),cola_actual(0):** Constructor de la clase heredada, toma como parámetros los punteros a los vectores con las colas, CPUs y RRHs que componen el sistema y se los pasa como parámetro al constructor de la clase base, además inicializa la variable *cola_actual* a 0.
- **void decision(std::string directorio, FILE *delays, int slot, int *numout, float *bytesout):** Se trata de la función que da utilidad al *scheduler*, provee a las distintas colas acceso a la CPU de forma ordenada. En los Algoritmos 4 y 5 se muestra el funcionamiento de la misma.

Como el *scheduler Round Robin* no es capaz de tomar decisiones de *split*, el nivel de *split* se encuentra fijado en la configuración del sistema, siendo el mismo para todos los *slots*.

Algorithm 4: Función *decision()* del *scheduler Round Robin*: Inicialización

```
open(CPUfile);
/* Inicializar a 0 tiempo de CPU, tasa y capacidad del RRH     */
for i = 0 ; i <NUMERO_CPUS ; i++ do
    t_cpu[i] = 0;
end
for i = 0 ; i <NUMERO_COLAS ; i++ do
    tasaRRH[i][0] = 0;
    tasaRRH[i][1] = 0;
    for j = 0 ; j <NUMERO_CPUS ; j++ do
        capRRH[i][j] = 0;
    end
end
```

Algorithm 5: Función *decision()* del *scheduler Round Robin*: Decisión y procesado

```
/* Procesar paquetes para todas las CPUs                                    */
for n_cpu = 0 ; n_cpu <NUMERO_CPUS ; n_cpu++ do
    if cola_actual.getNumPaquetes() == 0 then
        cola_actual++;
        if cola_actual == NUMERO_COLAS then
            cola_actual = 0;
        end
        continue;
    end
    /* Capacidades equivalentes para el split fijado                        */
    capacidad_CPU = capEqSplitCPU(FIXED_SPLIT,n_cpu.getCapacidad());
    capacidad_RRH = capEqSplitRRH(FIXED_SPLIT,cola_actual.getRRH()->getCapacidad());
    capRRH[cola_actual][n_cpu] = capacidad_RRH;
    while t_cpu[n_cpu] + cola_actual.getPktPtr()->size/capacidad_CPU <= TIEMPO_SLOT
     do
        paquete = cola_actual.cogerPkt();
        /* Añadir tiempo de procesado al tiempo de CPU                      */
        t_cpu[n_cpu] += paquete->size/capacidad_CPU;
        /* Calcular los retardos                                            */
        tiempoCOLA = slot - paquete->tiempo;
        tiempoCPU = paquete->size/capacidad_CPU;
        tiempoPROP = cola_actual.getRRH()->getTpropagacion();
        tiempoRRH = cola_actual.getRRH()->calculaDelay();
        tasaRRH[cola_actual][0] ++;
        tasaRRH[cola_actual][1] += paquete->size;
        /* Escribir fichero de retardos                                     */
        write(delayfile);
        free(paquete);
        if cola_actual.getNumPaquetes() == 0 then
            break;
        end
    end
    cola_actual++;
    if cola_actual == NUMERO_COLAS then
        cola_actual = 0;
    end
end
/* Actualizar tasa y capacidad RRH (Algoritmo 3)                            */
/* Escribir fichero de uso de CPU                                           */
for n_cpu = 0 ; n_cpu <NUMERO_CPUS ; n_cpu++ do
    write(CPUfile);
end
```

3.7.2. MAX COLA

El segundo *scheduler* implementado se trata del *Max Cola*. Su funcionamiento consiste en comprobar cual es la cola con mayor número de bytes y proveerla de acceso a la CPU. En caso de que se tenga más de una CPU se realiza nuevamente la comprobación de la cola más llena teniendo en cuenta los paquetes que ha sacado de la cola la CPU previa.

Al conceder acceso a la CPU a la cola más larga se logra un comportamiento de máxima estabilidad en las colas, funcionando de un modo muy similar al *scheduler Min Drift-Plus-Penalty* si este no tiene en cuenta los retardos a la hora de calcular la función de coste.

Como la comprobación para encontrar la cola más larga se realiza en cada *slot* no es necesario almacenar información del estado del *scheduler* por lo que no se tienen variables de clase, como si ocurre con el *scheduler Round Robin* al tener que guardar el número de la cola que tiene acceso a la CPU.

La búsqueda de la cola más larga se hace sin tener en cuenta los paquetes que han entrado en el mismo *slot* en el que se realiza la comprobación, ya que no pueden ser procesados en el *slot* en el que llegan al sistema.

Existen dos métodos de clase: el constructor y la implementación de la propia función decisión del *scheduler*.

- **max_cola_scheduler(std::vector<cola>& colas_,std::vector<cpu>& cpus_,std::vector<rrh>& rrhs_) : super_scheduler(colas_,cpus_,rrhs_):** Constructor de la clase heredada, toma como parámetros los punteros a los vectores con las colas, CPUs y RRHs que componen el sistema y se los pasa como parámetro al constructor de la clase base.
- **void decision(std::string directorio, FILE *delays, int slot, int *numout, float *bytesout):** Función que confiere utilidad al *scheduler*, provee a las distintas colas acceso a la CPU atendiendo únicamente a la longitud de estas. En los Algoritmos 6 y 7 se muestra el funcionamiento de la misma.

Algorithm 6: Función *decision()* del *scheduler Max Cola*: Inicialización

```
open(CPUfile);
/* Inicializar a 0 tiempo de CPU, tasa y capacidad del RRH     */
for i = 0 ; i <NUMERO_CPUS ; i++ do
    t_cpu[i] = 0;
end
for i = 0 ; i <NUMERO_COLAS ; i++ do
    tasaRRH[i][0] = 0;
    tasaRRH[i][1] = 0;
    for j = 0 ; j <NUMERO_CPUS ; j++ do
        capRRH[i][j] = 0;
    end
end
```

Algorithm 7: Función *decision()* del *scheduler Max Cola*: Decisión y procesado

```
/* Procesar paquetes para todas las CPUs                                    */
for n_cpu = 0 ; n_cpu <NUMERO_CPUS ; n_cpu++ do
    /* Buscar cola más larga                                                */
    max_bytes = 0;
    n_cola_max = -1;
    for n_cola = 0 ; n_cola <NUMERO_COLAS ; n_cola++ do
        if n_cola.getBytesCola() - n_cola.getBytesCola(slot) >max_bytes then
            max_bytes = n_cola.getBytesCola() - n_cola.getBytesCola(slot);
            n_cola_max = n_cola;
        end
    end
    if n_cola_max == -1 then
        break;
    end
    /* Capacidades equivalentes para el split fijado                        */
    capacidad_CPU = capEqSplitCPU(FIXED_SPLIT,n_cpu.getCapacidad());
    capacidad_RRH = capEqSplitRRH(FIXED_SPLIT,n_cola_max.getRRH()->getCapacidad());
    capRRH[n_cola_max][n_cpu] = capacidad_RRH;
    while t_cpu[n_cpu] + n_cola_max.getPktPtr()->size/capacidad_CPU <= TIEMPO_SLOT
     do
        paquete = n_cola_max.cogerPkt();
        /* Añadir tiempo de procesado al tiempo de CPU                      */
        t_cpu[n_cpu] += paquete->size/capacidad_CPU;
        /* Calcular los retardos                                            */
        tiempoCOLA = slot - paquete->tiempo;
        tiempoCPU = paquete->size/capacidad_CPU;
        tiempoPROP = n_cola_max.getRRH()->getTpropagacion();
        tiempoRRH = n_cola_max.getRRH()->calculaDelay();
        tasaRRH[n_cola_max][0] ++;
        tasaRRH[n_cola_max][1] += paquete->size;
        /* Escribir fichero de retardos                                     */
        write(delayfile);
        free(paquete);
        if n_cola_max.getNumPaquetes() == 0 then
            break;
        end
    end
end
/* Actualizar tasa y capacidad RRH (Algoritmo 3)                            */
/* Escribir fichero de uso de CPU                                           */
for n_cpu = 0 ; n_cpu <NUMERO_CPUS ; n_cpu++ do
    write(CPUfile);
end
```

3.7.3. WEIGHTED FAIR QUEUING

Se podría considerar al *Weighted Fair Queuing* como una generalización del *Round Robin* que permite un reparto desigual de los recursos de procesado entre las colas, por lo que presenta una mayor complejidad.

El reparto se logra mediante la creación de un *array* con el orden de prioridades, de un modo similar al ejemplificado en la Sección 2.1.3. A modo de ejemplo, tomaremos un sistema compuesto por 3 colas con las siguientes prioridades:

$$prioridades = [24, 12, 6]$$

A continuación la función *mcd(prioridades)* devuelve el máximo común divisor de todos los elementos del *array* $prioridades$. Se divide cada elemento del *array* entre el Máximo Común Divisor (MCD):

$$prioridades' = \frac{[24, 12, 6]}{6} = [4, 2, 1]$$

Se calcula el tamaño del *array* que va a contener el orden de las colas como la suma de las prioridades una vez eliminados los divisores comunes:

$$tamanio\_orden = \sum_{i=0}^{NUMERO\_COLAS-1} prioridades[i];$$

Por último se crea el *array* con el orden repitiendo el número de la cola tantas veces como sea el valor de su prioridad una vez reducida:

$$orden = [0, 0, 0, 0, 1, 1, 2]$$

El *scheduler WFQ* se encarga de mantener un puntero que recorre ordenadamente ese *array* concediendo acceso a la CPU a la cola que se indique en cada posición. Una vez se ha concedido acceso a la CPU el puntero avanza a la siguiente posición.

Se tienen cuatro variables de clase:

- **int cola_actual:** Almacena el número de la cola a la que la CPU está dando servicio.
- **int tamanio_orden:** Almacena el tamaño del *array* que contiene el orden de las colas.
- **int posicion_orden:** Puntero a la posición del *array* de orden que tiene acceso a la CPU.
- **int *orden:** *Array* de orden de las colas.

Existen cuatro métodos de clase, el constructor de la clase, la implementación de la función *decision()* y las dos funciones para eliminar los factores comunes de las prioridades:

- **wfq_scheduler(std::vector<cola>& colas_,std::vector<cpu>& cpus_,std::vector<rrh>& rrhs_) : super_scheduler(colas_,cpus_,rrhs_),cola_actual(0),tamanio_orden(0),posicion_orden(0):** Constructor de la clase heredada que toma como parámetros los punteros a los vectores con las colas, CPUs y RRHs que componen el sistema y se los pasa como parámetro al constructor de la clase base, además se inicializan a 0 las variables de clase encargadas de mantener del orden de las colas.
- **void decision(std::string directorio, FILE *delays, int slot, int *numout, float *bytesout):** Función que da utilidad al scheduler, provee acceso a las CPUs siguiendo el orden indicado en el *array orden*. En los Algoritmos 8 y 9 se muestra el funcionamiento de la misma.

- **int mcd(int *prioridades):** Devuelve el Máximo Común Divisor de los elementos del *array* pasado como parámetro.
- **void reduce_mcd(int *prioridades):** Elimina los factores comunes del *array* pasado como parámetro.

Algorithm 8: Función *decision()* del *scheduler Weighted Fair Queuing*: Inicialización

```
open(CPUfile);
/* Generar array de posiciones                                    */
if tamanio_orden == 0 then
    for i = 0 ; i <NUMERO_COLAS ; i++ do
        prioridades[i] = colas[i].getPrioridad();
    end
    reduce_mcd(prioridades);
    for i = 0 ; i <NUMERO_COLAS ; i++ do
        tamanio_orden += prioridades[i];
    end
    posicion_orden = 0;
    for i = 0 ; i <NUMERO_COLAS ; i++ do
        for j = prioridades[i] ; j >0 ; j– do
            orden[posicion_orden] = i;
            posicion_orden++;
        end
    end
    posicion_orden = 0;
end
/* Inicializar a 0 tiempo de CPU, tasa y capacidad del RRH        */
for i = 0 ; i <NUMERO_CPUS ; i++ do
    t_cpu[i] = 0;
end
for i = 0 ; i <NUMERO_COLAS ; i++ do
    tasaRRH[i][0] = 0;
    tasaRRH[i][1] = 0;
    for j = 0 ; j <NUMERO_CPUS ; j++ do
        capRRH[i][j] = 0;
    end
end
```

Algorithm 9: Función *decision()* del *scheduler Weighted Fair Queuing*: Decisión y procesado

```
/* Procesar paquetes para todas las CPUs                                    */
for n_cpu = 0 ; n_cpu <NUMERO_CPUS ; n_cpu++ do
    /* Elegir cola según indique el array de posiciones                     */
    if colas[orden[posicion_orden]].getNumPaquetes() == 0 then
        posicion_orden++; if posicion_orden == tamanio_orden then
            posicion_orden = 0;
        end
        continue;
    end
    cola_actual = orden[posicion_orden];
    /* Capacidades equivalentes para el split fijado                        */
    capacidad_CPU = capEqSplitCPU(FIXED_SPLIT,n_cpu.getCapacidad());
    capacidad_RRH = capEqSplitRRH(FIXED_SPLIT,cola_actual.getRRH()->getCapacidad());
    capRRH[cola_actual][n_cpu] = capacidad_RRH;
    while t_cpu[n_cpu] + cola_actual.getPktPtr()->size/capacidad_CPU <= TIEMPO_SLOT
     do
        paquete = cola_actual.cogerPkt();
        /* Añadir tiempo de procesado al tiempo de CPU                      */
        t_cpu[n_cpu] += paquete->size/capacidad_CPU;
        /* Calcular los retardos                                            */
        tiempoCOLA = slot - paquete->tiempo;
        tiempoCPU = paquete->size/capacidad_CPU;
        tiempoPROP = cola_actual.getRRH()->getTpropagacion();
        tiempoRRH = cola_actual.getRRH()->calculaDelay();
        tasaRRH[cola_actual][0] ++;
        tasaRRH[cola_actual][1] += paquete->size;
        /* Escribir fichero de retardos                                     */
        write(delayfile);
        free(paquete);
        if cola_actual.getNumPaquetes() == 0 then
            break;
        end
    end
end
/* Actualizar tasa y capacidad RRH (Algoritmo 3)                            */
/* Escribir fichero de uso de CPU                                           */
for n_cpu = 0 ; n_cpu <NUMERO_CPUS ; n_cpu++ do
    write(CPUfile);
end
```

3.7.4. MIN DRIFT-PLUS-PENALTY

Se trata del *scheduler* más complejo de los presentados, pues además de conceder acceso a las CPUs debe elegir el nivel de *split* entre la CPU y el RRH. Se basa en una función de coste que se evalúa para todas las posibles opciones para elegir la que presente menor coste.

La función de coste tiene dos componentes, una que atiende a la estabilidad de las colas y otra que atiende al retardo. La componente de estabilidad se obtiene de la Ecuación 3.8, donde se tiene que K es el número total de colas, $Q_k(t)$ es la longitud en bytes de la cola k para el *slot* t, $a_k(t)$ son los bytes que han llegado a la cola k en el *slot* t y $b_k(t)$ son los bytes que pueden ser sacados de la cola k en el *slot* t en la opción para la que se está evaluando la función, y dependerá tanto de el reparto de las CPUs como del nivel de *split* que se asigne:

$$CompEstabilidad = \sum_{k=1}^{K} Q_k(t) \cdot [a_k(t) - b_k(t)] \tag{3.8}$$

La componente debida al retardo se calcula con la Ecuación 3.9. Se tienen unos coeficientes que se han denotado como *pesos* para asignar mayor o menor importancia a las distintas componentes del retardo de forma que primen o no sobre la estabilidad de las colas.

$$CompRetardo = pesoColas \cdot delayColas + pesoCPU \cdot delayCPU + pesoRRH \cdot delayRRH \tag{3.9}$$

Los valores de los pesos se cargan desde el fichero de inicialización al iniciar la ejecución. Por último el valor de la opción se calcula como la suma de ambas componentes.

$$CosteOpcion = CompEstabilidad + CompRetardo \tag{3.10}$$

En caso de que la componente de retardo sea infinita para todas las opciones, esto es, que el *delay* debido al RRH es infinito para todos los repartos de CPU y *splits* posibles, se toma la opción que proporciona mayor estabilidad.

Para recorrer todos los posibles repartos de colas en las CPUs, que como se explicó en la Sección 2.1.4 son en total $\#Opciones = (\#Colas + 1)^{\#CPUs}$, se emplea la función *opcionToTurno()*, que a partir de un número de opción llena un *array* con tantas posiciones como CPUs tenga el sistema con la cola asignada a cada CPU. Esta conversión se basa en un cambio de base del número de opción a base número de colas + 1, ya que se puede tomar la decisión de dejar la CPU sin cola asignada.

Por ejemplo, para un sistema con dos colas y dos CPUs el número de opciones será:

$$\#Opciones = (\#Colas + 1)^{\#CPUs} = (2 + 1)^2 = 3^2 = 9$$

De este modo, para comprobar todas las opciones se tendrá un ciclo que recorra el número de opción desde 0 hasta 8 y transforme cada uno de ellos a base 3, donde el primer dígito se corresponde con la cola asignada a la primera CPU y el segundo con la cola asignada a la segunda CPU. En la Figura 3.5 se muestran todas las opciones y su cambio de base.

Opción	CPU 1	CPU 0
0	0	0
1	0	1
2	0	2
3	1	0
4	1	1
5	1	2
6	2	0
7	2	1
8	2	2

Figura 3.5: Opciones para 2 CPUs y 2 colas

De un modo similar y siguiendo el ejemplo anterior, el número de combinaciones de *split* se calcula del siguiente modo:

$$\#Splits = (\#NivelesSplit)^{\#CPUs} = 4^2 = 16 \tag{3.11}$$

En este caso se emplea la función *opcionToSplit()* para transformar el número de *split* en el *array* que asigna un nivel de *split* a cada CPU, recorriendo las opciones de *split* desde 0 hasta 15. La base usada para el cambio es el número de niveles de *split*, que en este caso es igual a 4. En la Figura 3.6 se muestran todas las opciones de *split* y su cambio de base, donde el primer dígito será el nivel de *split* desde el nivel 0 al 3 para la primera CPU y el segundo dígito para la segunda.

Opción	CPU 1	CPU 0
0	0	0
1	0	1
2	0	2
3	0	3
4	1	0
5	1	1
6	1	2
7	1	3
8	2	0
9	2	1
10	2	2
11	2	3
12	3	0
13	3	1
14	3	2
15	3	3

Figura 3.6: Opciones de split para 2 CPUs y 4 niveles de split

Para cada *opción* se recorren todas las opciones de *split* en un segundo ciclo anidado al primero por lo que al final de los dos ciclos se han comprobado todas las combinaciones posibles, evaluando la función de coste para todas ellas. Una vez se obtienen todos los costes se elige la *opción* con el menor de ellos y se pasa a procesar los datos según lo indiquen dicha *opción* y nivel de *split*.

Los retardos que se emplean para obtener la componente de retardo del coste en cada opción se obtienen de la función *calculaDelay()*, cuya salida es la suma de los retardos de todos los paquetes que se puedan sacar con la configuración que se pasa a la función como parámetro, divididos en sus tres componentes.

En la función se reproduce cuál sería el resultado de la configuración de *opción* y nivel de *split* que recibe como parámetro sin modificar las colas y se devuelve un *array* con cuatro componentes: el retardo debido a la espera en las colas, el retardo debido al tiempo de procesado en la CPU, el retardo debido a la RRH y el número de bytes de salida.

Se tiene una única variable de clase que se emplea, al igual que en los *schedulers* anteriores, para guardar el número de la cola que se encuentra siendo procesada por la CPU:

- **int cola_actual:** Almacena el número de la cola que se está procesando por la CPU.

Los métodos de la clase son cinco: un constructor, las dos funciones de cambio de *opción* a nivel de *split* y turno de cola, la función de calculo del retardo y la función *decision()* propia del *scheduler*. A continuación se detalla cada uno de ellos:

- **mdpp_scheduler(std::vector<cola>& colas_,std::vector<cpu>& cpus_,std::vector<rrh>& rrhs_) : super_scheduler(colas_,cpus_,rrhs_),cola_actual(0):** Constructor de la clase heredada que toma como parámetros los punteros a los vectores con las colas, CPUs y RRHs que componen el sistema y se los pasa como parámetro al constructor de la clase base, además se inicializa a 0 la variable *cola_actual*.
- **void opcionToTurno(int *turno_cpu,int opcion):** Realiza el cambio del número de *opción* que recibe como parámetro al reparto de colas de cada CPU. Devuelve un *array* con tantas posiciones como CPUs se tengan en el sistema, siendo el contenido cada una de ellas el número de cola indexado desde 1 que debe ser procesado por la CPU en dicha posición.
- **void opcionToSplit(int *split_cpu,int opcion):** Realiza el cambio del número de *opción* que recibe como parámetro al reparto de niveles de *split* de cada CPU. Devuelve un *array* con tantas posiciones como CPUs se tengan en el sistema, siendo el contenido cada una de ellas el nivel de *split* de la CPU en dicha posición.
- **void calculaDelay(int *turno_cpu, int *split_cpu, int slot, float *delay_dec_split):** Función que simula el resultado de la salida para el reparto de CPUs y niveles de *split* pasados como parámetro y devuelve la suma de los retardos de todos los paquetes que saldrían del sistema bajo esas condiciones.
- **void decision(std::string directorio, FILE *delays, int slot, int *numout, float *bytesout):** Función que da utilidad al *scheduler*, provee acceso a las CPUs según ordene la opción de coste mínimo. El funcionamiento de la misma se ha subdividido en dos partes: en los Algoritmos 10 y 11 se muestra el pseudocódigo del cálculo y búsqueda de la opción mínima y en los Algoritmos 12 y 13 se muestra el procesado de los paquetes según indique la opción mínima.

Algorithm 10: Función *decision()* del *scheduler Min Drift-Plus-Penalty*: Cálculo opciones

```
/* Calcular a_k para el slot actual                                    */
for i = 0 ; i <NUMERO_COLAS ; i++ do
    a_k[i] = colas[i].getBytesCola(slot);
end
n_opciones = pow(NUMERO_COLAS + 1,NUMERO_CPUS);
n_splits = pow(NUMERO_SPLITS,NUMERO_CPUS);
/* Recorrer todas las opciones                                         */
for opcion = 0 ; opcion <n_opciones ; opcion++ do
    opcionToTurno(turno_cpu,opcion);
    /* Recorrer todos los splits para cada opción                      */
    for split = 0 ; split <n_splits ; split++ do
        opcionToSplit(split_cpu,split);
        for n_cola = 0 ; n_cola <NUMERO_COLAS ; n_cola++ do
            /* Calcular Q_k y b_k para la opción y split               */
            Q_k[n_cola] = colas[n_cola].getBytesCola() - a_k[n_cola];
            b_k[n_cola] = 0;
            for n_cpu = 0 ; n_cpu <NUMERO_CPUS ; n_cpu++ do
                if turno_cpu[n_cpu] == n_cola + 1 then
                    b_k[n_cola] +=
                      capEqSplitCPU(split_cpu[n_cpu],cpus[n_cpu].getCapacidad());
                end
            end
            /* Calcular componente de estabilidad para la cola k      */
            sum_k[opcion][split][n_cola] = Q_k[n_cola] * (a_k[n_cola] - b_k[n_cola]);
        end
        /* Calcular componente de estabilidad como la suma de la
           misma para cada cola                                        */
        sum_mdpp[opcion][split][0] = 0;
        for n_cola = 0 ; n_cola <NUMERO_COLAS ; n_cola++ do
            sum_mdpp[opcion][split][0] += sum_k[opcion][split][n_cola];
        end
        /* Calcular componente del retardo si los pesos son
           distintos de 0                                              */
        if !(peso_delay[0] == 0 && peso_delay[1] == 0 && peso_delay[2] == 0) ) then
            calculaDelay(turno_cpu,split_cpu,slot,delay_dec_split);
        end
        if peso_delay[0] == 0 && peso_delay[1] == 0 && peso_delay[2] == 0 then
            sum_mdpp[opcion][split][1] = 0;
        else
            sum_mdpp[opcion][split][1] = ( peso_delay[0] * delay_dec_split[0] + peso_delay[1]
              * delay_dec_split[1] + peso_delay[2] * delay_dec_split[2] );
        end
    end
end
```

Algorithm 11: Función *decision()* del *scheduler Min Drift-Plus-Penalty*: Búsqueda de la opción mínima

```
/* Tomar como opción mínima inicial la opción 0                    */
min_opcion[0] = 0;
min_opcion[1] = 0;
min = sum_mdpp[0][0][0] + sum_mdpp[0][0][1];
/* Recorrer todas las opciones buscando opción mínima              */
for i = 0 ; i <n_opciones ; i++ do
    for j = 0 ; j <n_splits ; j++ do
        /* Descartar valores NaN                                   */
        if std::isnan(sum_mdpp[i][j][0]+sum_mdpp[i][j][1]) then
            continue;
        else if std::isnan(min) then
            min = sum_mdpp[i][j][0]+sum_mdpp[i][j][1];
            min_opcion[0] = i;
            min_opcion[1] = j;
        end
        /* Opción menor que mínimo actual                          */
        if sum_mdpp[i][j][0]+sum_mdpp[i][j][1] <min then
            min = sum_mdpp[i][j][0]+sum_mdpp[i][j][1];
            min_opcion[0] = i;
            min_opcion[1] = j;
        end
    end
end
/* Ignorar componente delay si opción mínima es infinito o NaN     */
if min == std::numeric_limits<double>::infinity() || std::isnan(min) then
    /* Elegir opción más estable                                   */
    min_opcion[0] = 0;
    min_opcion[1] = 0;
    min = sum_mdpp[0][0][0];
    for i = 0 ; i <n_opciones ; i++ do
        for j = 0 ; j <n_splits ; j++ do
            if std::isnan(sum_mdpp[i][j][0]) then
                continue;
            else if std::isnan(min) then
                min = sum_mdpp[i][j][0];
                min_opcion[0] = i;
                min_opcion[1] = j;
            end
            if sum_mdpp[i][j][0] <min then
                min = sum_mdpp[i][j][0];
                min_opcion[0] = i;
                min_opcion[1] = j;
            end
        end
    end
end
```

Algorithm 12: Función *decision()* del *scheduler Min Drift-Plus-Penalty*: Inicialización

```
/* Pasar de la opción mínima a turno y split de CPU            */
opcionToTurno(turno_cpu,min_opcion[0]);
opcionToSplit(split_cpu,min_opcion[1]);
write(decisionfile);
/* Inicializar a 0 tiempo de CPU, tasa y capacidad del RRH     */
for i = 0 ; i <NUMERO_CPUS ; i++ do
    t_cpu[i] = 0;
end
for i = 0 ; i <NUMERO_COLAS ; i++ do
    tasaRRH[i][0] = 0;
    tasaRRH[i][1] = 0;
    for j = 0 ; j <NUMERO_CPUS ; j++ do
        capRRH[i][j] = 0;
    end
end
```

Algorithm 13: Función *decision()* del *scheduler Min Drift-Plus-Penalty*: Procesado

```
/* Procesar paquetes para todas las CPUs                                    */
for n_cpu = 0 ; n_cpu <NUMERO_CPUS ; n_cpu++ do
    /* Elegir cola según indique la opción mínima                           */
    if turno_cpu[n_cpu] == 0) then
        continue;
    end
    cola_actual = turno_cpu[n_cpu] - 1;
    if colas[cola_actual].getNumPaquetes() == 0 then
        continue;
    end
    /* Capacidades equivalentes para el split elegido                       */
    capacidad_CPU = capEqSplitCPU(split_cpu[n_cpu],cpus[n_cpu].getCapacidad());
    capacidad_RRH =
     capEqSplitRRH(split_cpu[n_cpu],cola_actual.getRRH()->getCapacidad());
    capRRH[cola_actual][n_cpu] = capacidad_RRH;
    while cola_actual.getPktPtr()->tiempo <slot && t_cpu[n_cpu] +
     cola_actual.getPktPtr()->size/capacidad_CPU <= TIEMPO_SLOT do
        paquete = cola_actual.cogerPkt();
        /* Añadir tiempo de procesado al tiempo de CPU                      */
        t_cpu[n_cpu] += paquete->size/capacidad_CPU;
        /* Calcular los retardos                                            */
        tiempoCOLA = slot - paquete->tiempo;
        tiempoCPU = paquete->size/capacidad_CPU;
        tiempoPROP = cola_actual.getRRH()->getTpropagacion();
        tiempoRRH = cola_actual.getRRH()->calculaDelay();
        tasaRRH[cola_actual][0] ++;
        tasaRRH[cola_actual][1] += paquete->size;
        /* Escribir fichero de retardos                                     */
        write(delayfile);
        free(paquete);
        if cola_actual.getNumPaquetes() == 0 then
            break;
        end
    end
end
/* Actualizar tasa y capacidad RRH (Algoritmo 3)                            */
/* Escribir fichero de uso de CPU                                           */
for n_cpu = 0 ; n_cpu <NUMERO_CPUS ; n_cpu++ do
    write(CPUfile);
end
```

4. Resultados

A lo largo de esta sección se presentan los resultados obtenidos de las diferentes simulaciones que se han realizado para comprobar el rendimiento del *scheduler* implementado en este trabajo.

Se tienen tres escenarios, el primero, al que se ha denotado como escenario 0, no se plantea para analizar el *scheduler* propiamente, sino para contrastar su funcionamiento frente a alternativas tradicionales, carentes de la capacidad de tomar decisiones de nivel de *split*.

En el segundo escenario analizado, llamado escenario 1, se analiza el *scheduler Min Drift-Plus-Penalty* bajo cuatro distintos supuestos, para evaluar su rendimiento frente a diferentes patrones de tráfico y repartos de capacidad en los RRHs.

El último escenario, conocido como escenario 2, incrementa la complejidad del sistema respecto al anterior, añadiendo una CPU y una cola y ciertas restricciones en el uso de los recursos.

4.1. Métricas

Para realizar la comparación entre los diferentes casos analizados se atiende a cuatro tipos de métrica: el retardo, la estabilidad de las colas, el reparto de los *splits* y el *throughput*.

4.1.1. Retardo

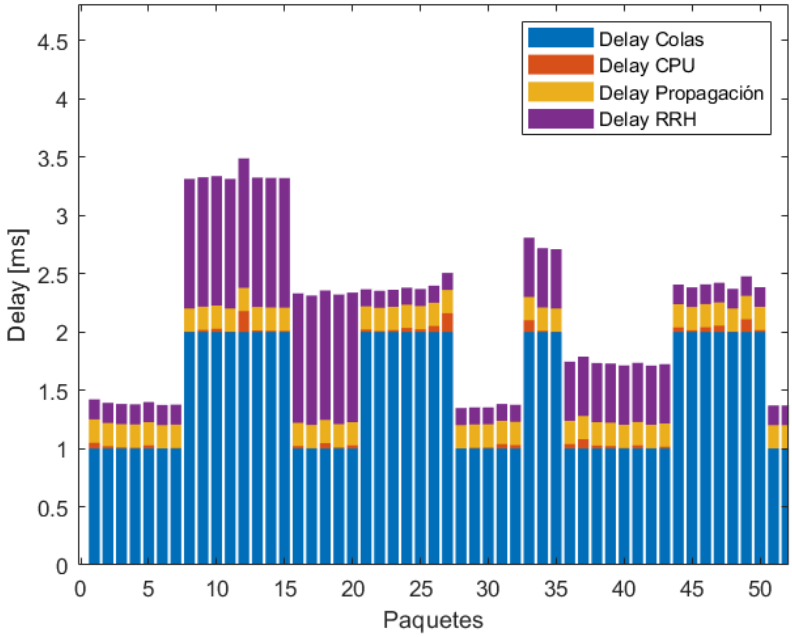


Figura 4.1: Representación del retardo por paquete

La primera de las métricas se trata del retardo, tiene cuatro contribuciones:

- Retardo debido a las colas: Este elemento mide el tiempo de espera del paquete en la cola y se calcula restando el *slot* de entrada del paquete a la cola al *slot* de salida del mismo. Como los paquetes no pueden ser procesados en el mismo *slot* en el que llegan esta componente del retardo será como mínimo de un *slot*.

- Retardo debido al procesado en la CPU: Se trata del tiempo que se tarda en realizar las tareas asociadas al paquete en la BBU. Se obtiene dividiendo el tamaño del paquete entre la capacidad equivalente de la CPU para el nivel de *split* en el que se encuentre trabajando.
- Retardo debido al enlace entre BBU y RRH: Se debe al tiempo que tarda el paquete en ser transmitido entre la *BBU Pool* y los RRHs. Está configurado como un valor fijo de 0.2 ms para todas las simulaciones.
- Retardo debido al RRH: Aparece tanto por el *buffer* de espera en el RRH como por el procesado en el mismo. Se obtiene de la función *calculaDelay()* del RRH, cuyo funcionamiento se ha explicado en la Sección 3.6. Se trata de la componente del retardo más importante ya que si no se controla bien el tráfico que se envía al RRH puede crecer exponencialmente.

El retardo total de un paquete se obtiene, por lo tanto, como la suma de las cuatro componentes del retardo. La Figura 4.1 representa cada una de las componentes con distintos colores para cada paquete.

Los valores que se encuentran representados en las gráficas de los resultados se corresponden con la media de dichas componentes para cada cola a lo largo de toda la simulación.

4.1.2. ESTABILIDAD

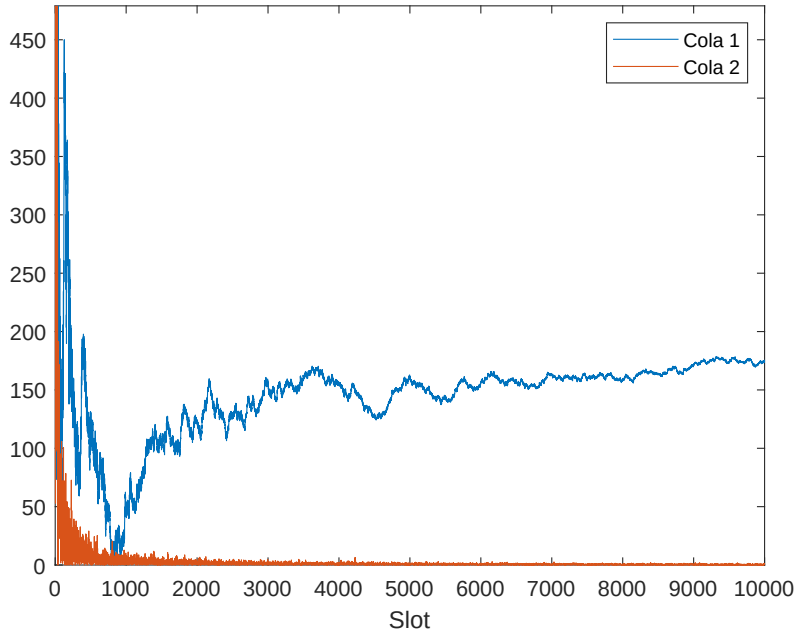


Figura 4.2: Representación de la estabilidad de las colas

La segunda métrica empleada se trata de la estabilidad, conocida como *mean rate stability*. Es un valor que se obtiene para cada cola en todos los *slots* de la simulación con la Ecuación 4.1, donde $a_k(i)$ representa el número de bytes que llegaron a la cola k en el *slot* i y $b_k(i)$ representa el número de bytes que salieron de la cola k en el *slot* i.

Por lo tanto, para obtener el valor del *mean rate* para cierta cola, k, en cierto *slot* se calcula el número de bytes que llegan a la cola k menos los bytes que salen de la misma, y se realiza el sumatorio para cada *slot* desde el primero hasta el *slot* de interés, dividido entre el número de *slots*:

$$estabilidad_k(slot) = \frac{\sum_{i=1}^{slot} a_k(i) - b_k(i)}{slot} \tag{4.1}$$

Para que una cola sea estable el valor del *mean rate* debe tender a 0 si el número de *slots* tiende a infinito.

En la Figura 4.2 se representa la estabilidad de dos colas a lo largo de los 10 000 *slots* de una simulación. Se aprecia que la cola 1 es inestable dado que no tiende a 0, como si ocurre en el caso de la cola 2.

4.1.3. SELECCIÓN DE SPLITS

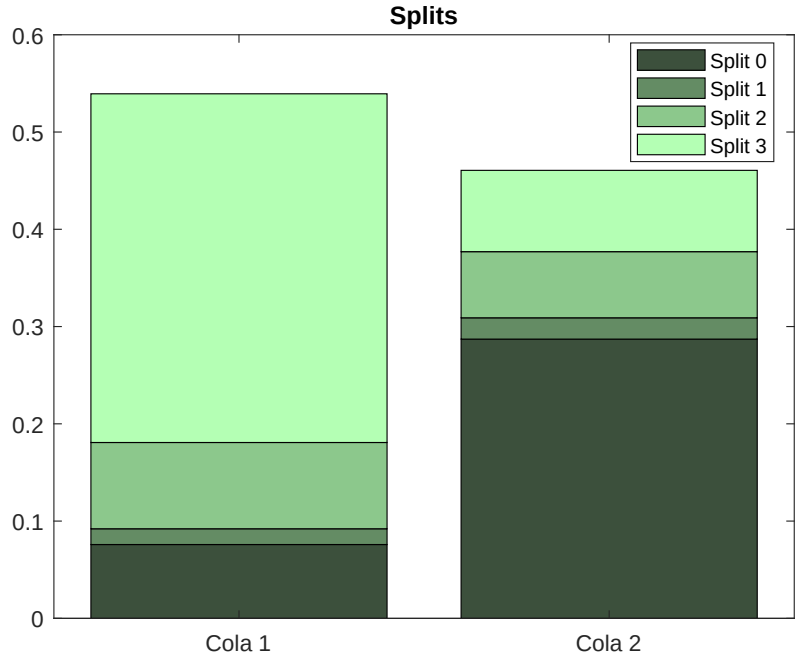


Figura 4.3: Representación del reparto de los niveles de *split*

La tercera de las métricas se trata de la probabilidad con la que se selecciona cada cola y el reparto de cada uno de los niveles de *split* en estas. Al tratarse de una probabilidad la suma de las mismas debe ser 1 para cada simulación.

Se obtiene a partir de la información del fichero de decisiones. Cada vez que una cola y un nivel de *split* son elegidos se incrementa un contador en 1. Una vez se han analizado todas las decisiones tomadas a lo largo de la simulación, se dividen todos los contadores entre el número total de decisiones y se representan los resultados.

En la Figura 4.3 se tiene un ejemplo de esta métrica para una simulación con dos colas, donde se puede comprobar que si se suman las alturas de ambas barras se llega a 1. Los niveles de *split* van desde el 0 al 3 que, recordando lo mencionado en secciones anteriores, se corresponden con el de menor y de mayor centralización, respectivamente. De la figura se puede suponer también que los tráficos de entrada de ambas colas son distintos ya que, como se verá posteriormente, si se tienen entradas similares las decisiones son también similares.

4.1.4. THROUGHPUT

La última de las métricas representadas en los resultados se trata de la tasa de transferencia efectiva, o *throughput*, que se obtiene a partir del número de bytes que han salido de la cola durante la simulación, el número de *slots* y la duración de un *slot* en milisegundos, como se ve en la Ecuación 4.2:

$$Throughput_k[kbps] = \frac{BytesOut_k}{NumSlots \cdot t_{slot}[ms]} \tag{4.2}$$

Como se verá posteriormente, siempre que no se esté trabajando en el punto de saturación del sistema, el *throughput* dependerá directamente de los tráficos de entrada de las colas.

4.2. ELECCIÓN DEL PESO DE LOS RETARDOS MIN DRIFT-PLUS-PENALTY

A fin de encontrar un buen valor para el peso de los retardos se realiza un barrido de los mismos en potencias de 10, desde 10^3 hasta 10^9 con pasos de 10^2 con unos parámetros de simulación similares a los que se contemplan para los escenarios planteados. Se incluyen también los resultados con peso 0, con el objetivo de comparar el caso de atender únicamente a la estabilidad de las colas.

Los resultados obtenidos para cada uno de los pesos simulados se han representado en la Figura 4.4, compuesta por cuatro gráficas. En la Figura 4.4a se muestra el uso de la CPU promediado cada 40 *slots* con los diferentes pesos comparados. Se observa que, a medida que aumenta el peso de la componente del retardo en la decisión del *scheduler*, el uso de la CPU en cada *slot* aumenta.

Esto se explica observando la Figura 4.4d donde se puede apreciar que cuanto mayor es el peso del retardo mayor es la centralización. Esto es, aumenta la probabilidad de los niveles de *split* más altos, reduciendo la capacidad de extraer paquetes de la cola, pero también la carga que tiene que soportar la RRH y por lo tanto el retardo debido a este.

Atendiendo a la Figura 4.4b se comprueba lo mencionado en el párrafo anterior, el retardo debido a la RRH se ve claramente reducido con el aumento del peso de la componente del retardo, a cambio de un ligero incremento en el tiempo de espera en las colas.

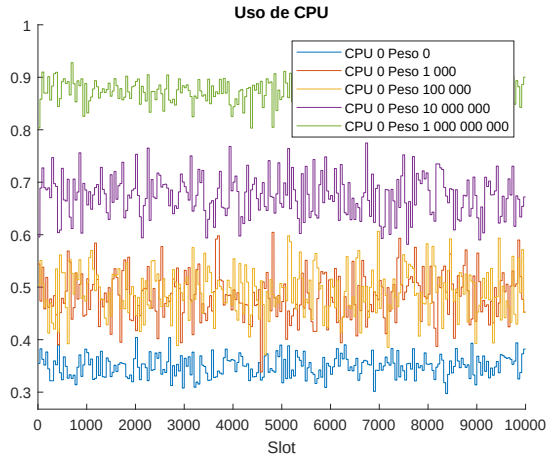
Como muestra la Figura 4.4c, ninguno de los pesos que se han utilizado llega a perjudicar la estabilidad de las colas. Sin embargo, un aumento excesivo del mismo podría llegar a forzar al *scheduler* a emplear niveles de *split* demasiado centralizados como para poder procesar todo el tráfico de entrada del sistema que pasaría a dejar de ser estable, lo que generaría un uso de CPU cercano al 100 %, y en que la estabilidad no estaría garantizada.

El reparto de los niveles de *split* entre las simulaciones, visible en la Figura 4.4d, prueba de nuevo lo visto y mencionado con anterioridad. Un mayor peso lleva a niveles de centralización mayores. Destaca también que en caso de atender únicamente a la estabilidad de las colas se emplea solo el *split* más descentralizado, con el objetivo de vaciar las colas lo más rápido posible, lo que lleva a cargar en exceso la RRH y un aumento significativo del retardo.

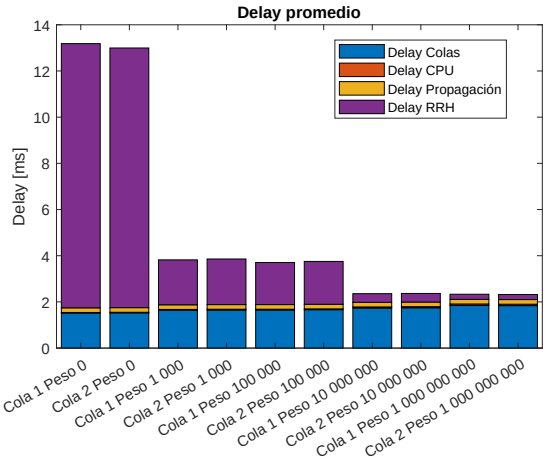
En vista de estos resultados se ha elegido para las simulaciones de los escenarios propuestos el valor de peso 10^7, ya que 10^9 no provee una gran mejora y en caso de un aumento en el tráfico de entrada tendría una menor flexibilidad.

Se toma el valor positivo para el peso del retardo debido a la CPU y a las RRHs y con valor negativo para el peso debido al retardo en las colas. Esto se debe a que se desea que la contribución del retardo de

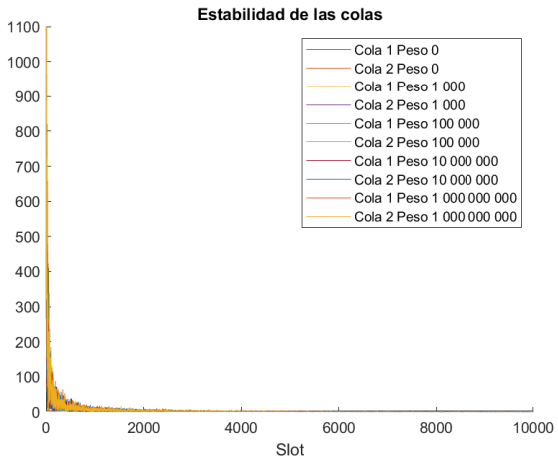
las colas sea positivo para la elección de la configuración, maximizando el número de paquetes que salen o tomando los paquetes que lleven mayor tiempo en las colas. Un valor positivo en el peso del retardo provocará una contribución negativa al coste, por lo que los retardos debidos a la CPU y las RRHs serán minimizados.



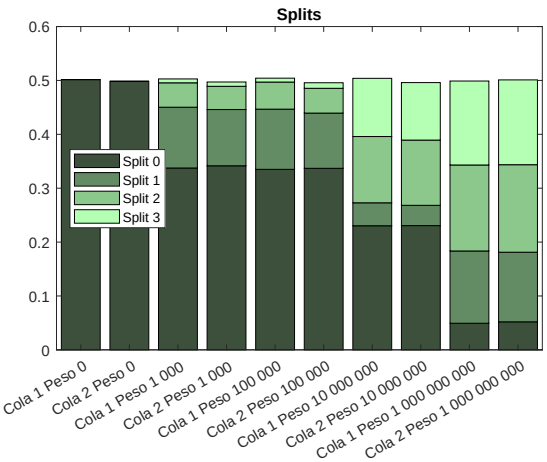
(a) Uso de CPU



(b) Delay



(c) Estabilidad



(d) Split

Figura 4.4: Resultados comparación pesos

4.3. Escenario 0

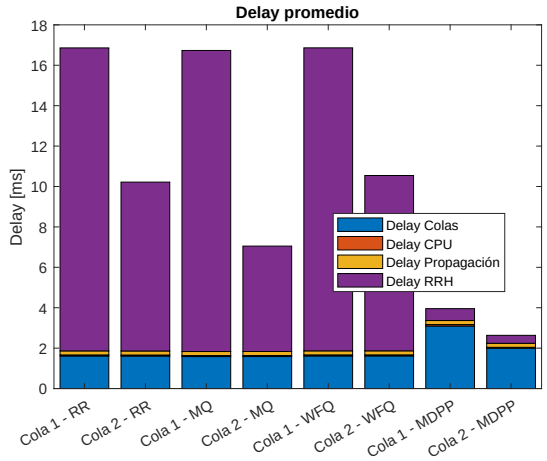
En primer lugar, en el escenario 0, se realiza la comprobación y comparación de los cuatro *schedulers* implementados. Para el reparto de tareas entre la CPU y la RRH se ha decidido tomar 5 funcionalidades, todas ellas con peso de procesado 1, que dejan 4 posibles niveles de *split*, valor que se mantendrá en todas las simulaciones. El escenario consta de una única CPU y dos colas con sus respectivas RRHs. Se simulan los cuatro *schedulers* con los mismos parámetros de entrada, y se confrontan los resultados para ver las diferencias.

- Capacidad nominal CPU: 20 000
- Retardo enlace: 0.2 ms
- Prioridad colas: 1

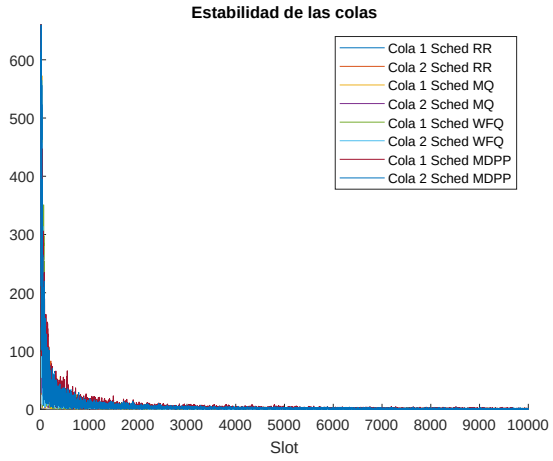
- Tasa colas: 7
- Tamaño medio paquetes colas: 500 bytes
- Nivel de split fijado: 1
- Peso retardos (Min Drift-Plus-Penalty): 10 000 000

Dentro del escenario se analizan dos casos, el primero un caso simétrico donde las dos RRHs tienen la misma capacidad nominal e igual al 60 % de la capacidad de la CPU; el segundo, un caso asimétrico donde la RRH asociada a la cola 1 cuenta con un 30 % de la capacidad nominal de la CPU y la RRH asociada a la cola 2 con un 60 %.

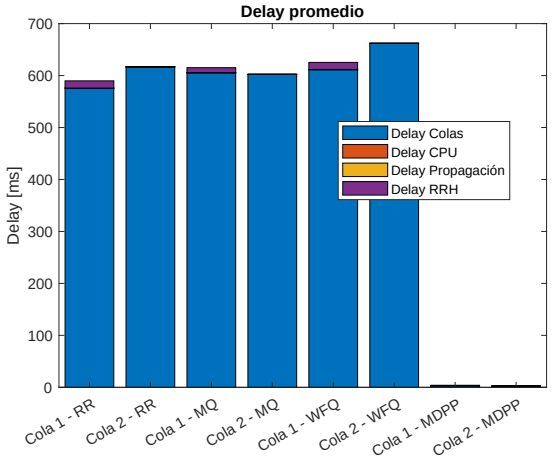
Para los *schedulers* sin capacidad de decisión sobre el nivel de *split* se ha tomado como nivel fijo el *split* 1, el segundo más descentralizado. En la Figura 4.5 se muestran los retardos y estabilidad para los niveles de *split* 1 y 2 en el caso de RRHs asimétricas. Como se puede extraer de las gráficas, el nivel de *split* 1 es la última opción estable y será por lo tanto el empleado para el resto de simulaciones. En el Anexo B se tienen los retardos y estabilidades para todos los niveles de *split* en ambos casos, simétrico y asimétrico.



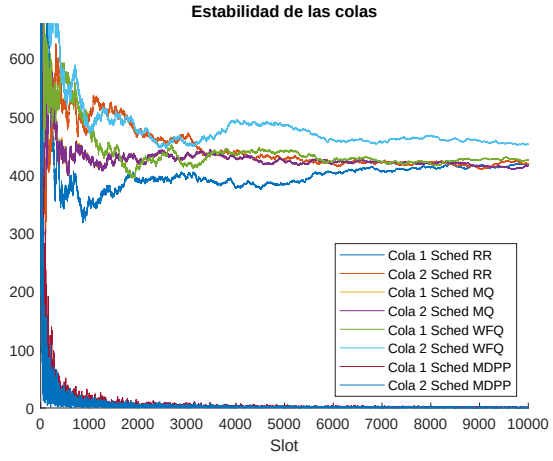
(a) Retardo split 1



(b) Estabilidad split 1



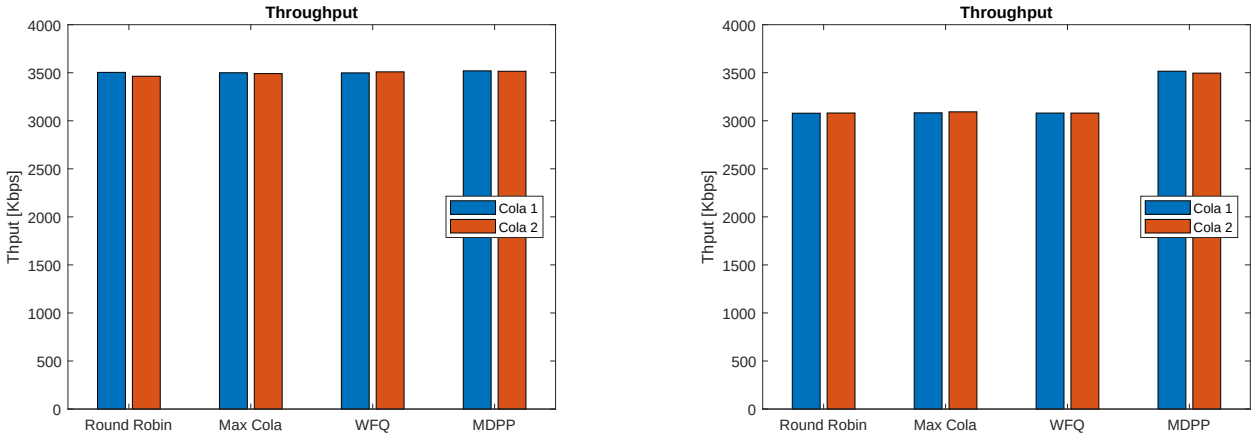
(c) Retardo split 2



(d) Estabilidad split 2

Figura 4.5: Resultados por *split* RRHs asimétricas

Además del claro impacto de la inestabilidad de las colas en el retardo, cabe destacar la reducción que aparece en el *throughput* del sistema, representado en la Figura 4.6, donde se aprecia que, ver Figura 4.6a (el caso estable), el rendimiento obtenido por todos los *schedulers* es muy similar, mientras que en la Figura 4.6b, donde ya aparece la inestabilidad en las colas, el rendimiento del *scheduler Min Drift-Plus-Penalty* es superior a los demás, ya que este consigue mantener la estabilidad en las colas, como se advierte en la Figura 4.5d.



(a) Throughput split 1

(b) Throughput split 2

Figura 4.6: Comparación *throughput* diferentes *splits* RRHs asimétricas

Se han simulado los casos simétrico y asimétrico, y los resultados se han representado en las Figuras 4.7 y 4.8, respectivamente. Dichas figuras se encuentran subdivididas en cuatro gráficas, la primera muestra los valores promedios del retardo para cada cola y cada *scheduler*, donde se han eliminado los retardos infinitos debidos a las RRHs, que se han igualado al mayor valor distinto de infinito; la segunda representa el valor de la estabilidad para cola y *scheduler* en todos los *slots*; la tercera exhibe la selección de nivel de *split* y cola del *scheduler Min Drift-Plus-Penalty*; y la cuarta el *throughput* obtenido para cada cola y *scheduler*.

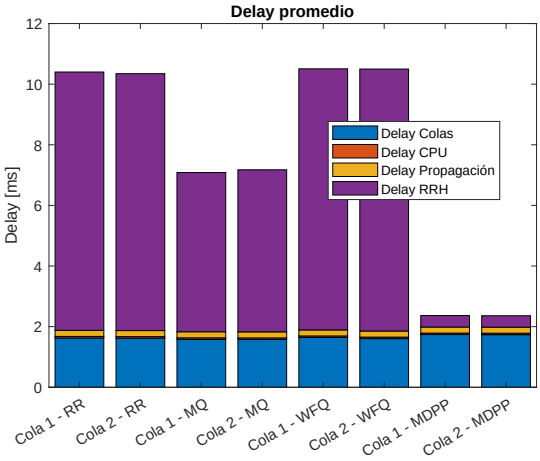
En el primero de los casos, el simétrico, se puede ver la clara ventaja de poder decidir el nivel de *split* de forma dinámica en términos de retardo. En la Figura 4.7a se aprecia que el retardo medio de los paquetes cuando se emplea el *scheduler Min Drift-Plus-Penalty* es mucho menor que cuando se emplea un *scheduler* con políticas de nivel de *split* fijas. Esto se debe principalmente al retardo debido a la RRH, por la configuración elegida. En caso de cambiar el nivel de *split* fijado a uno menos centralizado se podría perjudicar la estabilidad, ya que mermaría la capacidad de la CPU de drenar paquetes de las colas. Por el otro lado, el retardo debido a la RRH sería mucho menor, al tener una disminución en las tareas a procesar en los mismos. En caso de alcanzar la inestabilidad, esta mejora en el retardo debido a la RRH se vería rápidamente contrarrestada con un deterioro mucho mayor por los tiempos de espera en las colas, demostración también de que una elección fija del nivel de *split* realizada de forma indebida puede llevar a un empeoramiento del rendimiento del sistema, como se mostró en las Figuras 4.5 y 4.6.

En la Figura 4.7b se aprecia que las dos colas permanecen estables con los cuatro *schedulers* como ya se comprobó al fijar del nivel de *split* para las simulaciones.

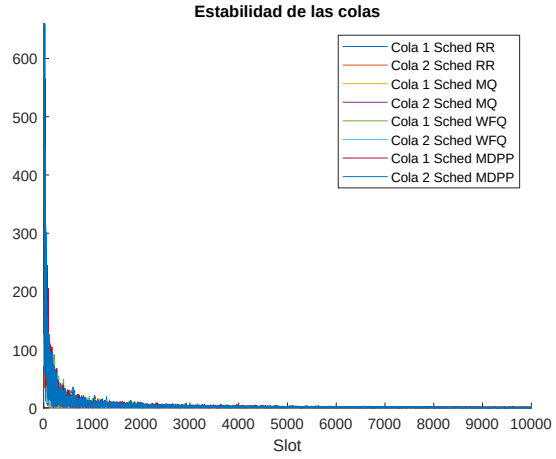
Respecto a la selección del nivel de *split* por el *scheduler Min Drift-Plus-Penalty*, visible en la Figura 4.7c, cabe destacar el empleo de configuraciones bastante descentralizadas. Además, al tratarse del escenario simétrico, no hay diferencias notables en las decisiones tomadas para ambas colas.

El *throughput* del sistema es muy similar para todos los *schedulers* en las dos colas, ya que no se dan

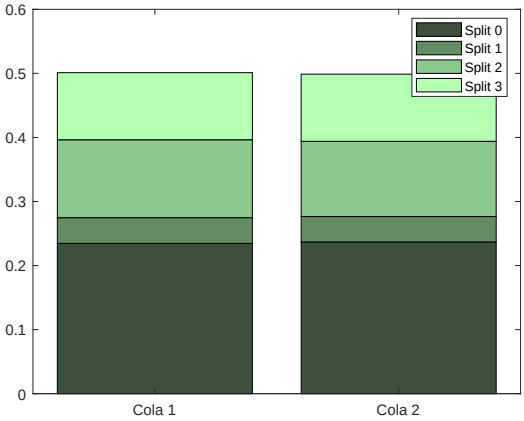
situaciones de inestabilidad a lo largo de las simulaciones, por lo que todos los paquetes que entran al sistema salen en un tiempo reducido.



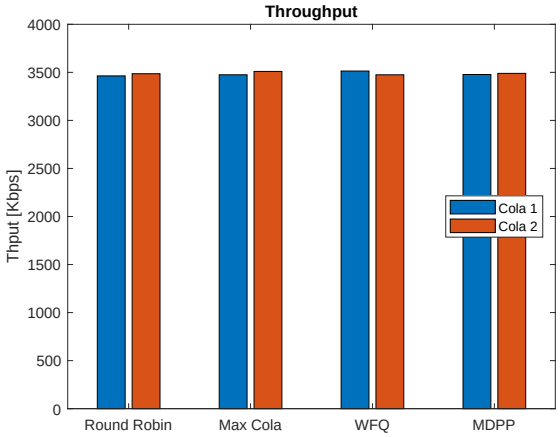
(a) Delay



(b) Estabilidad



(c) Split MDPP



(d) Throughput

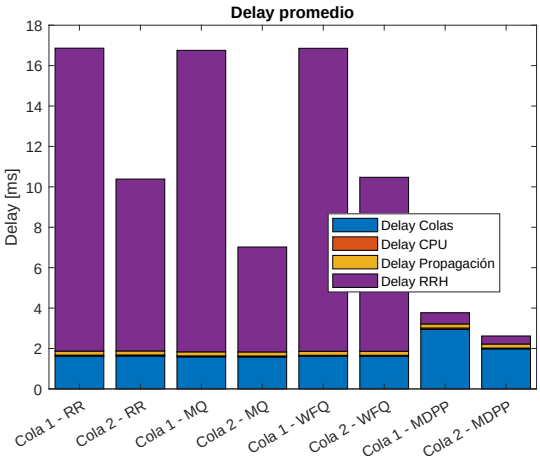
Figura 4.7: Resultados del escenario 0 RRHs simétricas

En el caso asimétrico se observa de forma clara la capacidad del *scheduler Min Drift-Plus-Penalty* para regular el tráfico que debe cursar la RRH, gracias a la selección del nivel de *split* en el que trabaja. Esta cualidad es de especial interés para este escenario, donde una RRH cuenta con una capacidad mucho menor que el otro.

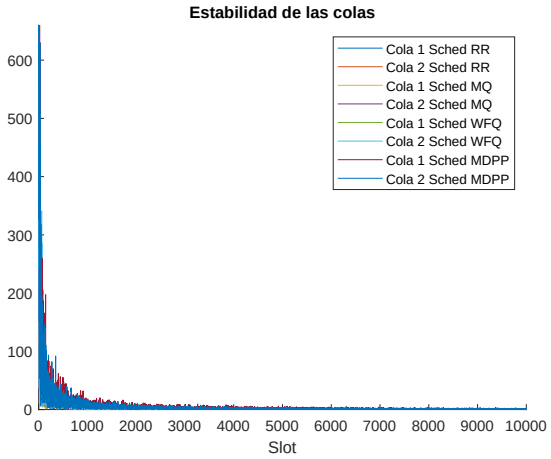
Se puede apreciar que para los tres *schedulers* que son incapaces de decidir el nivel de *split*, el retardo debido a la RRH de menor capacidad es muy superior al debido a la RRH de mayor capacidad. En el *scheduler Min Drift-Plus-Penalty* no se da esta situación y ambos retardos son muy similares. Por el contrario, el retardo debido a la espera en las colas es ligeramente superior en la cola con RRH de capacidad de procesado reducida, y que el *scheduler* toma un nivel de *split* más centralizado para esta cola, reduciendo el trabajo que se debe realizar en la RRH, a cambio de aumentar el que se debe ejecutar en la CPU centralizada. Al encontrarse la CPU con más labores a efectuar es capaz de extraer de la cola un número menor de paquetes, lo que conlleva mayores tiempos de espera.

Esto se hace palpable en la Figura 4.8c, donde se aprecia el nuevo reparto de niveles de *split* para el caso asimétrico. Mientras que la cola 2, con capacidad de procesado en la RRH igual al caso simétrico,

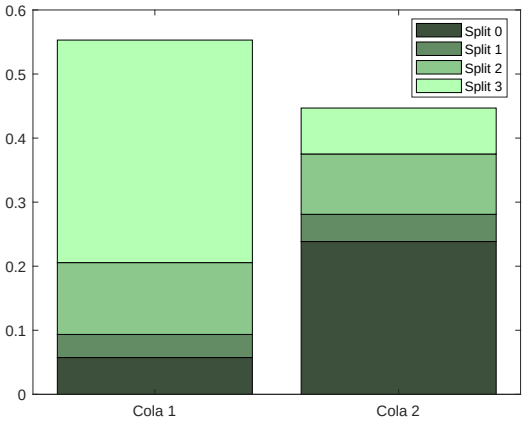
mantiene un reparto similar al anterior, la cola 1, con capacidad reducida, muestra un reparto mucho más centralizado. De esta forma se compensa la carencia de capacidad en la RRH con una mayor carga de tareas en la CPU centralizada.



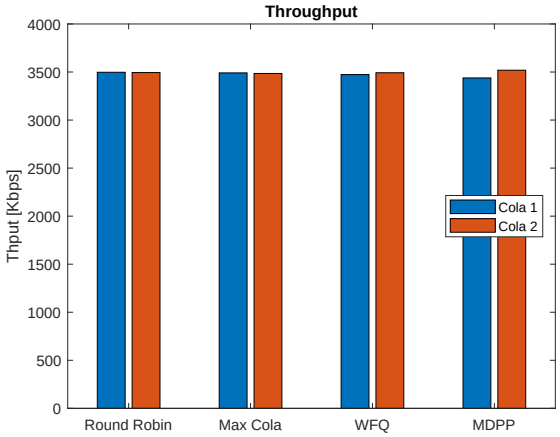
(a) Delay



(b) Estabilidad



(c) Split MDPP



(d) Throughput

Figura 4.8: Resultados del escenario 0 RRHs asimétricas

4.4. ESCENARIO 1

El primero de los escenarios propuestos se compone de una única CPU que da servicio a dos colas con su RRH asociada. En la Figura 4.9 se puede ver un esquemático del sistema configurado para este escenario.

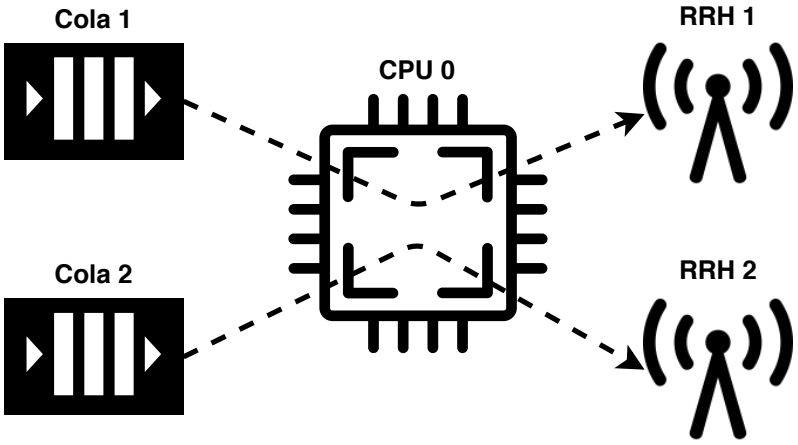


Figura 4.9: Esquema escenario 1

En el escenario hay ciertos parámetros que permanecen invariantes a lo largo de los distintos casos presentados, se encuentran listados a continuación:

- Capacidad nominal CPU: 20 000
- Retardo enlace: 0.2 ms
- Prioridad colas: 1
- Tamaño medio paquetes colas: 500 bytes
- Scheduler: Min Drift-Plus-Penalty
- Peso retardos: 10 000 000

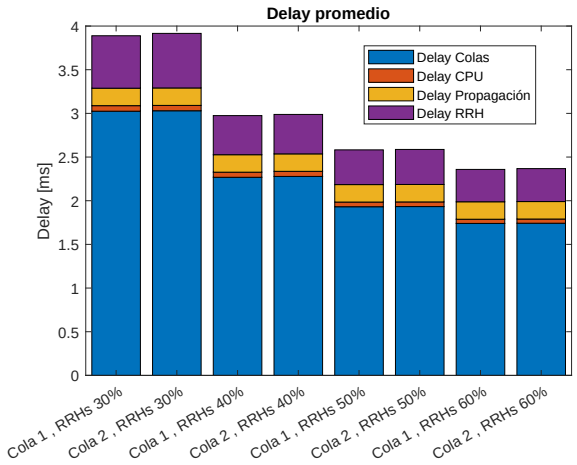
4.4.1. CASO 1: TRÁFICO Y RRHS HOMOGÉNEOS

El primero de los casos planteados para el escenario 1 consiste en variar únicamente la capacidad nominal de las RRHs desde un 30 % de la capacidad de la CPU hasta un 60 % de la misma, siendo igual para ambas RRHs en todas las simulaciones y dejando constantes el resto de los parámetros.

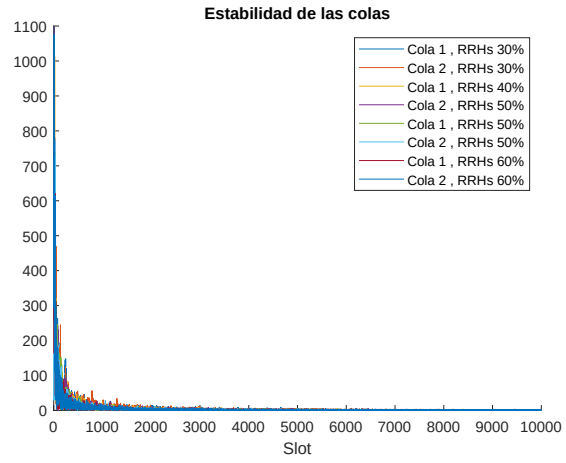
La tasa de entrada de ambas colas es i.i.d. y sigue una distribución de Poisson con tasa, λ, de valor 7 paquetes por *slot*. Se tienen un total de cuatro simulaciones donde la capacidad de las RRHs toma los siguientes valores:

- Capacidad nominal RRHs (30 %): 6 000
- Capacidad nominal RRHs (40 %): 8 000
- Capacidad nominal RRHs (50 %): 10 000
- Capacidad nominal RRHs (60 %): 12 000

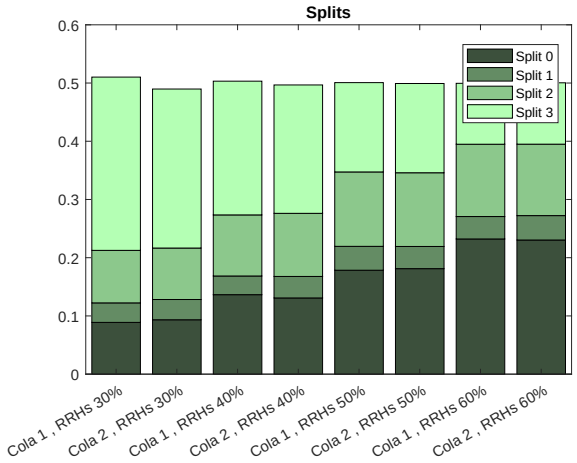
Los resultados de las cuatro simulaciones se muestran en la Figura 4.10, que se encuentra dividida en cuatro gráficas, una para cada una de las métricas mencionadas con anterioridad.



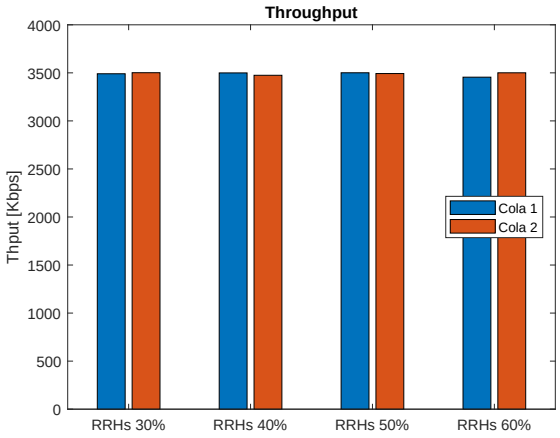
(a) Delay



(b) Estabilidad



(c) Split



(d) Throughput

Figura 4.10: Resultados del escenario 1 caso 1

La Figura 4.10a representa mediante barras los retardos para cada cola y cada simulación. Cada barra se subdivide en las cuatro componentes del retardo. De la figura se puede extraer que a medida que aumenta la capacidad de las RRHs se pueden pasar más funciones a estos y, por lo tanto, la capacidad equivalente de la CPU se ve incrementada al tener un número menor de tareas, reduciendo el tiempo de procesado. Esto implica que la CPU es capaz de drenar un mayor número de paquetes de las colas, disminuyendo la longitud media de estas y el retardo debido a la espera de los paquetes. Una mayor capacidad en las RRHs conlleva también una reducción del retardo debido al tránsito de los paquetes por ellos.

En la Figura 4.10b se muestra la estabilidad de las dos colas para cada una de las simulaciones. Se aprecia que en todos los casos analizados las colas presentan un comportamiento estable, pues su valor medio tiende a 0. Esto se debe a que las tasas de entrada no son lo suficientemente altas para saturar a la CPU.

La Figura 4.10c muestra el reparto de decisiones en las diferentes colas y en las distintas simulaciones. Como se podía esperar, al tener tráfico simétrico en ambas colas, las decisiones son similares dentro de la misma simulación. Se observa que a medida que incrementa la capacidad en las RRHs crece también la probabilidad de seleccionar niveles de *split* más bajos, esto se traduce en llevar más tareas a la RRH y menos a la CPU, reduciendo la centralización.

Por último se tiene la Figura 4.10d, que muestra el *throughput* para ambas colas en las cuatro simulaciones. El *throughput* es casi idéntico para todas ellas, ya que el tráfico de entrada es muy similar y el sistema no se encuentra saturado en ningún punto de las simulaciones, como se comprobó al analizar la estabilidad.

4.4.2. Caso 2: Tráfico homogéneo y RRHs heterogéneas

En el segundo caso se analiza el comportamiento con RRHs heterogéneas. Para ello se toman los mismos valores entre el 30 % y el 60 % de la capacidad de la CPU al igual que en el caso anterior, pero ahora se realizan todas las combinaciones posibles con capacidades diferentes en las RRHs. Se tienen seis simulaciones:

- Capacidad nominal RRH 1 (30 %) - RRH 2 (40 %): 6 000 - 8 000
- Capacidad nominal RRH 1 (30 %) - RRH 2 (50 %): 6 000 - 10 000
- Capacidad nominal RRH 1 (30 %) - RRH 2 (60 %): 6 000 - 12 000
- Capacidad nominal RRH 1 (40 %) - RRH 2 (50 %): 8 000 - 10 000
- Capacidad nominal RRH 1 (40 %) - RRH 2 (60 %): 8 000 - 12 000
- Capacidad nominal RRH 1 (50 %) - RRH 2 (60 %): 10 000 - 12 000

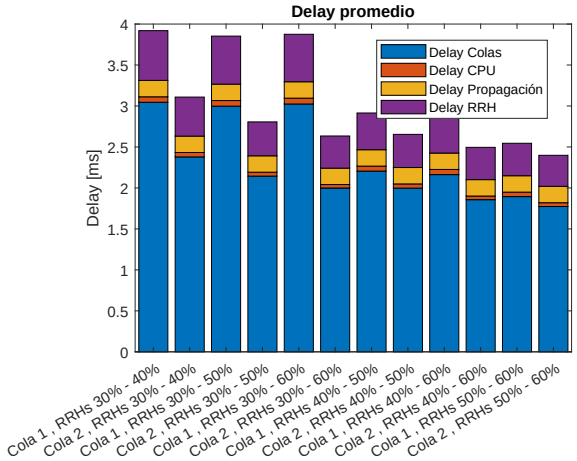
En la Figura 4.11 se muestran los resultados de las simulaciones, nuevamente divididos según las cuatro métricas comparadas.

Los valores del retardo se representan en la Figura 4.11a. Al tener distintas capacidades los valores son distintos para las dos colas dentro de la misma simulación. A medida que aumenta la diferencia entre las capacidades de las RRHs, la desigualdad se vuelve más palpable, como en el caso de la simulación donde la RRH 1 tiene el 30 % de la capacidad de la CPU y la RRH 2 el 60 %, la más dispar de todas. Se aprecia que cuanto mayores son las capacidades menores son los retardos.

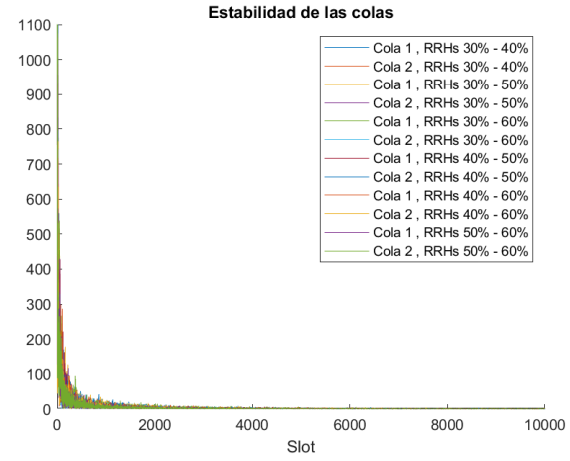
Tal y como se ve en la Figura 4.11b las colas son estables para todos los casos analizados, incluida la simulación en la que la RRH 1 tiene el 30 % de la capacidad de la CPU y la RRH 2 el 40 %, la que menor capacidad presenta en las RRHs.

El reparto de los *splits* se puede ver en la Figura 4.11c y permite comprender mejor el comportamiento del retardo. Las simulaciones con RRHs de menor capacidad tienden hacia niveles de *split* más altos, es decir, mayor centralización, a fin de reducir el retardo ligado a la RRH. Es de interés resaltar que en los casos donde se mantiene la capacidad de la RRH 1 constante y se incrementa la capacidad de la RRH 2 se ve que la cola 2, correspondiente a la RRH con mayor capacidad, accede a la CPU en un menor número de ocasiones que la cola 1, a medida que la capacidad de la RRH 2 crece. Esto se debe a que al tener la RRH 2 mayor capacidad posibilita niveles de *split* más bajos, lo que permite a su vez que la CPU pueda sacar un mayor número de paquetes de la cola por *slot* y hace necesario un menor número de accesos para mantener la estabilidad de la cola.

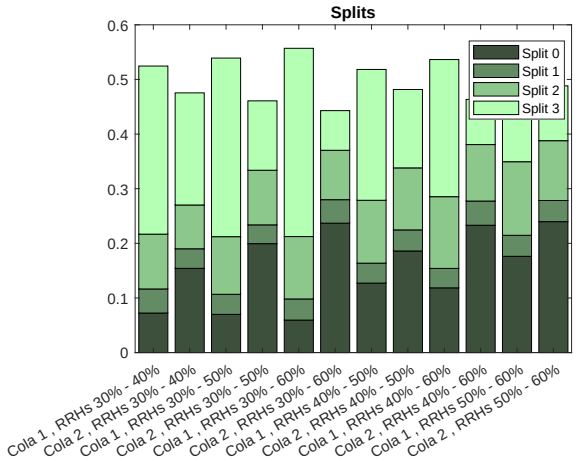
En la Figura 4.11d se muestran los valores del *throughput* para las distintas simulaciones. Nuevamente se observa que son prácticamente iguales, debido a que las tasas de entrada son las mismas para ambas colas en todas las simulaciones y a que el sistema no se ha visto saturado en ninguna configuración.



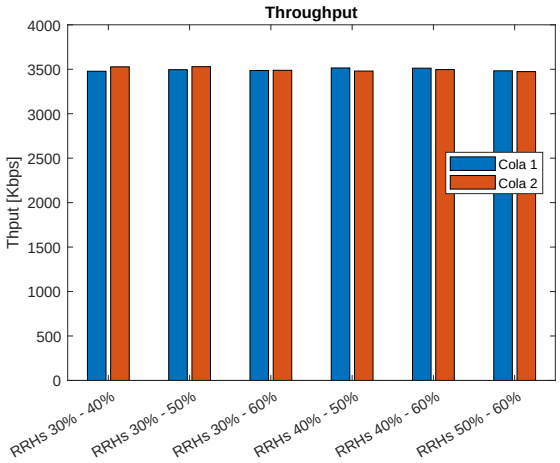
(a) Delay



(b) Estabilidad



(c) Split



(d) Throughput

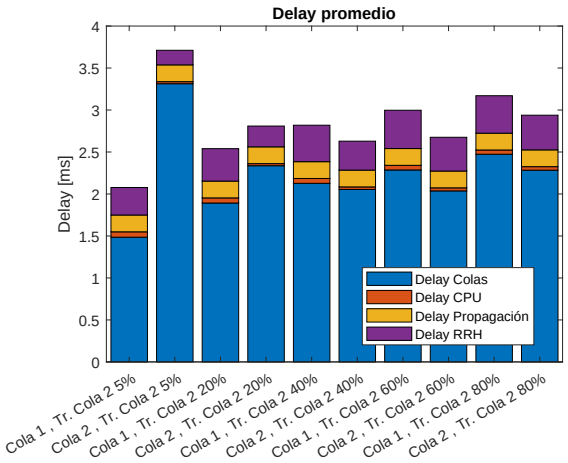
Figura 4.11: Resultados del escenario 1 caso 2

4.4.3. Caso 3: Tráfico heterogéneo

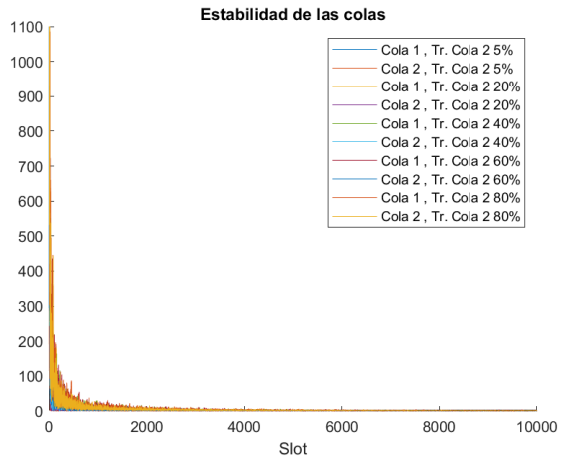
El tercer caso analizado consiste en mantener constantes e iguales las capacidades de las RRHs que se han fijado en el 60 % de la capacidad de la CPU, 12 000, y variar el tráfico que llega a una de las colas, la segunda, manteniendo el tráfico de la primera constante.

Se trabaja directamente sobre la tasa de entrada de las colas. Para este caso se ha ha tomado como tasa de entrada de la cola 1 10 paquetes por *slot*. La tasa de la cola 2 se va modificando con valores 5 %, 20 %, 40 %, 60 % y 80 % de la capacidad de la cola 1, lo que genera cinco situaciones con unos valores finales de:

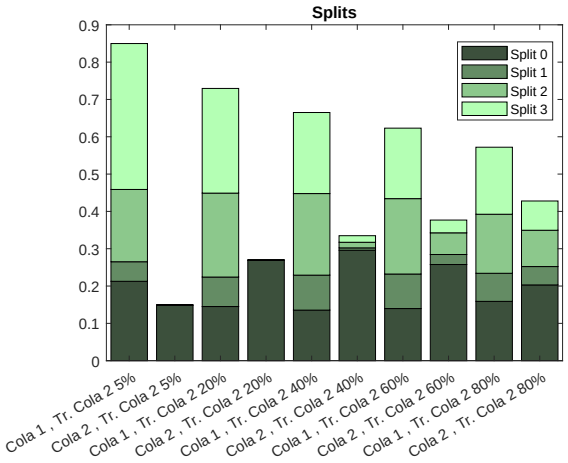
- Tasa cola 1 (100 %) - cola 2 (5 %): 10 - 0.5
- Tasa cola 1 (100 %) - cola 2 (20 %): 10 - 2
- Tasa cola 1 (100 %) - cola 2 (40 %): 10 - 4
- Tasa cola 1 (100 %) - cola 2 (60 %): 10 - 6
- Tasa cola 1 (100 %) - cola 2 (80 %): 10 - 8



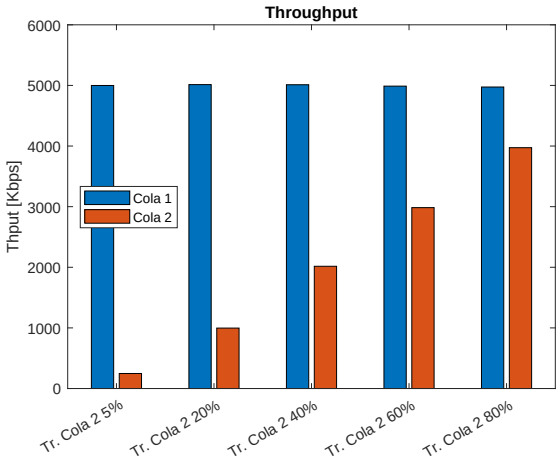
(a) Delay



(b) Estabilidad



(c) Split



(d) Throughput

Figura 4.12: Resultados del escenario 1 caso 3

Los resultados del retardo, en la Figura 4.12a, muestran algo contrario a lo que cabría esperar: el retardo de la cola con menor tráfico es mayor que el de la cola con mayor tráfico cuando se tienen valores del tráfico muy pequeños en aquella. La explicación es que, pese a que el retardo debido a la CPU y a la RRH de la cola con menor tráfico es menor que los respectivos de la cola con tráfico mayor, la espera de los paquetes en la cola es mucho mayor en el caso de la primera. La razón es que al tener una tasa de entrada muy reducida se tardan varios *slots* en juntar un número de paquetes necesarios para no infrautilizar demasiado la CPU. A medida que el tráfico en la segunda cola aumenta se llega a la situación esperada, donde esta cola tiene un menor retardo que la cola 1, de mayor tráfico, debido principalmente a la diferencia en los tiempos de espera en las colas.

Atendiendo a la Figura 4.12b, se puede afirmar que las colas se mantienen estables para todas las simulaciones, siendo el peor caso para la simulación con tasas de 10 y 8 para las colas 1 y 2, respectivamente.

La Figura 4.12c muestra las decisiones de *split* para las distintas simulaciones de este caso. Se puede ver que cuanto mayor es la diferencia en el tráfico de entrada también lo es la probabilidad de que las colas accedan a la CPU. Para tráficos muy bajos se emplea únicamente el *split* 0, de mínima centralización, ya que se desea vaciar la cola cuanto antes y la RRH tiene poca carga por lo que puede permitirse ocupar un mayor número de funciones. Con el incremento del tráfico se aumenta también la carga de la

RRH por lo que comienzan a verse niveles de *split* más centralizados.

Como se ha comentado anteriormente, el *throughput* depende directamente del tráfico de entrada, como se evidencia en la Figura 4.12d. La cola 1 muestra un valor constante de 5000 kbps, ya que no se alcanza la saturación del sistema en ningún momento y siempre se logra servir a las dos colas. La cola 2, sin embargo presenta valores de aproximadamente 250 kbps, un 5 % de 5000 kbps; 1000 kbps, un 20 % de 5000 kbps; 2000 kbps, un 40 % de 5000 kbps; 3000 kbps, un 60 % de 5000 kbps y 4000 kbps, un 80 % de 5000 kbps; valores que coinciden con los esperados de forma teórica.

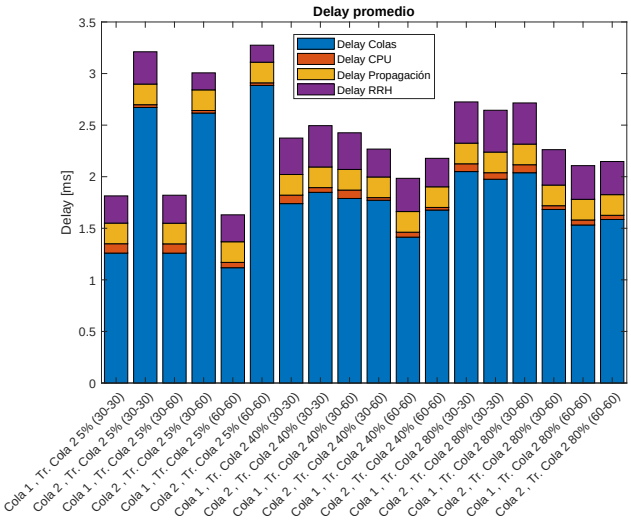
4.4.4. Caso 4: Tráfico y RRHs heterogéneos

El cuarto caso analizado consiste en realizar todas las combinaciones posibles de los anteriores, es decir, variar el tráfico de la cola 2, manteniendo constante en valor 10 el de la cola 1. Se toman valores relativos respecto a este del 5 %, 20 %, 40 %, 60 % y 80 % y para cada uno de esos valores del tráfico se realizan las combinaciones de capacidad de procesado de las RRHs de 30 %-30 %, 30 %-40 %, 30 %-50 %, 30 %-60 %, 40 %-40 %, 40 %-50 %, 40 %-60 %, 50 %-50 %, 50 %-60 % y 60 %-60 %; lo que da un total de 50 simulaciones.

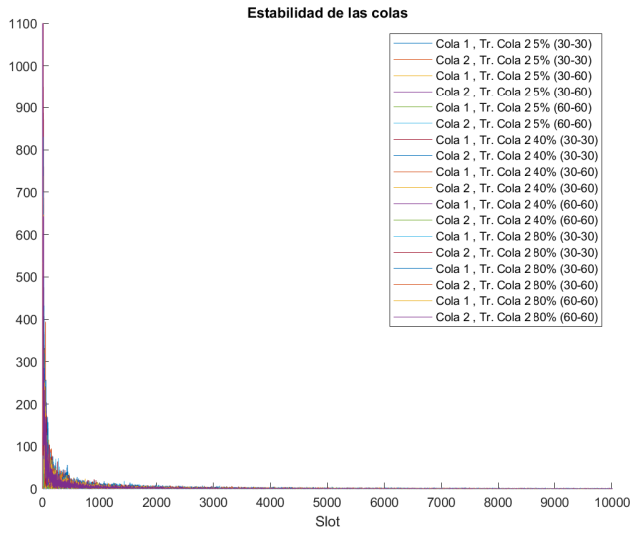
A fin de hacer más comprensibles las gráficas se ha reducido el número de elementos en las mismas mediante una selección. Para ver las representaciones con todos los elementos se puede acudir al Anexo C. Las simulaciones elegidas para ser mostradas en las siguientes figuras son el caso de menor capacidad en las RRHs, con un 30 % de la capacidad de la CPU cada uno; el caso con mayor diferencia en la capacidad de las RRHs, con un 30 % la RRH asociada a la cola 1 y un 60 % la asociada a la cola 2; y el caso con mayor capacidad en las RRHs, con un 60 % ambas. Se ha tomado de cada caso las simulaciones con valores del tráfico de la cola 2 del 5 %, 40 % y 80 % del tráfico de la cola 1, lo que hace un total de 9 combinaciones, cuyos resultados se han representado en la Figura 4.13.

- Tasa cola 2 (5 %): 0.5 ; Capacidad RRH 1 (30 %) - RRH 2 (30 %): 6 000 - 6 000
- Tasa cola 2 (5 %): 0.5 ; Capacidad RRH 1 (30 %) - RRH 2 (60 %): 6 000 - 12 000
- Tasa cola 2 (5 %): 0.5 ; Capacidad RRH 1 (60 %) - RRH 2 (60 %): 12 000 - 12 000
- Tasa cola 2 (40 %): 4 ; Capacidad RRH 1 (30 %) - RRH 2 (30 %): 6 000 - 6 000
- Tasa cola 2 (40 %): 4 ; Capacidad RRH 1 (30 %) - RRH 2 (60 %): 6 000 - 12 000
- Tasa cola 2 (40 %): 4 ; Capacidad RRH 1 (60 %) - RRH 2 (60 %): 12 000 - 12 000
- Tasa cola 2 (80 %): 8 ; Capacidad RRH 1 (30 %) - RRH 2 (30 %): 6 000 - 6 000
- Tasa cola 2 (80 %): 8 ; Capacidad RRH 1 (30 %) - RRH 2 (60 %): 6 000 - 12 000
- Tasa cola 2 (80 %): 8 ; Capacidad RRH 1 (60 %) - RRH 2 (60 %): 12 000 - 12 000

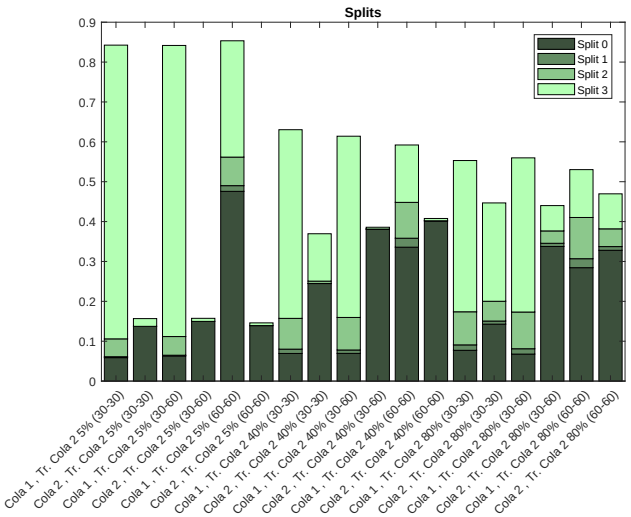
Los retardos promedio obtenidos de las simulaciones se han reproducido en la Figura 4.13a. Lo primero a destacar de la misma es que, de forma un tanto contra intuitiva y como ya se vio en el caso 3, la cola con tráfico muy reducido presenta un gran retardo; esto es debido al tiempo de espera mientras se acumulan un número de paquetes suficientes para ser procesados de forma eficiente. Por el mismo motivo se ve que el caso con mayor diferencia en el tráfico de entrada es también el que mayor diferencia presenta en el retardo de cada cola. Se puede observar también que el incremento en el tráfico no se ve reflejado en un gran aumento del retardo, ya que a medida que el tráfico aumenta la centralización lo hace también, manteniendo el retardo debido a la RRH bajo, a costa de un ligero aumento de los tiempos de espera en las colas. Incluso en el caso de mayor diferencia de capacidad en las RRHs y mayor tráfico, no se logra una gran diferencia en el retardo de ambas colas, gracias a la selección de los niveles de *split* adecuados.



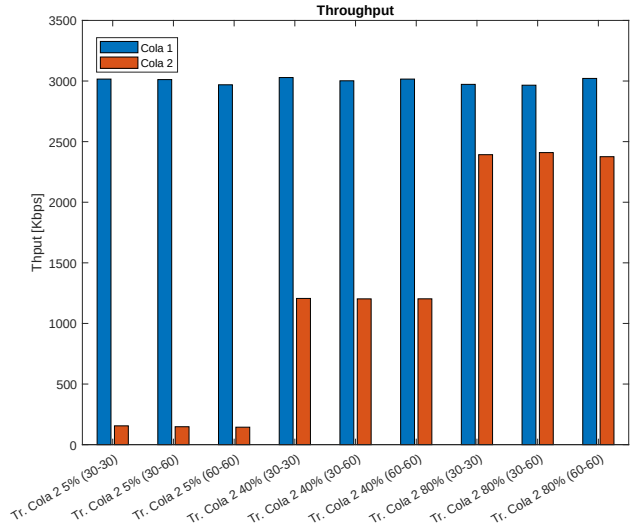
(a) Delay



(b) Estabilidad



(c) Split



(d) Throughput

Figura 4.13: Resultados del escenario 1 caso 4

En la Figura 4.13b se puede comprobar que ambas colas han permanecido estables para todas las simulaciones, lo cual encaja con los retardos debidos a la espera en ellas, que se mantienen en valores reducidos.

Las decisiones del *scheduler* en cuanto a niveles de *split* y concesión de acceso a la CPU a las colas se han representado en la Figura 4.13c. De la misma cabe señalar que la probabilidad de dar acceso al procesador a una cola u otra depende del tráfico relativo entre ambas, y el reparto de niveles de *split* de la capacidad de la RRH asociado a la cola. Se aprecia como, a medida que el tráfico de la cola 2 aumenta y se acerca al de la cola 1, la probabilidad de elegir ambas colas se acerca al 50 %. De un modo similar, destaca que una mayor capacidad en la RRH se refleja en una mayor descentralización, es decir, niveles de *split* menores.

Por último, el *throughput* de cada simulación separado por colas se muestra en la Figura 4.13d. Como cabe esperar a la vista de que el sistema se mantuvo estable en todas las simulaciones, el rendimiento tiene los valores previsibles, manteniéndose constante entre simulaciones con el mismo tráfico de entrada, y aumentando el *throughput* de la cola 2 a medida que el tráfico de entrada lo hace.

4.5. ESCENARIO 2

El segundo escenario que se plantea está formado por tres colas, dos CPUs y tres RRHs. La CPU 0 da servicio a las colas 1 y 2 y la CPU 1 da servicio a las colas 2 y 3, de modo que la cola 2 puede ser procesada por ambas CPUs mientras que las colas 1 y 3 solo pueden ser procesadas por una de las CPUs. En la Figura 4.14 se muestra un diagrama del escenario.

Se han configurado ambas CPUs con una misma capacidad nominal e igual a 20 000. Las tres RRHs tienen capacidad 10 000, un 50 % de la capacidad de la CPU. El retardo debido a los enlaces es de 0.2 ms, las tasas de entrada a las colas tienen un valor de 11 y el tamaño medio del paquete es de 500 bytes para todas ellas. Se ha usado un valor del peso de los retardos de 10 000 000, igual al escenario anterior, tomando valor negativo en el caso del retardo debido a las colas.

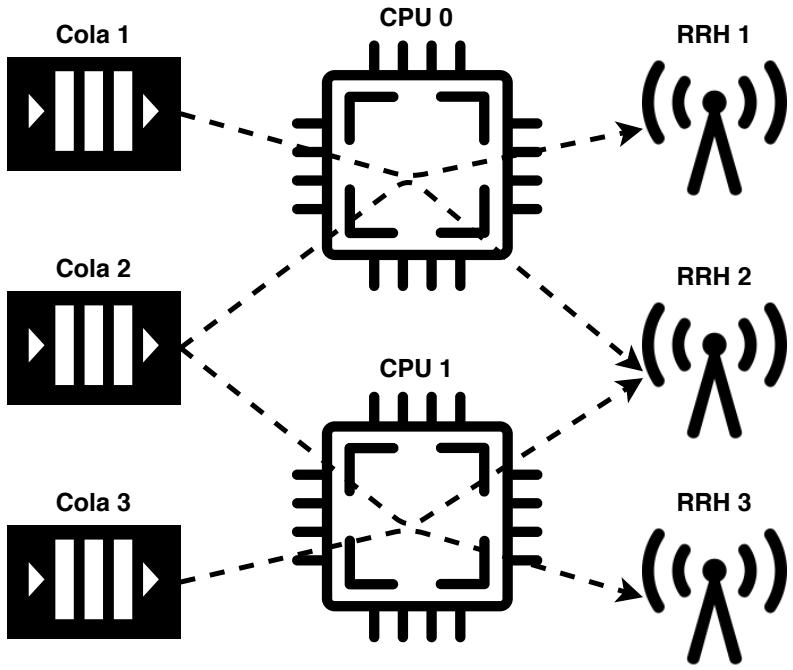


Figura 4.14: Esquema escenario 2

La Figura 4.15a muestra los resultados del retardo en este escenario. Se ve que bajo las mismas condiciones de entrada y capacidad de la RRH, la cola que tiene acceso a ambas CPUs presenta un menor retardo promedio, principalmente debido a la contribución de un menor tiempo de espera en la cola, ya que las demás contribuciones son muy similares para las tres colas.

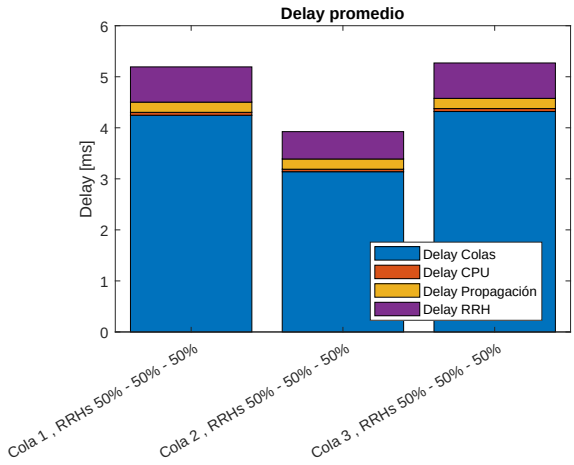
El motivo detrás de este comportamiento es que la cola 2, al tener acceso a ambas CPUs, puede disfrutar de una mayor capacidad de procesado, pudiendo llegar a emplear ambas CPUs simultáneamente, lo que permite servir un mayor número de paquetes de la cola en un mismo *slot*, reduciendo su longitud.

De la Figura 4.15b se destaca que las tres colas permanecen con valores estables en todos los *slots*, es decir, el sistema trabaja fuera de la zona de saturación durante toda la simulación.

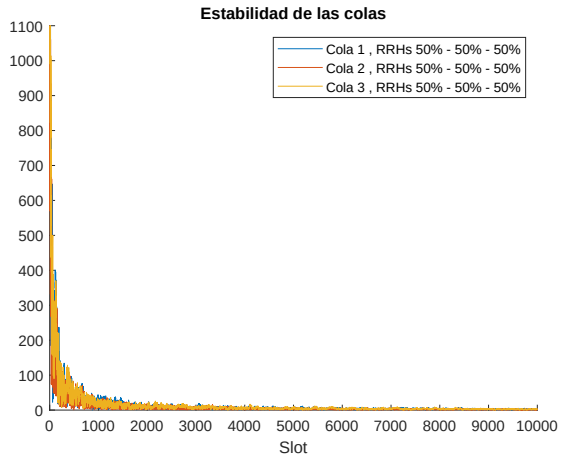
Observando la gráfica de las decisiones de nivel de *split* para cada cola, Figura 4.15c, se aprecia que la cola 2 accede en un menor número de ocasiones a una CPU que las colas 1 y 3, pero cuando lo hace suele ser con un nivel de *split* más bajo, que permite drenar un mayor número de paquetes de la cola. El reparto de niveles de *split* para las colas 1 y 3 es muy similar, ya que cuentan con las mismas características.

Los resultados del *throughput* se han representado en la Figura 4.15d. Al tener las mismas características de tráfico de entrada en las tres colas, y no encontrarse el sistema trabajando en saturación, se tiene

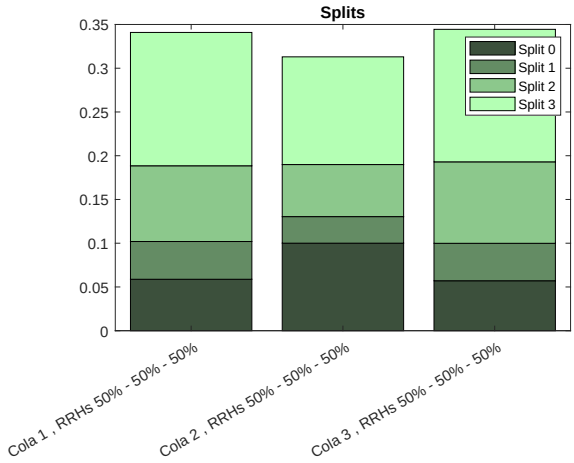
la misma tasa de salida para todas ellas.



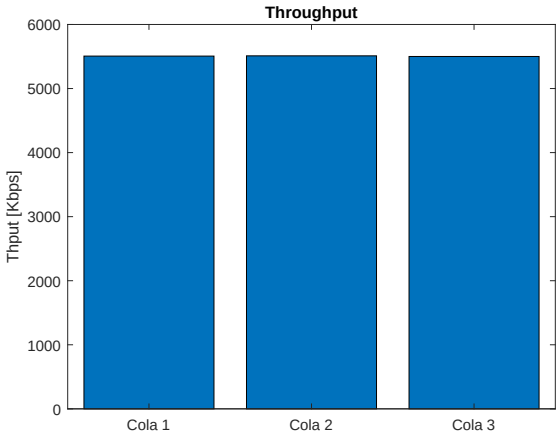
(a) Delay



(b) Estabilidad



(c) Split



(d) Throughput

Figura 4.15: Resultados del escenario 2

5. CONCLUSIONES

A lo largo de este trabajo se ha realizado una introducción a los conceptos de arquitectura C-RAN y división funcional flexible. Además, se ha detallado el desarrollo de un *scheduler* capaz de tomar decisiones de asignación de recursos de procesado a diferentes colas y decisiones de nivel de *split* con el objetivo de reducir el retardo de los paquetes que atraviesan el sistema.

El *scheduler* implementado se ha basado en la optimización de Lyapunov, una herramienta que permite estabilizar colas reales o virtuales a la par que optimizar un objetivo de rendimiento, en este caso, el retardo. La técnica consiste en usar una función de Lyapunov para obtener un valor escalar que represente el estado del sistema. Dicho valor tenderá a crecer a medida que el sistema avanza hacia estados no deseados, por lo que se tomarán las acciones necesarias para reducir ese valor al mínimo posible. Además, se ha añadido un término de penalización, el retardo, de forma que las decisiones tomadas optimizan de forma conjunta la estabilidad y minimizan este parámetro de rendimiento. Otra ventaja de esta técnica es que su implementación es simple y eficiente, y no requiere conocimiento estadístico del sistema que optimiza.

Se ha comparado este *scheduler* frente a otros más tradicionales, incapaces de explotar las ventajas de aplicar *flexible functional splitting* a sus decisiones, resaltando sus diferencias. Por último, se ha analizado el rendimiento del *scheduler* desarrollado bajo diferentes escenarios y se han examinado los resultados obtenidos comprobando su flexibilidad gracias a sus capacidades de balanceo de carga y de adaptación a los diferentes patrones de tráfico, mediante los cambios de *split* dinámicos.

Algunas líneas de trabajo futuro que quedan abiertas son la toma en consideración de colas dinámicas en las RRHs, similares a las empleadas en este trabajo para las CPUs; la introducción de políticas de descarte de paquetes, por ejemplo, estableciendo límites al tamaño de las colas; la elaboración de un método para elegir los pesos óptimos para los retardos en base a la carga de la red, siendo una opción el uso de inteligencia artificial; o el desarrollo de una implementación práctica en laboratorio, utilizando plataformas SDR.

Anexos

A. Fichero de inicialización

En este Anexo se detalla la estructura de el fichero de configuración del sistema que se emplea en las simulaciones. Se ha organizado por filas, cada una con la información indicada a continuación.

En la primera fila se tiene el número entero de colas, el número de CPUs y el número de RRHs.

Tras ello se tiene la capacidad de procesado de cada una de las CPUs, cada valor en una fila y en orden descendente.

Se tiene después el valor fijo de retardo de los enlaces a las RRHs, el orden se utilizará para identificar después cada enlace con su RRH, por lo que habrá el mismo número de enlaces que RRHs.

Las siguientes filas llevan los parámetros de las RRHs: la capacidad y su enlace asociado.

Las colas por su parte llevan cinco parámetros asociados: la longitud máxima de la cola, pudiendo emplearse -1 para crear una cola infinita; la tasa de paquetes que llegan a la cola, tanto si es un número fijo como el parámetro de la distribución de paquetes uniforme o de Poisson; la prioridad, para su uso en el *scheduler WFQ*; el tamaño de los paquetes, tanto si es un número fijo como el parámetro para la distribución de tamaño exponencial negativa; y la RRH asociada a la cola.

Por último se tiene el *scheduler* que se va a emplear en la simulación, donde se contemplan cuatro opciones:

- Round Robin: 0
- Max Cola: 1
- Weighted Fair Queuing: 2
- Min Drift-Plus-Penalty: 3

En caso de tener el *scheduler Min Drift-Plus-Penalty*, aparecen tres filas de información extra: los pesos para cada una de las componentes del retardo. La primera de ellas indica el peso del retardo debido a la espera en las colas, la segunda el peso del retardo debido al procesado en la CPU y la tercera el peso del retardo debido a el RRH y la transmisión por el enlace.

```
# Num Colas , Num CPUs , Num RRH
2 4 2

# CPUs: Capacidad
20000
15000
10000
10000

# Enlaces: Retardo
0.1
0.2

# RRHs: Capacidad , enlace
15000 0
```

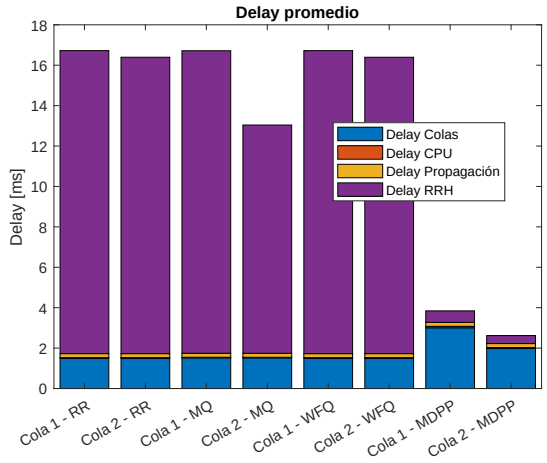
```
20000 1

# Colas: Tamaño , Tasa ,  Prioridad , Tamaño Pkts , RRH
-1 30 4 1000 0
-1 30 2 1300 1

# Scheduler: 0 - RR, 1 - MaxCola, 2 - WFQ, 3 - MinDriftPlusPenalty
3

# Peso delay colas
-10000000

# Peso delay CPU
10000000

# Peso delay RRH
10000000
```

B. Comparación de schedulers por splits

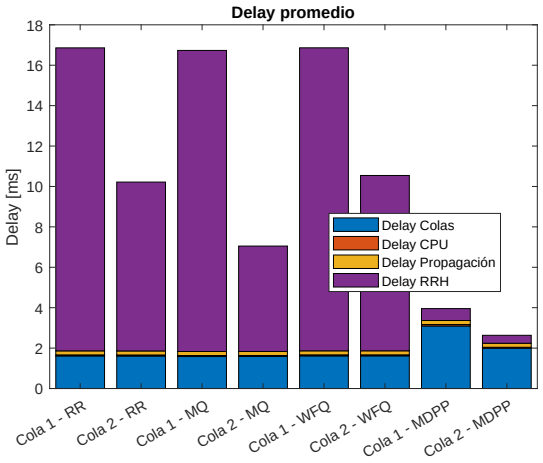
En este Anexo se muestran los resultados de la comparación de los distintos *schedulers* en los diferentes *splits* en términos de retardo y estabilidad.

En la Figura B.1 se muestran los retardos para RRHs asimétricas, mientras que los correspondientes a RRHs simétricas se tienen en la Figura B.2.

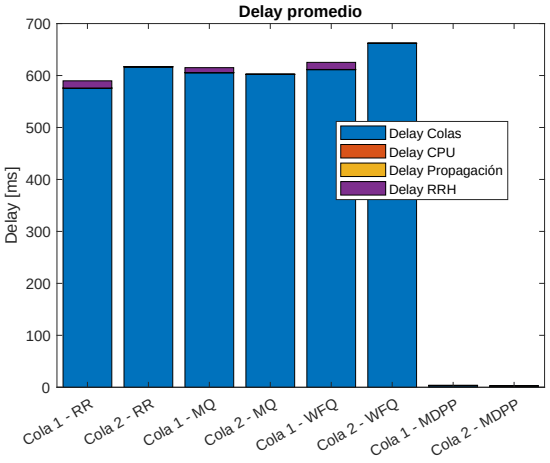
Las Figuras B.3 y B.4 muestran los resultados de estabilidad para RRHs asimétricas y simétricas, respectivamente.



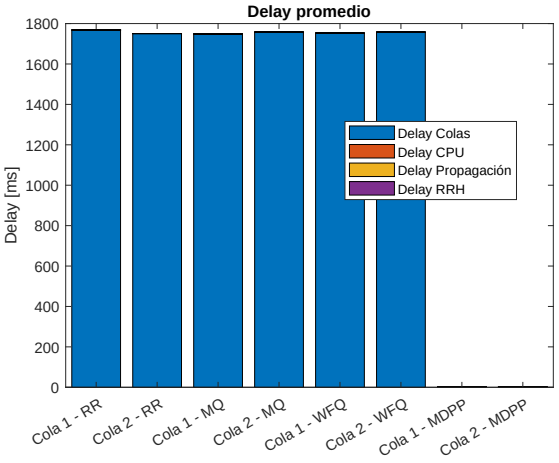
(a) Split 0



(b) Split 1

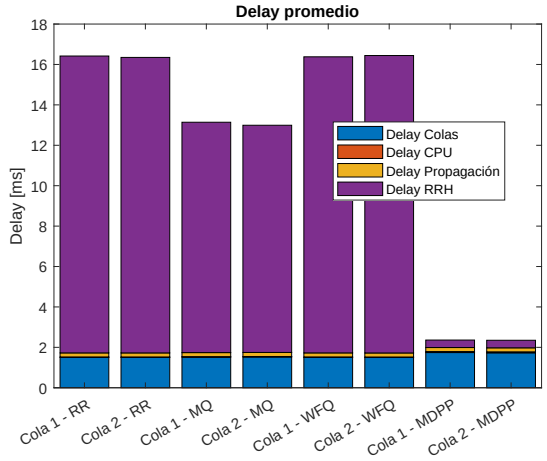


(c) Split 2

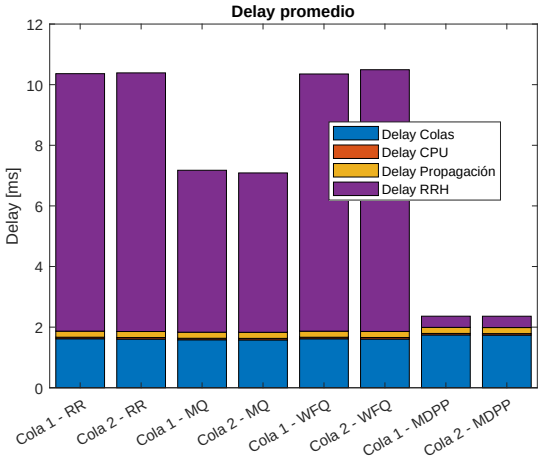


(d) Split 3

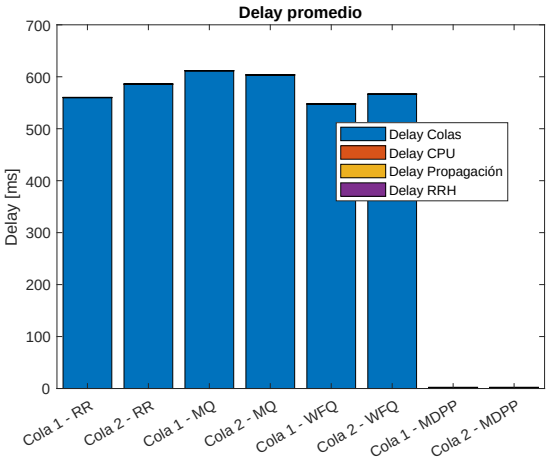
Figura B.1: Resultados del retardo por *split* RRHs asimétricas



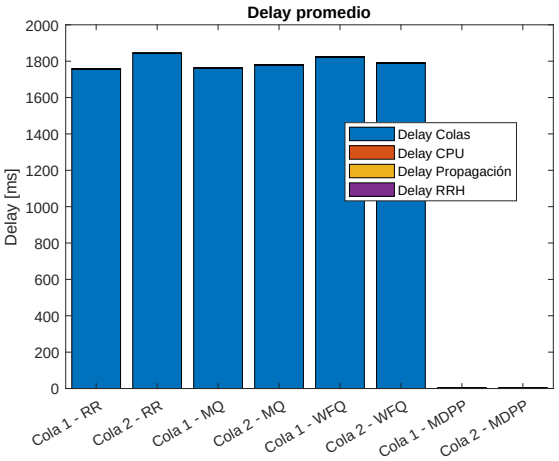
(a) Split 0



(b) Split 1

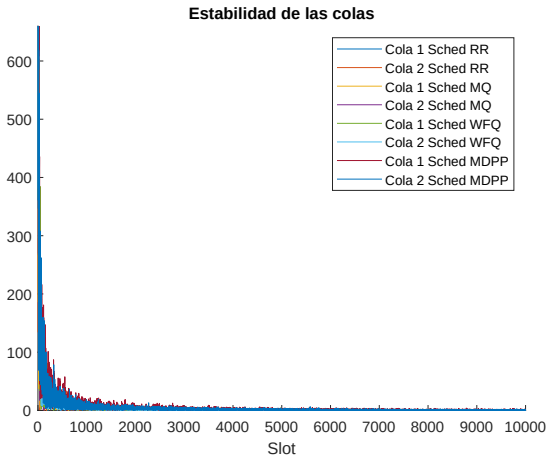


(c) Split 2

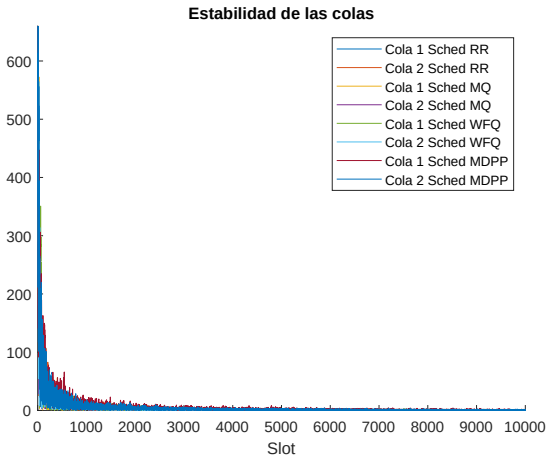


(d) Split 3

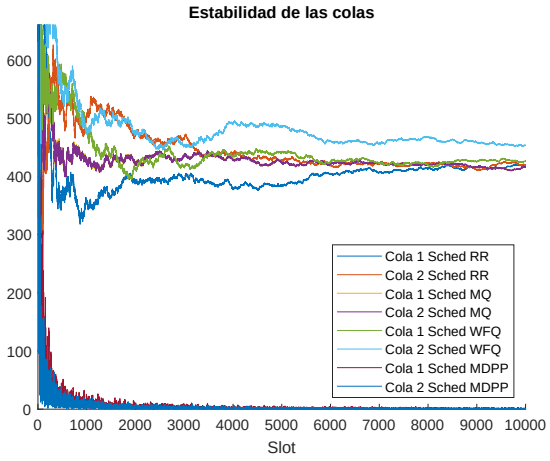
Figura B.2: Resultados del retardo por *split* RRHs simétricas



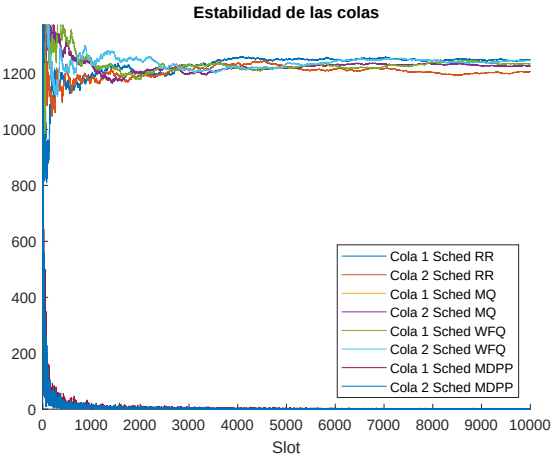
(a) Split 0



(b) Split 1

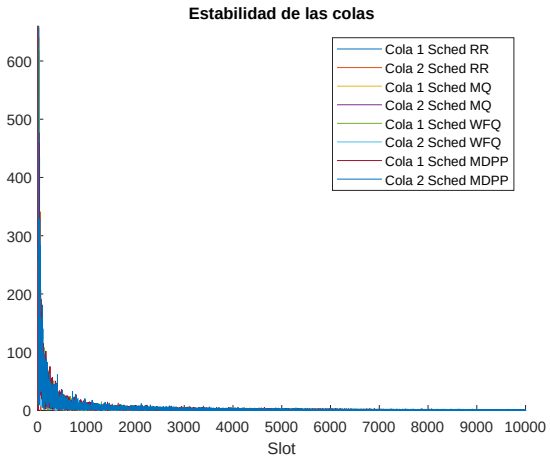


(c) Split 2

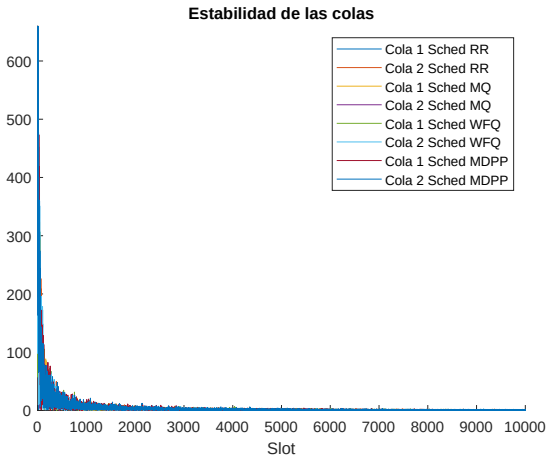


(d) Split 3

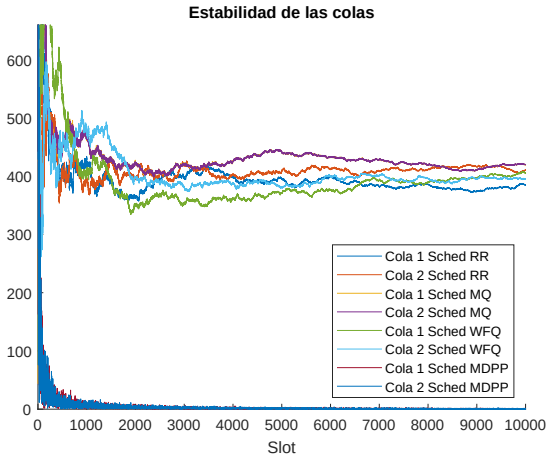
Figura B.3: Resultados de la estabilidad por *split* RRHs asimétricas



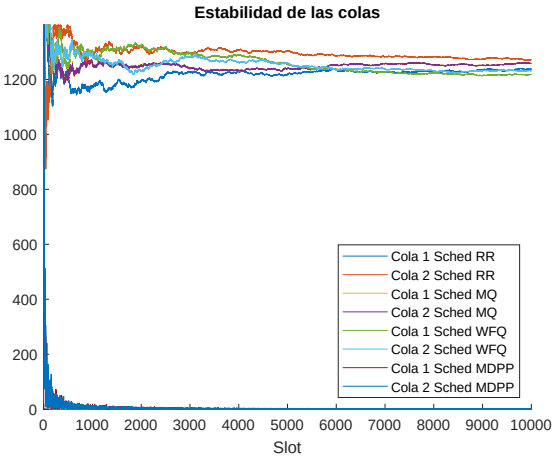
(a) Split 0



(b) Split 1



(c) Split 2



(d) Split 3

Figura B.4: Resultados de la estabilidad por *split* RRHs simétricas

C. Escenario 1 Caso 4: Figuras

Este Anexo contiene las gráficas correspondientes a todas las simulaciones del Escenario 1 Caso 4. En la Figura C.1 se tienen los resultados del retardo, en la Figura C.2 los de la estabilidad, en la Figura C.3 los repartos de nivel de *split*, y en la Figura C.4 el *throughput* de cada simulación.

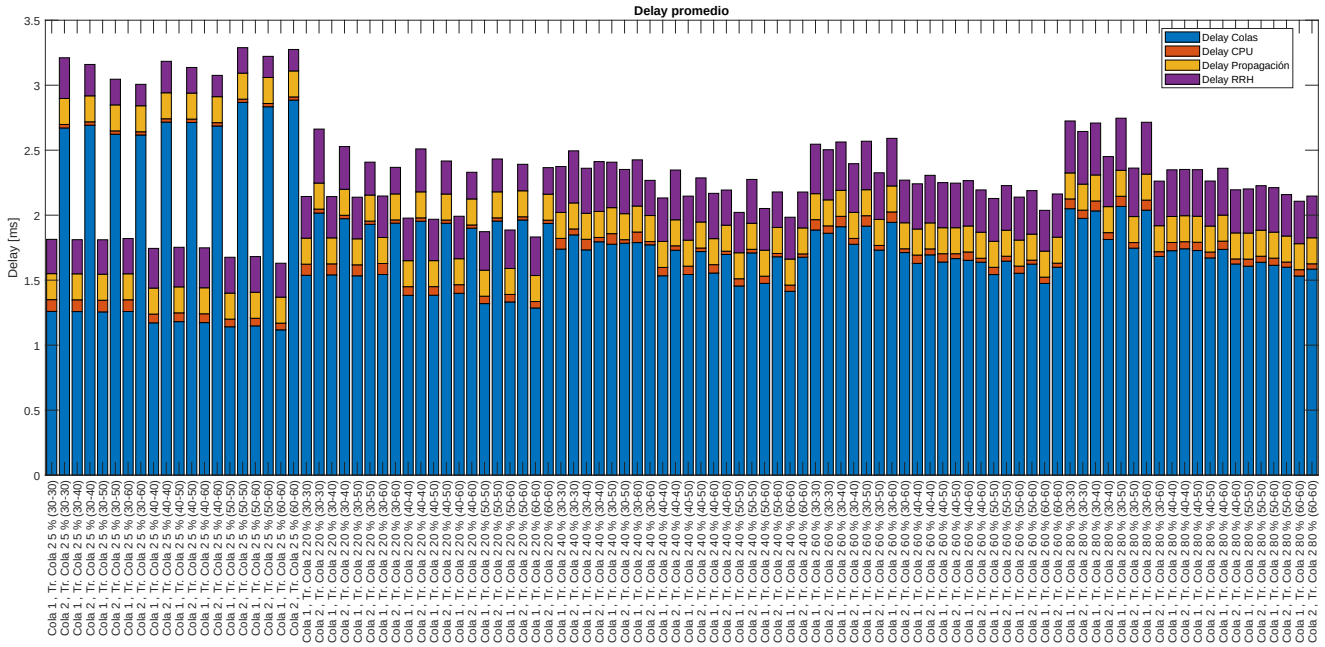


Figura C.1: Escenario 1 Caso 4: Delay

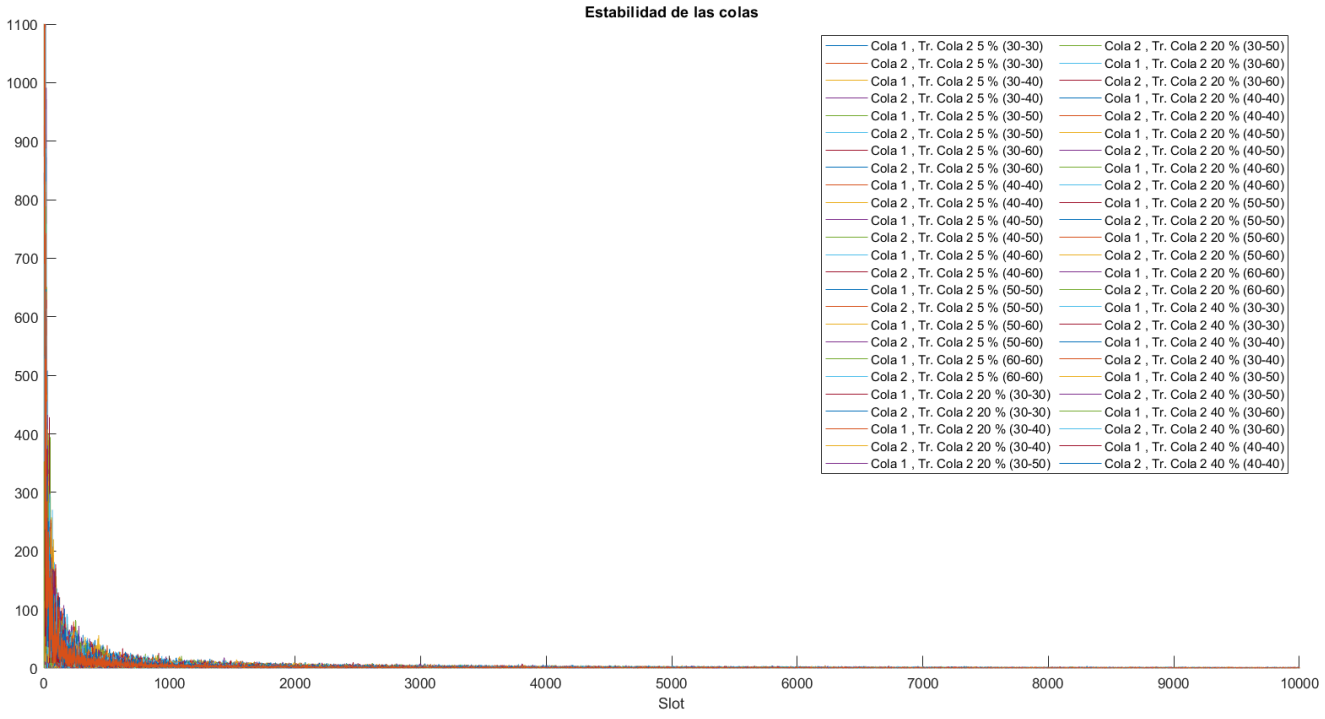


Figura C.2: Escenario 1 Caso 4: Estabilidad

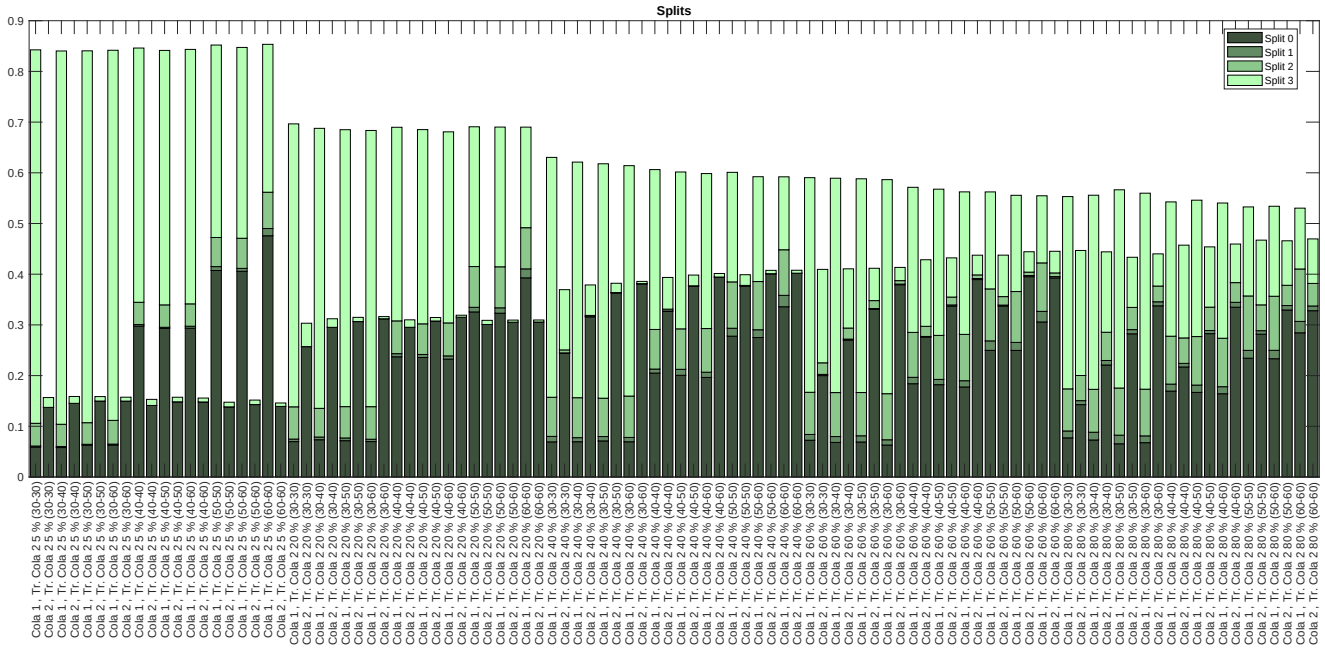


Figura C.3: Escenario 1 Caso 4: Split

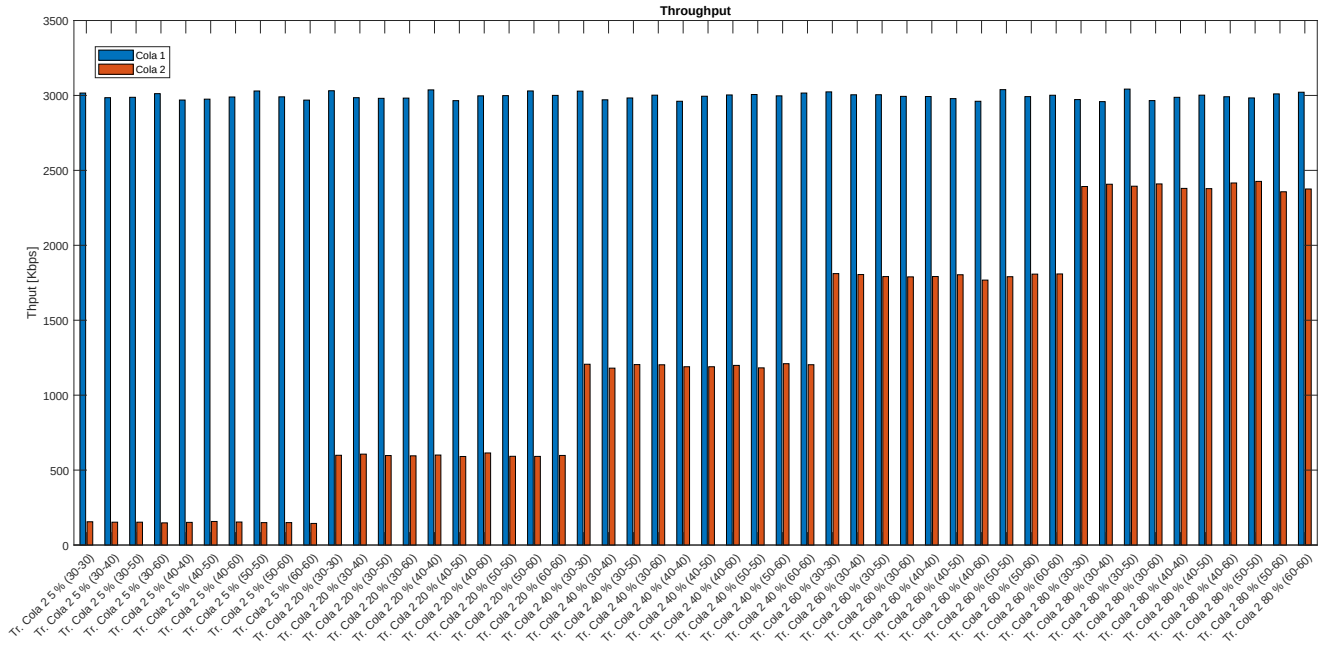


Figura C.4: Escenario 1 Caso 4: Throughput

REFERENCIAS

[1] International Telecommunication Union. *M.2083 : IMT Vision - "Framework and overall objectives of the future development of IMT for 2020 and beyond"*. English. Recommendation ITU-R M.2083-0. ITU-R, sep. de 2015. URL: https://www.itu.int/rec/R-REC-M.2083-0-201509-I/es (visitado 08-09-2020).

[2] Y. Lin y col. "Wireless network cloud: Architecture and system requirements". En: *IBM Journal of Research and Development* 54.1 (ene. de 2010), 4:1-4:12. ISSN: 0018-8646, 0018-8646. DOI: 10.1147/JRD.2009.2037680. URL: http://ieeexplore.ieee.org/document/5429914/ (visitado 09-09-2020).

[3] Cisco. *Cisco Annual Internet Report - Cisco Annual Internet Report (2018–2023) White Paper*. en. Inf. téc. 2020. URL: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html (visitado 08-09-2020).

[4] A. Checko y col. "Cloud RAN for Mobile Networks—A Technology Overview". En: *IEEE Communications Surveys Tutorials* 17.1 (2015), págs. 405-426. ISSN: 1553-877X. DOI: 10.1109/COMST.2014.2355255.

[5] J. Hoydis, S. ten Brink y M. Debbah. "Massive MIMO: How many antennas do we need?" En: *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. Monticello, IL: IEEE, sep. de 2011, págs. 545-550. ISBN: 9781457718175 9781457718182. DOI: 10.1109/Allerton.2011.6120214. URL: http://ieeexplore.ieee.org/document/6120214/ (visitado 09-09-2020).

[6] I. Hwang, B. Song y S. S. Soliman. "A holistic view on hyper-dense heterogeneous and small cell networks". En: *IEEE Communications Magazine* 51.6 (jun. de 2013), págs. 20-27. ISSN: 0163-6804, 1558-1896. DOI: 10.1109/MCOM.2013.6525591. URL: https://ieeexplore.ieee.org/document/6525591/ (visitado 08-09-2020).

[7] D. Gesbert y col. "Shifting the MIMO Paradigm". En: *IEEE Signal Processing Magazine* 24.5 (sep. de 2007), págs. 36-46. ISSN: 1053-5888. DOI: 10.1109/MSP.2007.904815. URL: http://ieeexplore.ieee.org/document/4350224/ (visitado 09-09-2020).

[8] China Mobile Researh Institute. "C-ran the Road towards Green Ran". En: oct. de 2011.

[9] J. Wu y col. "Cloud radio access network (C-RAN): a primer". En: *IEEE Network* 29.1 (ene. de 2015), págs. 35-41. ISSN: 1558-156X. DOI: 10.1109/MNET.2015.7018201.

[10] C.-L. I y col. "Toward green and soft: a 5G perspective". En: *IEEE Communications Magazine* 52.2 (feb. de 2014), págs. 66-73. ISSN: 0163-6804. DOI: 10.1109/MCOM.2014.6736745. URL: http://ieeexplore.ieee.org/document/6736745/ (visitado 07-09-2020).

[11] A. Martinez Alba y W. Kellerer. "A Dynamic Functional Split in 5G Radio Access Networks". En: *2019 IEEE Global Communications Conference (GLOBECOM)*. ISSN: 2576-6813. Dic. de 2019, págs. 1-6. DOI: 10.1109/GLOBECOM38437.2019.9013336.

[12] M. Chiosi y col. "Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action". English. En: Darmstadt-Germany, oct. de 2012. URL: http://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[13] S. Ahmadi, ed. *5G NR: Architecture, technology, implementation, and operation of 3GPP new radio standards*. Cambridge: Elsevier, 2019. ISBN: 9780081022672.

[14] 3rd Generation Partnership Project. *Study on new radio access technology: Radio access architecture and interfaces*. English. Technical Report 3GPP TR 38.801 V14.0.0. 3rd Generation Partnership Project, mar. de 2017. URL: https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3056.

[15] A. Checko, H. Christiansen y M. Berger. "Evaluation of energy and cost savings in mobile Cloud RAN". En: *Proceedings of OPNETWORK 2013*. Ene. de 2013.

[16] C.-L. I y col. "Recent Progress on C-RAN Centralization and Cloudification". En: *IEEE Access* 2 (2014), págs. 1030-1039. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2014.2351411. URL: https://ieeexplore.ieee.org/document/6882182/ (visitado 28-09-2020).

[17] 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. ISSN: 1553-877X, 2373-745X. DOI: 10.1109/COMST.2018.2868805. URL: https://ieeexplore.ieee.org/document/8479363/ (visitado 22-09-2020).

[18] I. Chih-Lin y col. "NGFI, the xHaul". En: *2015 IEEE Globecom Workshops (GC Wkshps)*. San Diego, CA: IEEE, dic. de 2015, págs. 1-6. ISBN: 9781467395267. DOI: 10.1109/GLOCOMW.2015.7414147. URL: https://ieeexplore.ieee.org/document/7414147/ (visitado 25-09-2020).

[19] A. Checko y col. "Evaluating C-RAN fronthaul functional splits in terms of network level energy and cost savings". En: *Journal of Communications and Networks* 18.2 (abr. de 2016), págs. 162-172. ISSN: 1229-2370, 1976-5541. DOI: 10.1109/JCN.2016.000025. URL: https://ieeexplore.ieee.org/document/7487973/ (visitado 23-09-2020).

[20] H. Wang, M. Aftab Hossain y C. Cavdar. "Cloud RAN architectures with optical and mm-Wave transport technologies". En: *2017 19th International Conference on Transparent Optical Networks (ICTON)*. Girona: IEEE, jul. de 2017, págs. 1-4. ISBN: 9781538608593. DOI: 10.1109/ICTON.2017.8025007. URL: https://ieeexplore.ieee.org/document/8025007/ (visitado 25-09-2020).

[21] J. Duan, X. Lagrange y F. Guilloud. "Performance Analysis of Several Functional Splits in C-RAN". En: *2016 IEEE 83rd Vehicular Technology Conference (VTC Spring)*. Nanjing, China: IEEE, mayo de 2016, págs. 1-5. ISBN: 9781509016983. DOI: 10.1109/VTCSpring.2016.7504410. URL: http://ieeexplore.ieee.org/document/7504410/ (visitado 28-09-2020).

[22] *Common Public Radio Interface*. URL: http://www.cpri.info/spec.html (visitado 25-09-2020).

[23] P. Rost y col. "Cloud technologies for flexible 5G radio access networks". En: *IEEE Communications Magazine* 52.5 (mayo de 2014), págs. 68-76. ISSN: 0163-6804. DOI: 10.1109/MCOM.2014.6898939. URL: http://ieeexplore.ieee.org/document/6898939/ (visitado 28-09-2020).

[24] D. Harutyunyan y R. Riggio. "Flex5G: Flexible Functional Split in 5G Networks". En: *IEEE Transactions on Network and Service Management* 15.3 (sep. de 2018), págs. 961-975. ISSN: 1932-4537, 2373-7379. DOI: 10.1109/TNSM.2018.2853707. URL: https://ieeexplore.ieee.org/document/8408566/ (visitado 23-09-2020).

[25] A. Maeder y col. "Towards a flexible functional split for cloud-RAN networks". En: *2014 European Conference on Networks and Communications (EuCNC)*. Bologna, Italy: IEEE, jun. de 2014, págs. 1-5. ISBN: 9781479952809. DOI: 10.1109/EuCNC.2014.6882691. URL: http://ieeexplore.ieee.org/document/6882691/ (visitado 30-09-2020).

[26] I. Koutsopoulos. "Optimal functional split selection and scheduling policies in 5G Radio Access Networks". En: *2017 IEEE International Conference on Communications Workshops (ICC Workshops)*. Paris, France: IEEE, mayo de 2017, págs. 993-998. ISBN: 9781509015252. DOI: 10.1109/ICCW.2017.7962788. URL: http://ieeexplore.ieee.org/document/7962788/ (visitado 30-09-2020).

[27] L. F. Díez Fernández, V. González Carril y R. Agüero Calvo. *Optimización conjunta del nivel split y scheduling en redes 5G*. spa. Universidad de Zaragoza, 2019. ISBN: 9788409211128. URL: https://repositorio.unican.es/xmlui/handle/10902/18649 (visitado 30-09-2020).

[28] C.-Y. Chang y col. "FlexCRAN: A flexible functional split framework over ethernet fronthaul in Cloud-RAN". En: *2017 IEEE International Conference on Communications (ICC)*. Paris, France: IEEE, mayo de 2017, págs. 1-7. ISBN: 9781467389990. DOI: 10.1109/ICC.2017.7996632. URL: http://ieeexplore.ieee.org/document/7996632/ (visitado 01-10-2020).

[29] A. M. Alba, J. H. G. Velásquez y W. Kellerer. "An adaptive functional split in 5G networks". En: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. Abr. de 2019, págs. 410-416. DOI: 10.1109/INFCOMW.2019.8845147.

[30] L. Bracciale y P. Loreti. "Lyapunov drift-plus-penalty optimization for queues with finite capacity". En: *IEEE Communications Letters* (2020), págs. 1-1. ISSN: 1558-2558. DOI: 10.1109/LCOMM.2020.3013125.

[31] A. Gowda y col. "Delay analysis of mixed fronthaul and backhaul traffic under strict priority queueing discipline in a 5G packet transport network: Delay analysis of mixed fronthaul and backhaul traffic under strict priority queueing discipline in a 5G packet transport network". en. En: *Transactions on Emerging Telecommunications Technologies* 28.6 (jun. de 2017), e3168. ISSN: 21613915. DOI: 10.1002/ett.3168. URL: http://doi.wiley.com/10.1002/ett.3168 (visitado 19-07-2020).

[32] H. C. Tijms. *A first course in stochastic models*. New York: Wiley, 2003. ISBN: 9780471498803 9780471498810.