Universidad ORT Uruguay
Facultad de Ingeniería

Diseminación de contenido en redes de datos distribuidas

Entregado como requisito para la obtención del título de Ingeniero en Electrónica

Ismael Valentín Rodríguez Brena - 150167
Tutor: Andrés Ferragut
Tutor: Fernando Paganini

2015

DECLARACIÓN DE AUTORÍA

Yo, Ismael Valentín Rodríguez Brena, declaro que el trabajo que se presenta en esa obra es de mi propia mano. Puedo asegurar que:

- La obra fue producida en su totalidad mientras realizaba la carrera de Ingeniería en Electrónica;
- Cuando he consultado el trabajo publicado por otros, lo he atribuido con claridad;
- Cuando he citado obras de otros, he indicado las fuentes. Con excepción de estas citas, la obra es enteramente mía;
- En la obra, he acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, he explicado claramente qué fue contribuido por otros, y qué fue contribuido por mi;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Ismael Valentín Rodríguez Brena
4 de marzo de 2015

AGRADECIMIENTOS

A mi familia y amigos que me acompañaron y apoyaron durante todo el desarrollo de este trabajo, especialmente a Lucía que con su llegada al mundo trajo mucha felicidad para todos.

A mis tutores Andrés y Fernando, quienes me guiaron durante este proyecto y cuya exigencia logró sacar lo mejor de mí.

ABSTRACT

El tema central de este proyecto son las redes de contenido distribuidas o CDN por sus siglas en inglés (*Content Delivery Network*). Estas redes se definen como una red superpuesta de computadoras que almacenan distintos contenidos con el fin de mejorar la experiencia de los usuarios.

Un primer objetivo es la optimización del desempeño de estas redes en función del ruteo y de la distribución de los contenidos a través de la red. En este sentido se presentan varias formulaciones llegando a resultados concluyentes sobre el ruteo en estas redes, pero el problema de optimización global no es tratable.

Constatada esta dificultad se focaliza el trabajo en las políticas de reemplazo en las memorias cachés. Para poder realizar este estudio se desarrolló desde cero un simulador el cual es uno de los mayores aportes de este proyecto. Este simulador presenta gran flexibilidad puesto que permitirá al usuario analizar distintas redes de contenido distribuidas, además posee un formato modular para que quien lo desee pueda modificar el código fuente de forma sencilla para poder agregar diferentes extensiones.

Mediante simulaciones se analiza el comportamiento de los algoritmos clásicos y se proponen alternativas. Estas alternativas lograron, de forma incremental, mejorar la *performance* de las redes en varias métricas de interés. Se presentan aquí estas modificaciones acompañadas de los ejemplos que motivaron su diseño.

Estos nuevos algoritmos son puestos a prueba en simulaciones con redes de mayor tamaño respaldando los buenos resultados de pruebas anteriores, demostrando que puede ser una alternativa interesante para redes de caché.

Palabras clave: Redes distribuidas de contenidos; Optimización convexa; Algoritmo de distribución de contenido; Simulador de redes de contenido

Contenido

Capítulo 1

Introducción

1.1 ANTECEDENTES

El desarrollo reciente de Internet se ha visto dominado por un cambio de paradigma, el cual esta transformando la red clásica fuertemente centrada en el envío de datos tradicional desde/hacia servidores en una red donde el centro está en el contenido creado y descargado por los usuarios. Además el crecimiento en la demanda por distintos archivos en la red y mayor complejidad de dichos contenidos provoca la necesidad de cambiar la infraestructura instalada. Es a partir de este cambio de infraestructura donde aparece la necesidad de la redes de entrega de contenidos o CDN por sus siglas en inglés (*Content Delivery Network*). Estas redes son definidas como "*una red superpuesta de computadoras que contienen copias de datos, colocados en varios puntos de una red con el fin de maximizar el ancho de banda para el acceso a los datos de clientes por la red*" en [1]. En este proyecto las computadoras que contienen copias de datos, las llamaremos memorias caché.

Este tipo de infraestructura es de gran ayuda sobre todo cuando se tiene en cuenta que es frecuente que múltiples usuarios accedan a un conjunto pequeño de contenidos. Por esta razón es de gran importancia el diseño de estrategias de cacheo eficientes. Este diseño no es sencillo puesto que involucra varios problemas interesantes. Entre estos problemas se destaca en [2] la necesidad de dimensionar de forma óptima las memorias caché y realizar una distribución inteligente de los contenidos en dichas memorias. Si se resuelven dichos problemas, debería suceder que los servidores replicantes (memorias caché) resuelvan gran parte de la demanda, lo que brinda el beneficio que los tiempos de respuesta percibidos por los usuarios sean menores, debido a la menor latencia en la comunicación con el servidor local.

1.2 ELEMENTOS DE *CONTENT DELIVERY NETWORKS*

La abstracción de las redes de entregas de contenidos que se utilizará es la descripta en [3], la cual es a su vez utilizada y justificada en [4], [5], [6] y [7]. La primera noción a tener en cuenta es que esta red se presenta como un grafo donde cada nodo corresponde a un *caché-router*, el cual se puede ver como un *router* que posee capacidad de almacenamiento, de esta forma el contenido que es redireccionado, también puede ser almacenado. Las aristas en este grafo corresponden a una relación de vecindad, es decir que ambos *routers* están conectados, de manera que si uno no puede atender un pedido se lo puede enviar a cualquiera de sus vecinos.

Además del grafo, los otros elementos de dicho modelo son los archivos, los cuales se replican a través de la red. A lo largo de este trabajo vamos a considerar que todos los archivos son del mismo tamaño, en nuestro caso serán de tamaño unitario, esto se ve justificado a partir de que archivos de tamaños diferentes, se pueden dividir en pedazos todos de la misma medida.

Un agente importante en estas redes de memorias caché, son los llamados *custodians*, donde se almacenan de forma permanente en cada uno de ellos varios o todos los archivos. Una hipótesis que se toma en [3] es que entre todos los *custodians* se almacenan todos los archivos del sistema, además todos los nodos saben cuales son los archivos almacenados por cada *custodian*. Estos elementos se pueden ver como los servidores de contenidos públicos.

En cuanto a la mecánica de funcionamiento de estas redes, los pedidos se pueden generar tanto de forma externa (clientes que piden por un archivo específico) o internos (cuando un nodo pasa el pedido a otro nodo). Cuando un pedido no puede ser atendido por un nodo este lo redirecciona mediante una tabla de ruteo, en la cual dependiendo del nodo y el archivo solicitado se indica a qué nodo se pasa el pedido. El pedido va pasando de un nodo a otro hasta que este es atendido. Si no se producen fallas en los enlaces se puede asegurar que este proceso tiene fin puesto que siempre hay al menos un *custodian* que tiene el archivo. En el caso de que halla fallas y no se pueda resolver el pedido, el sistema deberá informar al cliente que no se encontró el archivo deseado. Sin embargo este trabajo se centrará en escalas de tiempo rápidas, donde se asumirá que la topología se mantiene invariante, por lo tanto todos los pedidos serán resueltos por la red.

Es importante entender también el funcionamiento del reemplazo de archivos en los distintos cachés. En el caso de que el archivo deseado no esté en el caché se dice que se produce un *miss*. Si el archivo deseado se encuentra en el caché donde se está realizando el pedido, ya sea interno o externo, se dice que se produjo un *hit*. Cuando esto ocurre el archivo es descargado hacia el nodo de origen, esta descarga la realizan los nodos enviando el archivo al nodo que originalmente había enviado el pedido, esto tiene como resultado que el contenido vuelva por el mismo camino que se recorrió en búsqueda de éste. Normalmente en estos sistemas, cuando el

pedido se resuelve, al volver por el mismo camino, va guardando copias del archivo en todos los nodos intermedios. En este proyecto se analizarán algunas variaciones a este algoritmo y ver cómo estos cambios afectan la performance del sistema. Algo que puede suceder cuando el archivo se está descargando, es que al guardar en algún nodo el contenido no haya lugar para éste en el caché, y se deba producir un desalojo. El archivo a desalojar va a depender de la política de reemplazo del *buffer* de cada nodo.

Parece natural que se quiera maximizar la cantidad de *hits* y minimizar la cantidad de *miss* debido a que una menor cantidad de *miss* implicaría que se necesiten menos saltos para poder atender el pedido. Al minimizar los saltos de forma indirecta se esta minimizando el retardo en la respuesta al cliente. El *delay* y la cantidad de *hits*, los cuales están fuertemente relacionados, serán las dos medidas que se utilizarán principalmente en este proyecto.

1.3 PROBLEMAS DE DISEÑO DE REDES DE CONTENIDOS

Luego de analizar el estado del arte a grandes rasgos se han identificado los tres problemas principales:

(1) **Asignación de memoria y ubicación de contenidos**: El problema consiste en ubicar los contenidos es las diferentes cachés, de manera de optimizar una noción de costo, a partir de un perfil de demanda de contenidos y de topología de la red. Esta noción de costo de la red normalmente involucra la experiencia de los usuarios (*hits* o retardo percibido). Este problema se analiza en [2] con hipótesis muy restrictivas sobre el perfil de demanda y la topología de la red.

(2) **Ruteo**: Este problema consiste en definir el mapeo entre los clientes que realizan pedidos y los servidores replicantes, esto ha sido estudiado en los artículos [8] y [9], donde se identifican oportunidades de mejora introduciendo coordinación entre las capas de transporte y de red.

Los dos problemas que se han indicado, son los principales en este tipo de redes, pero son complejos de resolver. Además si consideramos el gran tamaño de estas redes, la solución debería ser descentralizada lo cual complica aún más el problema. Por esta razón se ha recurrido al estudio de heurísticas de las políticas de reemplazo de las memorias caché, de donde se desprende el tercer problema.

(3) **Análisis de las políticas descentralizadas de actualización de caché**: El problema a analizar aquí es: mantener actualizados los contenidos almacenados en las distintas memorias, dado que las mismas tienen una restricción de capacidad. En la literatura se analizan diferentes algoritmos de sustitución de los diferentes *buffers* de la red.

Según [10] los algoritmos de sustitución más utilizados son LRU (*Least Recently Used*), FIFO(*First In First Out*) y *Random*. Cada una de estas políticas decide a su manera qué archivo desalojar. En la primera política se desaloja el archivo que hace más tiempo que no es pedido en esa memoria, en el segundo caso se desaloja el archivo que hace más tiempo que está almacenado, mientras que en el último caso se desaloja un archivo al azar.

En [10] se destacan estas estrategias como las más utilizadas sobre por ejemplo LFU (*Least Frequently Used*) donde se desaloja el archivo que se accedió de forma menos frecuente en una última ventana de tiempo. Esto se debe a que estas políticas (LRU, FIFO y *Random*) dan la respuesta a cuál archivo debe ser sustituido en tiempo constante independientemente del tamaño del *buffer*.

En cuanto al análisis de estas políticas en [11] se halla una aproximación al comportamiento de los *buffer* LRU y FIFO aislados y en [12] se intenta realizar una aproximación al comportamiento de una red de *buffers* LRU. A su vez en [10] se realiza un estudio más profundo de la política de sustitución *Random*, pero nuevamente sin llegar a resultados exactos en topologías generales.

1.4 PLAN DE LA TESIS

El presente documento se encuentra organizado de la siguiente manera. En el Capítulo 1 se presentaron las principales características de un sistema de memorias caché, además se mostraron cuáles son los principales problemas a analizar.

En los problemas relacionados a estas redes, la cantidad de nodos y de contenidos tienden a ser muy grandes, por lo cual no parece natural una solución centralizada donde un ente tenga conocimiento de todas estas variables por lo que tiene sentido buscar una solución descentralizada al problema. Puesto que en otros problemas de asignación de recursos en redes se han utilizado métodos de optimización convexa aquí se intentarán aproximaciones similares. Es por esto que en el Capítulo 2, se planteará un modelo matemático, junto con distintas métricas de performance. Además en ese capítulo se trabaja dicho problema, buscando simplificarlo para que este sea resoluble mediante un algoritmo de optimización convexa.

Sin embargo como no fue sencillo realizar una formulación que permita resolver de forma óptima los problemas de la ubicación de contenidos y de ruteo de archivos, se recurrió al estudio de las heurísticas de las políticas de reemplazo. Se estudiarán estas políticas y se analizará cuándo es pertinente almacenar un archivo y cuándo no. Para realizar este estudio se generó un simulador que se introduce en el Capítulo 3. En dicho capítulo se presentará la estructura del simulador, los principales elementos del mismo, entre otros. Además se explicará la secuencia de pedidos y cuáles son los diferentes agentes que atienden a estos pedidos para ilustrar la mecánica del simulador.

Luego en el Capítulo 4, se presentarán diferentes mejoras a los algoritmos ahora

utilizados. Primero se indicará cuál es la falencia a mejorar del algoritmo, luego cuál es la solución, cómo se implementa esta solución en el simulador y por último pruebas que respalden que realmente se introdujeron mejoras en el desempeño del sistema.

En el Capítulo 5, se pondrá a prueba el algoritmo final en diferentes redes de mayor tamaño, para verificar el buen funcionamiento del mismo. Al hacer esto se puede verificar que en redes asimilables a las utilizadas hoy en día, el algoritmo es implementable y que mantiene los buenos resultados observados en el capítulo 4.

Para finalizar en el Capítulo 6 se detallan las conclusiones obtenidas y se presentan las posibles líneas de trabajo a futuro.

Capítulo 2

Performance de redes caché

2.1 MODELO PLANTEADO

En este capítulo se desarrollará un modelo matemático de las redes caché, indicando las distintas aproximaciones, así como las distintas hipótesis planteadas, además de introducir la notación a utilizar.

Primero plantearemos la topología de la red como un grafo G con N nodos y E aristas. Estos N nodos representan los *routers*-caché de la red mientras que las aristas, como se especificó en el capítulo anterior representan relación de vecindad. En un comienzo estas aristas sólo tendrán la información de la conectividad entre nodos, aunque más adelante se le asignará un parámetro de costo el cual será asociado al *delay* del enlace. En particular cuando nos referimos a los distintos nodos, utilizaremos los subíndices i y j. Cada nodo tiene una tasa de pedidos por segundo exógenos o externos (que llamaremos *rate*), por todos los archivos. Notaremos el *rate* de pedidos externos en el nodo i como λ_i , más adelante en este capítulo se discutirán algunas hipótesis sobre la independencia de estos pedidos. En cuanto al tamaño de los *buffers* se notará b_j a la cantidad de archivos que puede almacenar el *buffer* asociado al nodo j. Una hipótesis que se utilizará en este modelo es que todos los nodos tienen un *buffer* no nulo asociado ($b_j > 0 \; \forall j$) o un *custodian* asociado.

En cuanto a los archivos, estos se representan por un conjunto $F = \{f_1, ..., f_K\}$, que es global a todo el sistema. En cada uno de los nodos i, la probabilidad de que un cliente pida un archivo k en particular se notará p_i^k. Se han realizado distintos estudios de los perfiles de demanda de Internet entre ellos [13] y [14]. Estos estudios justifican que la distribución de probabilidades de pedidos por archivos en un nodo, es la llamada $Zipf(\alpha)$ donde $p^k = A.\frac{1}{k^\alpha}$ siendo A una constante de normalización. A su vez se utilizará la notación x_i^k para indicar el *rate* de pedidos externos por el archivo k en el nodo i.

Nuevamente recurriendo al modelo planteado en [3], utilizaremos las dos hipótesis

que plantean:

- *Zero Download Delay* de ahora en adelante ZDD, es la hipótesis, que el tiempo de descarga de un archivo es mucho menor que el tiempo entre pedidos. Al asumir ZDD se acepta que todos los movimientos de archivos realizados en las distintas cachés se producen antes del siguiente pedido externo, ya sea el guardar todos los archivos en el camino como sucede en el algoritmo clásico o los cambios que se verán en los diferentes algoritmos planteados. Esta hipótesis es justificada en [11], [7], [15] y [16]. Sin embargo en el capítulo 4 cuando se vean los resultados de las simulaciones se analizará cuál es el efecto que tiene asumir o no esta hipótesis.
- *Independent Reference Model* a partir de ahora IRM, es la hipótesis de que todos los pedidos externos son independientes entre si e independientes de los pedidos anteriores, esta hipótesis es discutida en [11] y [7]. Por esta razón se asumirá que los pedidos arriban como un proceso de Poisson. Debido a la independencia de pedidos exógenos se tiene que $x_i^k = p_i^k \lambda_i$.

Por último es importante remarcar que alguna de las métricas que se medirán son el *delay* promedio y la cantidad de *hits* en cada nodo. Como se explicó en el capítulo anterior estas dos métricas están fuertemente relacionadas. Además se medirá la discrepancia de contenidos en los diferentes *buffers*. Esta idea surge a partir de que el aumentar la variedad de archivos en una ruta implica un mejor aprovechamiento de la memoria de la red. Sin embargo de las tres medidas a la que se le dará prioridad es al *delay* puesto que es la métrica que perciben los distintos clientes de la red.

2.2 FORMULACIONES DEL PROBLEMA

En esta sección se realizará el estudio tanto del ruteo óptimo como la distribución ideal de los contenidos dadas la topología y los *rates* de pedidos en los diferentes nodos, antes explicitados como x_i^k. Para intentar resolver dicho problema se buscarán problemas de optimización convexa, como los vistos en el curso de maestría Optimización Convexa Aplicada a Redes, el cual se tomó como complemento para este proyecto.

Como presentación de los resultados hallados, se introducirá el problema inicialmente planteado y las diferentes formulaciones de éste a medida que se avanza. En cada paso se explicitará cuál es la idea de la nueva formulación y los problemas que presenta.

2.2.1 Adaptación del problema de *multicommodity*

La formulación del problema que se desea resolver surge como una variación del problema de optimización *multicommodity* el cuál se describe a continuación.

Para el problema de optimización *multicommodity* se considera una red formada por enlaces $l \in \mathcal{L}$, a partir de los cuales se definen rutas $r \in \mathcal{R}$. Estas rutas, por convención se tomará que comienzan en los servidores y finalizan en los clientes que desean realizar una descarga.

Se definen las clases de tráfico o "*commodities*" por un índice $k \in \mathcal{K}$. Notaremos con z_r^k al *rate* de pedidos por la clase de tráfico k en la ruta r. A su vez estos *rates* deben responder a la demanda de los clientes por cada uno de los *commodities* en cada uno de los nodos. La demanda por el *commodity* k en el nodo i se notará x_i^k. También se define y_l como el *rate* de pedidos que atraviesa el enlace l.

Las variables descriptas deben satisfacer las identidades básicas:

$$y_l = \sum_k \sum_{r \ni l} z_r^k, \quad x_i^k = \sum_{r \; termina \; en \; i} z_r^k$$

Finalmente queremos minimizar el costo total de la red, el cual está dado por una función ϕ_l para cada uno de los enlaces. En forma resumida podemos escribir el problema como:

Problema multicommodity

$$\min \quad \sum_l \phi_l(y_l) \tag{2.1}$$

$$s.t. \quad y_l = \sum_k \sum_{r \ni l} z_r^k \tag{2.2}$$

$$x_i^k = \sum_{r \; termina \; en \; i} z_r^k \tag{2.3}$$

Las diferencias entre este problema y el problema que se desea resolver es que ahora las clases de tráfico o *commodities* corresponden en su lugar a un archivo particular.

A modo de simplificar la notación de las ecuaciones, se indica $\mathcal{R}_i$ como el conjunto de rutas r que tienen su destino en el nodo i, de forma análoga se define $\hat{\mathcal{R}}_j$ como el conjunto de rutas que tienen comienzo en el nodo j. En el problema planteado (al igual que en el problema *multicommodity*) siempre que se refiera a rutas, serán las rutas de descarga por lo tanto estas comienzan en las memorias caché y finalizan en los clientes.

Dada esta notación se desprende de forma inmediata la transformación de la ecuación (2.3) por:

$$\sum_{r \in \mathcal{R}_i} z_r^k = x_i^k$$

Esta ecuación indica que la suma de *rates* de descargas por el archivo k que finalizan en i es igual a la demanda por dicho archivo x_i^k. Esta restricción es equivalente a decir que se atienden todos los pedidos externos al sistema.

En cuanto a la restricción (2.2), al igual que en el problema de *multicommodities* se tiene a y_l como el *rate* en que atravisea el enlace l, en este caso el *rate* de descarga. En particular se tiene que y_l es el acumulado del *rate* de descarga por todos los archivos de todas las rutas que contienen al enlace l. Esta reformulación mantiene la restricción como:

$$y_l = \sum_k \sum_{r \ni l} z_r^k$$

En cuanto a la función objetivo al igual que en el problema original se tienen funciones de costo ϕ_l sobre los *rates* de los enlaces. La única restricción que rige sobre dichas funciones es que todas sean convexas. En general esta función objetivo será asociada a un costo por esta razón se desea minimizar como se explicita en la ecuación (2.1).

Volviendo a lo discutido en la sección anterior, una de las métricas que se trató como interesante es la que implicaba el retardo (*delay*) que percibían los clientes. Asumiremos entonces que el *delay* de un enlace l es constante y de valor c_l. Si definimos Λ como el *rate* total que ingresa al sistema es decir:

$$\Lambda = \sum_i \lambda_i$$

También definiendo τ como el *delay* total se tiene que la cantidad de pedidos que están en la red es

$$\Lambda\tau = \sum_l c_l y_l$$

A partir de lo que se deduce

$$\tau = \sum_l \frac{c_l y_l}{\Lambda}$$

Entonces cambiando la función ϕ_l por el *delay* del enlace $\frac{c_l y_l}{\Lambda}$ obtenemos como nueva función objetivo el *delay* total. En particular como Λ es constante, se puede quitar de la función objetivo, puesto que minimizar $\frac{c_l y_l}{\Lambda}$ es equivalente a minimizar $c_l y_l$. Entonces podemos escribir la función objetivo como:

$$\min \qquad \sum_l c_l y_l$$

Por último en esta nueva formulación se deben incluir las restricciones sobre el tamaño de los *buffers*. Notamos b_j como el tamaño del *buffer* asociado al nodo j, por lo tanto se podrán guardar hasta b_j contenidos distintos en el nodo j. Una forma de ver que se almacenó un archivo k en el nodo j, es que hay una ruta con *rate* positivo ($z_r^k > 0$) por el archivo k que comienza en el nodo j. La idea detrás de esta formulación es que si la ruta (recordar que se habla de descarga) comienza en el nodo j, significa que el archivo k está siendo descargado de este nodo, lo que significa que el archivo está almacenado allí. Por lo tanto la restricción se traduciría a que el cardinal del conjunto de archivos de los cuales existe una ruta que comienza en el nodo j sea menor a b_j, escrito en la notación antes presentada sería:

$$\#\{k/\exists r \in \hat{\mathcal{R}}_j / z_r^k > 0\} \leq b_j$$

Habiendo introducido la notación y descripto la transformación del problema *multicommodity* se presenta la formulación del problema a resolver:

Problema 1

$$\min \qquad \sum_l c_l y_l \tag{2.4}$$

$$s.t. \qquad \sum_{r \in \mathcal{R}_i} z_r^k = x_i^k \tag{2.5}$$

$$y_l = \sum_k \sum_{r \ni l} z_r^k \tag{2.6}$$

$$\#\{k/\exists r \in \hat{\mathcal{R}}_j / z_r^k > 0\} \leq b_j \tag{2.7}$$

Es importante destacar que la restricción (2.7) no es convexa, por lo tanto el problema con esta formulación no es resoluble mediante métodos de optimización convexa.

Además otro problema de esta formulación es la gran cantidad de variables que tiene, debido a que la cantidad de rutas r crece rápidamente al incrementar la cantidad de nodos. Al tener tantas variables intentar resolver redes con unos 10 nodos se vuelve extremadamente complejo, por esta razón se intenta una nueva formulación del problema sin tener que escribir todas las rutas y al lograrse esto tener una cantidad más reducida de variables.

2.2.2 Simplificación del Ruteo

Para reescribir el problema nuevamente, primero se demostrará que tomando al *delay* como la función objetivo, de todas las rutas que van del nodo j al nodo i, las únicas rutas que nos interesarán serán las de costo mínimo (menor retardo acumulado), ya que la utilización de las otras rutas nunca pueden ser óptima. Antes de presentar la demostración es importante entender que a partir de este resultado se esta resolviendo el ruteo óptimo, el cual efectivamente siempre será el camino mínimo a dónde esté ubicado el archivo requerido.

Proposición 2.2.1. *En la solución del problema de optimización, podrán tener rate positivo únicamente las rutas de costo mínimo entre pares de nodos.*

Demostración. Tenemos que la función objetivo se puede reescribir como

$$\sum_l c_l y_l = \sum_l c_l \sum_k \sum_{r \ni l} z_r^k$$

Esto surge de unir la ecuaciones (2.4) y (2.5). Ahora reescribiendo la suma nuevamente ya que el costo del enlace es independiente del archivo se tiene

$$\sum_l c_l \sum_k \sum_{r \ni l} z_r^k = \sum_k \sum_l c_l \sum_{r \ni l} z_r^k$$

Como último cambio podemos escribir de una forma más cómoda la expresión al sustituir

$$\sum_k \sum_l c_l \sum_{r \ni l} z_r^k = \sum_k \sum_r z_r^k \underbrace{\sum_{l \in r} c_l}_{c_r} = \sum_k \sum_r c_r z_r^k$$

Utilizando este cambio donde c_r es el costo de la ruta, se escribe la suma en base a las rutas r en lugar de los enlaces l. Si se analiza el resto de las restricciones como son las ecuaciones (2.6) y (2.7), no hay restricciones sobre las rutas en si mismas sino únicamente en los nodos de comienzo y fin de estas (sólo aparecen $\mathcal{R}_i$ y $\hat{\mathcal{R}}_j$ en las restricciones). Por lo tanto si tenemos dos rutas r y $\hat{r}$ que van ambas desde el nodo j al nodo i no tendrán diferencias en las restricciones, pero la que minimiza el costo es la ruta cuyo costo sea menor. Se concluye que de todas las rutas $r \in \mathcal{R}_i \cap \hat{\mathcal{R}}_j$ la que minimiza la función objetivo es aquella de menor costo c_r.

□

En el caso de que haya dos rutas con el mismo costo mínimo se selecciona una de forma arbitraria. Para evitar este problema, de ahora en adelante asumiremos que no hay dos rutas mínimas con el mismo costo.

2.2.3 Formulación simplificada

Ahora se notará $j \to i$ a la ruta de costo mínimo desde j hasta i y notaremos también z_{ji}^k al *rate* de dicha ruta. Esto permite eliminar la notación de los conjuntos $\mathcal{R}_i$ y $\hat{\mathcal{R}}_j$, puesto que en la notación recién indicada queda implícito el origen y destino del *rate* de esa ruta.

Además resuelve el problema de disminuir la cantidad de variables que se manejan en el problema, puesto que se disminuyó a N^2K variables. Esto se debe a que ahora no debemos considerar rutas sino pares de nodos (N^2) y por cada par de nodos hay una variable por archivo, por lo tanto en total hay N^2K variables. Al disminuir la cantidad de incógnitas el problema es escalable a redes de mayor tamaño. A continuación se presenta la nueva formulación del problema

Problema 2

$$\begin{aligned}
\min \quad & \sum_l c_l y_l \\
s.t. \quad & \sum_{j \in N} z_{ji}^k = x_i^k && (2.8) \\
& y_l = \sum_k \sum_{j \to i \ni l} z_{ji}^k && (2.9) \\
& \#\{k : \sum_i z_{ji}^k > 0\} \leq b_j && (2.10)
\end{aligned}$$

Se observa que la función objetivo no varía. En cuanto a las restricciones se produce un mapeo inmediato entre el Problema 1 y el Problema 2. Analizando la ecuación (2.8) es idéntica a (2.5) a excepción que ahora no es necesario indicar la suma en el conjunto R_i, que ahora se transforma en la suma en todos los nodos j ya que la restricción está indicada en la nueva notación. La ecuación (2.9) no tiene ningún cambio sin ser por la notación. Por último resta comparar las ecuaciones (2.7) y (2.10), en la nueva formulación no es necesario el conjunto $\hat{R}_j$, y se cambia la existencia de una ruta y en su lugar se exige que la suma sea mayor a 0 ya que ambas formulaciones son equivalentes.

Igualmente el problema no es convexo puesto que la restricción (2.10) no involucra una función convexa, además de no ser sencilla para utilizar en los diferentes cálculos. Entonces el próximo paso es reformular esta ecuación utilizando una idea que se presenta en [2]. Dicha idea consta en la utilización de nuevas variables

$e_i^k = \{0, 1\}$ siendo esta variable la indicatriz de que el archivo k está almacenado en el nodo i, es decir

$$e_j^k = \mathbf{1} : \{\sum_i z_{ji}^k > 0\}$$

Utilizando estas variables (en total son NK), la restricción conflictiva es mucho más sencilla de escribir ya que se tiene de forma explicíta si se guarda o no un archivo en un nodo en particular.

El problema que tiene incluir estas nuevas variables es que las mismas son enteras, por lo cual este problema pasa a tener una parte de programación entera. A continuación se presenta la adaptación del problema para esta nueva notación con dichas variables.

Problema 3

$$\begin{aligned}
\min \quad & \sum_l c_l y_l \\
s.t. \quad & \sum_j z_{ji}^k e_j^k = x_i^k && (2.11) \\
& y_l = \sum_k \sum_{j \to i \ni l} z_{ji}^k && (2.12) \\
& \sum_k e_j^k \leq b_j && (2.13)
\end{aligned}$$

La función objetivo no cambió por lo que no necesita explicación, así que se pasará directo a las restricciones. Comenzaremos por la ecuación (2.13) que aparece en sustitución a la restricción (2.10). La nueva formulación es mucho más sencilla, además de ser de fácil compresión ya que se indica que en total hay guardado hasta b_j archivos, ya que cada archivo que es guardado en el nodo j aporta 1 a la sumatoria.

Es importante notar el cambio en la ecuación (2.11) ya que ahora aparece el producto $z_{ji}^k e_j^k$, ya que para que un *rate* z_{ji}^k aporte a los pedidos x_i^k debe comenzar su descarga en un nodo j que efectivamente contenga el archivo k. Esta restricción podría haber sido escrita separada en dos, por un lado escribiendo la restricción de la indicatriz y por otro lado la restricción de la respuesta a los clientes. Formalmente

$$\sum_j z_{ji}^k = x_i^k \quad e_j^k = \mathbf{1} : \{\sum_i z_{ji}^k > 0\}$$

A diferencia de la ecuación antes descripta en la restricción (2.12) no es necesario realizar el producto de las variables debido a que nunca puede ser óptimo en este

caso tener $z_{ji}^k > 0$ si $e_j^k = 0$, ya que se aumentaría y_l aumentando el costo total sin afectar el resto de las restricciones.

Una idea que surge inmediatamente al tener el inconveniente de las variables enteras es relajar el problema, tomando las variables e_i^k como reales y a partir de los resultados que se obtengan aproximarlos a soluciones enteras válidas. Sin embargo relajar el problema no es suficiente para obtener un problema de optimización convexa ya que el producto de la restricción (2.11) no posee una formulación convexa.

El siguiente cambio consta en omitir las variables y_l de las formulaciones anteriores para escribir el problema de una manera más reducida.

Además se realiza una simplificación de la función de costo a partir de que

$$\sum_l c_l \sum_k \sum_{j \to i \ni l} z_{ji}^k = \sum_k \sum_l c_l \sum_{j \to i \ni l} z_{ji}^k = \sum_k \sum_{i \in N,\, j \in N} z_{ji}^k \underbrace{\sum_{l \in j \to i} c_l}_{c_{ij}} = \sum_{i \in N,\, j \in N} \sum_k c_{ij} z_{ij}^k$$

Siendo c_{ij} el costo de la ruta mínima que une el nodo i con el nodo j.

Se presenta a continuación la nueva formulación;

Problema 4

$$\begin{aligned} \min \quad & \sum_{i \in N, j \in N} \sum_k c_{ij} z_{ji}^k \\ s.t. \quad & \sum_j z_{ji}^k e_j^k = x_i^k && (2.14) \\ & \sum_k e_j^k \leq b_j && (2.15) \end{aligned}$$

En este cambio se mantienen las restricciones (2.14) y (2.15), simplemente se une la función objetivo con la restricción (2.12), utilizando las igualdades antes planteadas.

Aunque el problema sigue sin ser resoluble utilizando técnicas de optimización convexa, se pueden deducir algunos resultados. El primero es la resolución del ruteo óptimo si se conocen las ubicaciones de todos los archivos. También se tiene que de todos los *rates* z_{ji}^k solamente existe un único j tal que $z_{ji}^k = x_i^k$ y para el resto de los nodos tiene un *rate* nulo. Este resultado tiene sentido ya que todos los pedidos realizados por un archivo k en un nodo i, serán descargados desde el mismo caché en el nodo j, que será el más cercano que contenga al archivo k.

Se intentaron otras aproximaciones para resolver este problema. Algunas de estas fueron generar dinámicas de precios mediante el multiplicador de Lagrange y también se realizaron intentos utilizando métodos de optimización estocástica (*Simulated Annealing*) sin éxito. Aunque el problema de optimización global no es tratable, se logró una formulación con una cantidad polinomial de variables.

Capítulo 3

Simulador de redes de contenidos

3.1 INTRODUCCIÓN

En este capítulo se describirá el simulador de redes de entregas de contenidos, el cual se desarrolló en su totalidad a lo largo de este proyecto. Dicho simulador fue implementado basado en el modelo de red descrito en los capítulos anteriores y los resultados a partir de las simulaciones son las que permitirán estudiar el desempeño de los distintos algoritmos de forma heurística.

Si bien se encontraron distintos simuladores de CDN, siendo el más destacado el *simCDN*, descripto en [17], ninguno de éstos realiza la abstracción deseada del problema, puesto que se centran en detalles de implementación de las CDNs que no se deseaban estudiar en esta instancia. A su vez el foco del proyecto consiste en implementar nuevos algoritmos lo cual no era sencillo en este simulador. Como resultado de esto nos propusimos realizar un simulador propio, el cual realizara la abstracción del problema planteado, con énfasis en los algoritmos de reemplazo y almacenamiento de archivos y no en las características intrínsecas de la red. El simulador fue programado en su totalidad en el lenguaje *Java* para utilizar un modelo orientado a objetos para así tener una mayor modularidad permitiendo realizar cambios en varios niveles así como una mayor flexibilidad en cuanto a la posibilidad de implementar diferentes algoritmos en cada subproblema del sistema.

Este capítulo tiene como finalidad ser una introducción al funcionamiento del simulador además de explicar la arquitectura del mismo, indicando cuál fue la razón por la que se tomó cada una de decisiones de diseño del sistema. Esta explicación comenzará indicando los objetivos perseguido en el diseño del simulador (sección 3.2). Luego se realizará una descripción general del sistema, analizando cuáles son los principales elementos del simulador (sección 3.3), para continuar con la descripción del funcionamiento del mismo (sección 3.4). Esta sección tendrá como hilo conductor varios ejemplos a partir de los cuales serán descriptos todos los elementos que surjan al momento de realizar una simulación.

Se sigue con la explicación del mecanismo de la ejecución interna del simulador, detallada en la sección 3.5, luego se detalla el generador aleatorio en la sección 3.6.

Luego en una nueva sección (3.7) se describen las distintas salidas que genera el simulador y la estructura de éstas. Además se agrega una sección para explicitar cómo se realizan los distintos sorteos, los cuales son fundamentales tanto para asegurar el buen funcionamiento como la veracidad de los resultados. Por último se detallan algunas pruebas realizadas para comprobar el correcto funcionamiento del simulador (sección 3.8).

3.2 OBJETIVOS PERSEGUIDOS

Para el desarrollo de este simulador se persiguieron varios objetivos distintos para que este fuera fiable y que permita simular una gran variedad de situaciones.

Uno de los principales objetivos es modelar topologías de forma arbitraria y no únicamente topologías de árbol o anillo. Para alcanzar este objetivo era necesario poder resolver el problema de ruteo, el cual no es trivial para topologías más complicadas. Sin embargo este problema se pudo resolver gracias al resultado hallado en el capítulo anterior sobre el ruteo. Por esta razón en el simulador se implementa que el camino a seguir para resolver un pedido sea el más corto hacia el *custodian* del archivo deseado.

Otro objetivo que se plantea para el simulador es que pueda resolver pedidos de forma asíncrona. Esto significó un cambio radical en la forma en que se estructuró el problema, puesto que pueden haber múltiples eventos que no sean de forma sincrónica. Se decidió estructurar el simulador como uno de eventos discretos, esto tiene la ventaja que permite emular de una manera mucho más fidedigna la realidad.

Se plantea también como objetivo que se puedan implementar distintas políticas de reemplazo en los *buffers*. Para lograr esto fue esencial el uso de la programación orientada a objetos. Al utilizar este tipo de programación se pudo generar una interfaz común a todos los *buffers* para la interacción con el resto de los elementos del sistema, mientras que a cada tipo de *buffer* se le carga su política de reemplazo particular.

Por último se esperan reflejar las distintas estadísticas de los pedidos además de poder medir las distintas métricas discutidas en capítulos anteriores. Para hacer esto posible el simulador imprime como salida diferentes archivos de texto con información suficiente para calcular todos estos valores.

3.3 DESCRIPCIÓN GENERAL

Este simulador recibe como entrada un archivo de texto plano, el cual indica las diferentes características de la red, como su topología, la ubicación de clientes, cachés, *custodians*, la cantidad total de archivos, entre otros. A partir de esta entrada se genera un elemento simulador que es el encargado de llevar el hilo de la simulación y generar las salidas las cuales permiten medir las diferentes métricas que se han discutido a través de esta documentación. Estas salidas son mediante archivos de texto plano, debido a que de esta manera las simulaciones se realizan de forma mucho más rápida que si se imprimen los datos mediante la salida estándar. En algunos casos esta mejora produce que la simulación sea hasta 30 veces más rápida. A lo largo del simulador se intentó utilizar las estructuras y algoritmos más eficientes en cuanto a tiempo de ejecución para que de esta manera se puedan realizar simulaciones de gran tamaño sin que el tiempo sea una restricción importante.

Para facilitar tanto el uso como la comprensión de este simulador se anexa un manual de usuario y se incluye en el medio óptico el código fuente del programa y la documentación de éste (*JavaDoc*). A continuación se comenzarán a desglosar los diferentes elementos del simulador para una descripción en mayor detalle.

3.3.1 Descripción de los principales elementos

Los principales elementos de todo simulador de eventos discretos son los agentes, los eventos y los generadores de números aleatorios. Los eventos en este simulador emularán los diferentes mensajes, pedido de archivos y descargas que se manejan en la red. Los agentes son quienes se encargan de atender y generar los distintos eventos. Al mismo tiempo un generador de números aleatorios es el encargado de generar los parámetros de los eventos que serán creados por los agentes.

En cuanto a los eventos, se generan dos clases de *Java*, que son `Evento` y `listaDeEventos`. La primera clase modela al evento en si, indicando los diferentes parámetros de éste, como el agente que lo generó y el mensaje que porta entre otros. La segunda clase encapsula una cola ordenada de eventos (en *Java* se implementará mediante una *PriorityQueue*), ordenando estos por el tiempo en el que deben ser atendidos. La clase `Simulador`, la cual es la clase principal del programa, contará como una de sus variables una lista de eventos, donde se agendarán los eventos a ejecutar. Además esta clase mantendrá la información de todos los elementos de la red como la topología de la misma, ubicación de cada uno de los elementos y la interacción entre estos. Esta clase principal será desarrollada más adelante en este capítulo.

Continuando con el análisis de la implementación a grandes rasgos, se observa que se implementa una clase abstracta llamada `Agente`. Heredan de esta clase los diferentes agentes del sistema de red de contenidos. En este simulador las clases que

modelan los nodos y los clientes son las que heredan la interfaz de agente. Es importante recordar que en el modelo utilizado los nodos no sólo representan la ubicación geográfica sino que son la representación de los *routers*. Es por esta razón que tiene sentido verlos como agentes ya que generarán eventos de ruteo de los mensajes, ruteo que se verá como una secuencia de eventos en el simulador. Es claro que los clientes son agentes puesto que son ellos los que generan todos los pedidos de la red. Cuando el pedido es completado es el cliente el encargado de recibir finalmente la descarga para calcular el *delay*, es decir el retardo incurrido en completar el pedido. También estos clientes, al atender pedidos que son recientemente creados, son los encargados de generar automáticamente otro pedido para que de esta manera se mantenga el flujo de estos con el *rate* deseado.

Además en esta descripción se estudiarán los gestores de archivos o `FileManager`. Estos gestores de archivos son los encargados de almacenar los contenidos y tomar decisiones sobre cuáles de estos deben ser desalojados. En el caso de las redes de contenidos, tenemos dos elementos principales, que son los cachés y los *custodians*. En el simulador cada uno de estos dos elementos se representa como una clase, y ambas heredan de la clase abstracta `FileManager`. Las principales funcionalidades de esta clase son: mantener una lista de contenidos, encargandosé principalmente de contestar si un archivo se encuentra almacenado o no y de tomar las decisiones referentes a si se debe desalojar o no un archivo y en caso afirmativo decidir cuál es este archivo. Para tomar esta decisión los cachés deben tener en cuenta las políticas de reemplazo de sus *buffers*. Lo cual lleva a la creación de una nueva clase abstracta `Buffer` que modela la memoria interna del caché. Los diferentes tipos de *buffer* (LRU, FIFO y *Random*), heredan de esta clase abstracta, implementando según su política de reemplazo los métodos que están encargados de resolver qué archivo desalojar.

3.4 DESCRIPCIÓN DEL FUNCIONAMIENTO

Para realizar el análisis del funcionamiento de cada una de las partes del simulador, lo dividiremos en diferentes elementos los cuales serán descriptos a través de un ejemplo en particular. Siguiendo dicho ejemplo se especificará por qué el orden de los datos ingresados es el especificado, y a medida que se ingresan los mismos se expondrá cómo se modelan específicamente en el simulador. Al utilizar el ejemplo como guía permitirá que el lector se adentre en los detalles de la arquitectura de cada una de las clases implementadas y la razón de ser de cada una de las variables principales. Además se recorrerán los diferentes algoritmos implementados, y se observarán las decisiones de diseño tomadas. El análisis se realizará analizando tres secciones separadas: topología, aplicaciones y simulación.

3.4.1 Construcción de la Topología

En primer lugar, la construcción de una simulación comienza con la descripción de la topología. Esto permite verificar diferentes restricciones importantes en cuanto a conectividad de la red y evitar múltiples enlaces entre distintos nodos desde un principio. Supongamos que deseamos construir la topología indicada en la Figura 3.1, que consta de 3 nodos unidos formando una línea.

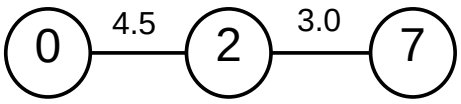

Figura 3.1. Topología que se desea implementar

Para implementar dicha topología se ingresa el siguiente archivo de texto de entrada:

```
begin Topologia
Nodo 0
Nodo 2
Nodo 7
Enlace 0 2 4.5
Enlace 2 7 3.0
end Topologia
```

El formato de la entrada, en cuanto al tipo de variables que se deben ingresar, el espaciado correcto y el orden de los datos, se aclarará en el apéndice correspondiente a Manual de Usuario. La clase `Reader` será la encargada de realizar todas las validaciones de formato y la interpretación del archivo de texto. A partir de esta interpretación se generarán todos los elementos presentados a continuación.

Los primeros elementos que se ingresan son los nodos (o *routers*) de la red. A cada nodo se le asigna un número arbitrario con el que el usuario lo identifica. En el caso del ejemplo en particular, se generan 3 nodos con las etiquetas 0, 2 y 7. En la figura 3.2 se observa la lista de las variables de la clase Nodo que conciernen a la topología de la red, más adelante se describirán las distintas variables que refieren a los diferentes actuadores de la red.

La variable `idUsuario`, es el identificador que le da el usuario al nodo. Este campo será utilizado para la comunicación con el usuario. En el ejemplo, esta variable tendría los valores 0, 2 y 7 para cada nodo respectivamente. Además no esta permitida la creación de varios nodos con el mismo identificador, para evitar que se produzcan confusiones. Este identificador no es utilizado para el procesamiento interno, para esto se utiliza la variable `id`, la cual se asigna de forma automática, numerando los nodos creados de 0 en adelante. La ventaja de esta identificación es

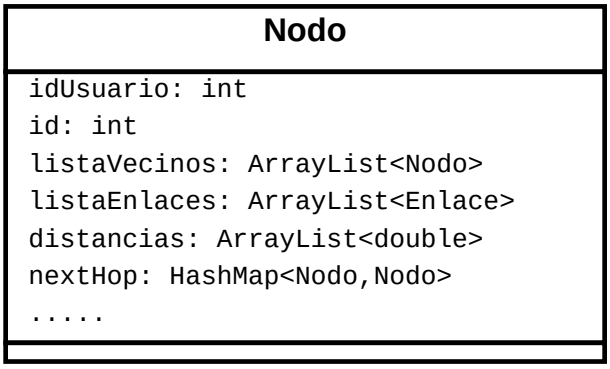

Figura 3.2. Algunas variables de la clase Nodo

que se puede utilizar como indexación de los nodos, es por esta razón que se utiliza para los algoritmos de carácter interno, invisibles al usuario.

El resto de las variables listadas, se relacionan con los enlaces que en el simulador se modelan como una nueva clase. Los enlaces se agregan mediante los comandos que comienzan con este mismo nombre y son los que conectan diferentes nodos. Al ir cargando los datos referentes a las conexiones entre nodos, se actualizan las variables referentes a enlaces, vecinos y distancias de la clase `Nodo`.

En la figura 3.3 se listan las variables de la clase `Enlace`.

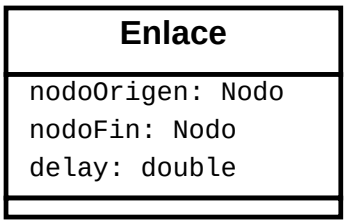

Figura 3.3. Variables de la clase Enlace

Las variables `nodoOrigen` y `nodoFin`, identifican los extremos del enlace. En este simulador, por decisión de diseño los grafos generados son no dirigidos. De todas maneras con pequeños cambios en el simulador perfectamente se puede implementar que los grafos sean dirigidos. Por último está la variable `delay`, que indica el retardo del enlace que interpretaremos como el costo del mismo. Por otra decisión de diseño se tiene en cuenta que cuando se crea un enlace no se generen múltiples aristas entre dos nodos, para que de esta manera sólo pueda haber hasta un arco entre un par de nodos dados.

Al crear el enlace, se cargan en los nodos extremos de éste las siguiente variables: `listaVecinos`, `listaEnlaces` y `distancias`. En el campo `listaVecinos` de la clase nodo, se almacena una lista de todos los nodos que están conectados a éste, mientra que en la variable `distancias` se guarda una lista del *delay* a cada uno

de los respectivos vecinos. Además en la clase nodo se guarda la lista de enlaces que tienen a este como uno de sus extremos, esta lista se almacena en la variable `listaEnlaces`.

Al terminar de introducir estos datos, se completa la descripción física de la topología. Al finalizar se realiza la validación de que el grafo resultante sea conexo para poder asegurar que desde un nodo se puede alcanzar cualquier otro, el cual pudiera contener el archivo buscado.

Además utilizando el resultado descripto en el Capítulo 2, sabemos que siempre se va a recorrer el camino mínimo entre cualquier par de nodos. Para hacer esto se utiliza el algoritmo de Dijkstra, que tiene la ventaja que cada uno de los nodos puede correrlo en paralelo. Al ejecutar este algoritmo se carga en la variable `nextHop`, de la clase nodo, la tabla de ruteo en donde se mapea el nodo destino al que se desea llegar con el próximo salto a dar. La clase que corre este algoritmo es `Topologia`, que tiene conocimiento de todos los nodos y enlaces de la red.

Además en la clase nodo se cargan diferentes datos sobre si allí se almacena una memoria caché, o un *custodian* y otros datos que son referentes al bloque de Aplicaciones, el cual se analizará en la siguiente sección.

3.4.2 Aplicaciones

En este bloque se indican los datos relacionados a los archivos y quiénes serán los encargados de manejar todo lo relacionado a estos. Por lo tanto aquí se definen la cantidad total de archivos que circulan por la red, los datos de los clientes, cachés y *custodians*. Al igual que en el análisis del bloque Topología, nuevamente se utiliza un ejemplo para guiar la explicación, el cual se presenta en la Figura 3.4

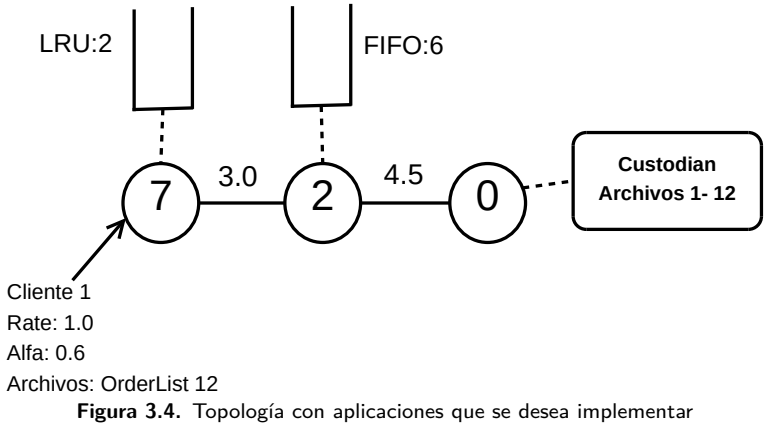

Figura 3.4. Topología con aplicaciones que se desea implementar

Para implementar las aplicaciones presentadas en la figura, se ingresan en el archivo de texto de entrada los siguientes comandos:

```
begin Aplicaciones
listaArchivos 12
Cliente 1 7 1 0.6 OrderList 12
Cache 7 2 LRU
Cache 2 6 FIFO
Custodian 0 RandomList 12
end Aplicaciones
```

Al igual que en el bloque anterior, este empieza y finaliza en `begin` y `end` respectivamente, la única diferencia es que ahora el bloque lleva el nombre Aplicaciones.

El primer comando que se debe ingresar es `listaArchivos`, esto es importante ya que este indica cuantos archivos en total hay en la red. Al ejecutar este comando se genera una lista global de archivos. La clase `Archivo`, que modela a éstos, tiene únicamente dos variables `id` y `tamano`. El simulador permite variar el tamaño de cada uno de los archivos, pero las políticas de remplazo creadas no lo tienen en cuenta. Sin embargo si se generara una política de sustitución que sí tenga en cuenta los tamaños se podrían realizar simulaciones sin necesidad de modificar el simulador. Aunque es una realidad que los archivos pueden ser de distintos tamaños, en la gran mayoría de los casos se asume que estos tienen dimensiones iguales, justificando que en caso contrario se puede dividir los archivos en pedazos más pequeños pero de igual medida, por esta razón el valor por defecto de la variable `tamano` será 1.

En cuanto a la variable `id`, se enumeran los archivos internamente de 0 a $K-1$, siendo K el total de archivos (en el ejemplo $K = 12$). Es importante recordar que esta lista de archivos es única para toda la simulación.

Siguiendo el ejemplo, el siguiente comando de la lista es el encargado de crear un cliente, quien será el encargado de realizar los distintos pedidos por los archivos. En particular podemos observar en la figura 3.5 las variables que modelan el Cliente.

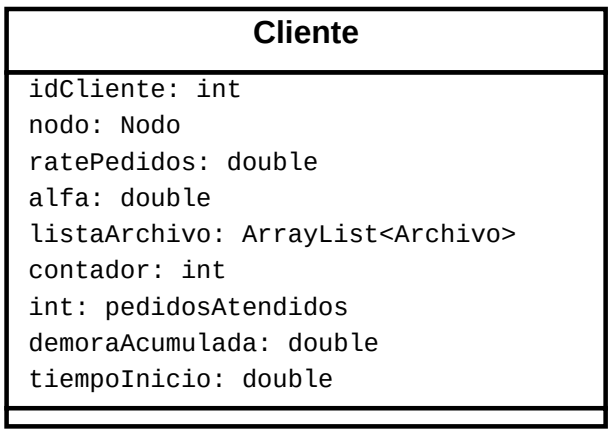

Figura 3.5. Variables de la clase Cliente

La primera variable de la lista es el identificador que utiliza el usuario para

referirse al cliente. Para evitar confusiones nuevamente, no se permite la creación de dos clientes con el mismo ID. El segundo valor del comando, es el identificador del nodo al cual está adosado el cliente. Al tener este identificador se puede cargar la variable `nodo` con el nodo correspondiente, esta es la única variable relacionada a la ubicación física del cliente dentro de la red.

El resto de los datos que se ingresan caracterizan cómo será el flujo de pedidos generado por el cliente. El siguiente dato es el *rate* o tasa de pedidos del cliente, se asumen que los pedidos son independientes y además estos arriban como un proceso de Poisson de intensidad `ratePedidos`, en este caso 1.

En cuanto a qué archivos serán requeridos por el cliente, tenemos los dos últimos datos. Se eligió modelar la popularidad de los contenidos utilizando la distribución $Zipf(\alpha)$ antes discutida, este parámetro α es cargado en la variable `alfa` (en este caso con valor 0,6). Para asignar probabilidades a cada archivo es importante ver en qué orden se considera la popularidad de los mismos. Para definir esto, se encuentra el último dato del comando (`OrderList 12`). Hay tres posibles comandos para indicar cuáles son los archivos y su orden, estos son:

- **OrderList** Este comando tiene una única variable L, que es la cantidad de archivos que se cargarán. En este caso se genera una lista con los archivos desde 0 hasta $L-1$ con los contenidos ordenados. Al estar ordenados el archivo 0 será el más popular, luego el 1 y así sucesivamente.
- **RandomList** Al igual que en el comando anterior, este tiene una única variable L, que es la cantidad de archivos que se cargarán. En este caso se genera una lista con los archivos desde 0 hasta $L-1$ pero el orden de esta es al azar.
- **List** En este comando se debe ingresar primero la cantidad de archivos y luego una lista de estos en orden, por ejemplo `List 3 4 2 8`, pide por 3 archivos, el 4, 2 y 8 en ese orden de popularidad.

Utilizando estos comandos se pueden generar con mayor o menor trabajo todo tipo de pedido con distribución Zipf del parámetro α indicado. El simulador además se asegura que los archivos, por los cuales se realizan los pedidos, efectivamente existen. También al crearse el cliente se carga en la variable `clienteAsociado` del nodo donde se adosó el cliente, esto permite la interacción entre el modelo creado de topología y el cliente. Será a través de este nodo que ingresarán a la red todos los pedidos del cliente. A su vez el cliente creado es agregado a la lista de clientes que mantiene la clase `Topologia`.

Las variables de la clase `Cliente` que no se han discutido hasta ahora son internas al sistema. La variable `contador` lleva la cuenta de la cantidad de pedidos que realizó el cliente. Esta cuenta es útil para verificar que la cantidad de pedidos sea coherente con el *rate* de pedidos y el tiempo de simulación. La variable `pedidosAtendidos`,

como lo dice su nombre cuenta la cantidad de pedidos que fueron atendidos por el cliente. También se cuenta con una variable `demoraAcumulada`, que carga la suma de los *delays* de todos los pedidos atendidos. Utilizando esta variable junto con `pedidosAtendidos`, se puede hallar el *delay* medio de todos los pedidos del cliente. La última de estas variables interna es `tiempoInicio`, en esta variable se carga cuánto tiempo se debe esperar para realizar el próximo pedido.

Ahora procedemos con el comando que ingresa una memoria caché (en el ejemplo se agregan dos) cuyas características se explicarán a continuación. En las dos líneas se mantiene la misma estructura, primero se indica a qué nodo está asociado el caché, luego el tamaño de este y por último el tipo de buffer. Con los datos ingresados en la entrada se pueden cargar las variables de la clase `Cache` las cuales se muestran en la figura 3.6.

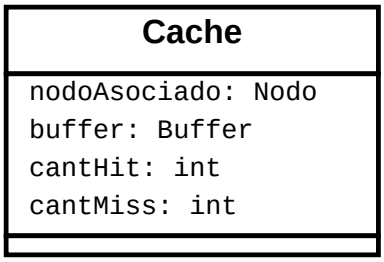

Figura 3.6. Variables de la clase Cache

El primer dato que se completa es el nodo donde se ubica la memoria caché (`nodoAsociado`); al igual que con la clase cliente este dato permite la interacción del elemento, que en este caso manejará los archivos, con la representación de la red física. Al mismo tiempo en la clase Nodo se carga la variable `fileManagerAsociado`, de esta manera no se permite guardar más de un caché o un custodian en ese nodo (una de las hipótesis descriptas anteriormente). En cuanto a las dos variables `cantHit` y `cantMiss`, ambas son inicializadas en 0 y serán utilizadas como contador de estas métricas.

Además se genera el *buffer* asociado a dicha memoria caché. En el simulador se diseñó esta parte a partir de una clase abstracta llamada `Buffer`, de la cual heredan distintos tipos de *buffer*, los cuales son `BufferFIFO`, `BufferLRU` y `BufferRandom`. Únicamente se implementaron estas tres políticas ya que como se destacó antes, son las más utilizadas porque determinan en tiempo constante cuál es el archivo desalojado. En la clase abstracta se presentan únicamente dos variables comunes a todos los *buffers*, `largo` que representa el tamaño del *buffer* y la variable `contenidos`, la cual es una *ArrayList* de `Archivo`. Además se implementa el método abstracto `modificar`, el cual es el encargado de decidir si hay que desalojar o no un archivo del *buffer* y en caso positivo decidir cuál. Esto permite proveer al caché con una interfaz unificada de *buffer* independiente de la política de reemplazo.

A continuación se describen la implementación de cada uno de los tipos de *buffer*, donde se explica la dualidad que se utiliza entre el *ArrayList* y la política de desalojo

deseada.

*Buffer LRU (*Least Recently Used*)*

En este tipo de *buffer*, como lo dice su nombre debe ser desalojado aquel archivo que hace más tiempo no fue requerido, esto sucede únicamente si no hay más lugar en memoria. Se implementa el siguiente pseudocódigo:

```
if(el archivo ya se ubica en memoria){
          se quita el archivo  y se pone al final de la lista
}else{
          if(!estaLleno && quiero almacenar el archivo){
                    se agrega el archivo nuevo al final de la lista
          }else{
                    se quita el archivo al comienzo de la lista
                    se agrega el archivo nuevo al final de la lista
          }
}
```

Al utilizar esta idea, uno se puede asegurar que los archivos estan ordenados en una lista (la cual en *Java* se implementó mediante un *ArrayList*), de manera que el último archivo por el que se realizó un pedido está en la última posición, por lo tanto el archivo a desalojar se encuentra al comienzo de la lista. Es importante destacar que ya se implementa la opción de almacenar o no un archivo, en los algoritmos clásicos normalmente cuando se produce la descarga siempre se almacena el archivo, pero veremos que en las diferentes variantes de algoritmos que se realizaron puede no ser conveniente almacenar siempre los contenidos.

*Buffer FIFO (*First In First Out*)*

Al igual que con el tipo de *buffer* anterior, se presentará el pseudocódigo, para analizar la implementación de esta política de desalojo mediante listas.

```
if(el archivo no se ubica en memoria){
    if(!estaLleno && quiero almacenar el archivo){
           se agrega el archivo nuevo al final de la lista
     }else{
          se quita el archivo al comienzo de la lista
          se agrega el archivo nuevo al final de la lista
     }
}
```

Se observa que el pseudocódigo presenta grandes similitudes que en el caso LRU. La única diferencia que se presenta es que el orden que se debe mantener en los archivos es distinto. En el caso LRU cada vez que se realizaba un pedido por un archivo que se encontraba en memoria, se lo desplazaba al último lugar. Como aquí lo que importa es el momento en el que se ingresó al *buffer* se mantiene el orden de forma natural, no siendo necesario otro movimiento aparte de remover el primero de la lista y agregar el archivo nuevo en el último lugar.

Buffer Random

Por último la implementación de la política de reemplazo *Random*, aquí el archivo desalojado se elige al azar. Esta idea de reemplazo tiene la desventaja de que los archivos no mantienen ningún orden en particular, por lo tanto esto se traduce en que no se almacena un mínimo historial de los pedidos lo cual se puede ver como información implicíta de la popularidad de los mismos. Esta noción de información implicíta será manejada más adelante cuando se discutan las diferentes aproximaciones algorítmicas para resolver el problema planteado en estas redes de distribución de contenidos.

```
if(el archivo no se ubica en memoria){
    if(!estaLleno && quiero almacenar el archivo){
           se agrega el archivo nuevo al final de la lista
     }else{
          se quita un archivo al azar de la lista
          se agrega el archivo nuevo al final de la lista
     }
}
```

Como no es necesario mantener ningún orden específico, al igual que en la política FIFO, se mantiene el orden natural, sólo que ahora se desaloja un archivo seleccionado al azar. Es importante destacar que tanto este sorteo como todos los que se realizarán en la simulación se generan a partir del mismo generador aleatorio para mantener la independencia entre sorteos.

Además de encargarse de los reemplazos, mediante el *buffer* asociado, la clase `Cache`, se encarga de llevar la cuenta de cuántos *hits* y *miss*, se han producido. Esta información será importante más adelante cuando se observe cómo avanzan en el tiempo dichas variables, lo cual permitirá entre otras cosas, medir la tasa media de *hits*, además de poder observar si se presenta algún tipo de comportamiento transitorio en el sistema.

Una vez finalizada la explicación del funcionamiento de los distintos tipos de *buffer*, y por tanto finalizando el análisis de la creación del `Cache`, pasamos al otro

elemento que también hereda de la clase `FileManager`, es decir la clase `Custodian`. Vemos que la estructura del comando de creación es muy sencilla. Primero se pasa el índice del nodo donde se ubicará el *custodian* y luego con la misma estructura que en el comando de cliente, se indica qué archivos serán almacenados de forma estática en dicho *custodian*. Si observamos las variables de la clase `Custodian` tenemos únicamente dos variables que se muestran en la figura 3.7.

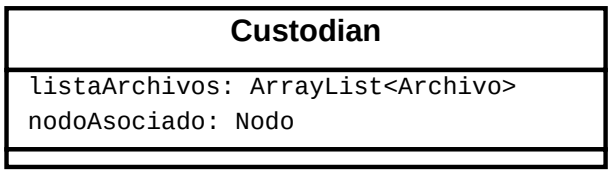

Figura 3.7. Variables de la clase Custodian

Las variables indican con su nombre qué es lo que representan, donde nuevamente `nodoAsociado` es la variable que une el *custodian* a la topología, mientras que `listaArchivos` indica qué archivos son almacenados. Además al crearse el nuevo objeto, este es almacenado en la variable `listaFileManagers` de la clase `Topologia`, al igual que cuando se genera un caché, pero además es almacenado en la variable `listaCustodians` de la misma clase.

El simulador se asegura antes de comenzar que todos los archivos presentes en la simulación estén almacenados al menos por un *custodian*, de esta manera se puede asegurar que siempre se podrá descargar el archivo deseado por el cliente. Además se distribuye por todos los nodos, la información que indica en qué nodo se encuentra el custodio de un archivo dado, en caso de que sea custodiado por varios, se selecciona el *custodian* que se ingresó último. Esta información se almacena en una tabla llamada `destinoArchivo` la cual indica, dado un archivo, dónde se encuentra el nodo con su *custodian*. Al tener estos datos, el ruteo de los pedidos, se hará hacia el nodo donde se custodia el archivo deseado por el camino mínimo descubierto por el algoritmo de Dijkstra antes mencionado.

3.4.3 Simulador

Resta por analizar el último de los bloques de la entrada, el cual será el encargado de indicar los datos sobre la simulación en si. Este bloque se encargará de indicar cuáles serán los algoritmos a utilizar así como el tiempo de simulación. A continuación se puede apreciar la entrada que servirá como guía en el análisis de este último bloque:

```
begin Simulacion
Tiempo 5000
PushAlgorithm 0
```

```
DuplicateCopies 1
PullAlgorithm 0
BarreraInercia 1
end Simulacion
```

El proceso de simulación se dirige desde una clase `Simulador` cuyas variables se muestran en la figura 3.8.

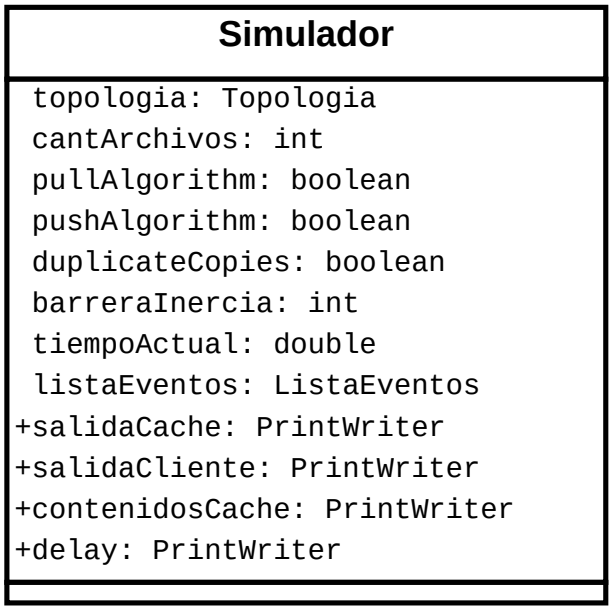

Figura 3.8. Variables de la clase Simulador

Las variables que están marcadas con un signo de más (+), serán analizadas más adelante puesto que corresponden a salida e impresión de datos. Analizando el resto de las variables, se observa que esta clase `Simulador` en primer lugar almacena una topología, la cual a su vez contiene toda la información que fue anteriormente discutida y procesada. Además mantiene la variable `cantArchivos`, que es global a la simulación puesto que la cantidad total de archivos en el universo de la simulación es fija. Luego como su nombre lo indica la variable `tiempoSimulacion`, indica cuánto tiempo durará la simulación, en el ejemplo se observa que se cargó un valor de 5000 el cual se puede asociar a segundos. Luego se destacan las variables booleanas para indicar la configuración de la simulación indicando cuáles algoritmos se implementan y cuáles no. En el comando se indica de forma natural con un valor de 1 si se implementa o 0 en caso contrario. Luego se ingresa el comando `BarreraInercia`, el cual está acompañado por un entero positivo, que será el valor de esta barrera, estos algoritmos serán desarrollados en el próximo capítulo. A forma de ejemplo se utilizó la configuración que emula el algoritmo clásico, en el próximo capítulo se apreciarán ventajas y desventajas de cada uno de estos algoritmos.

Puesto que ya se han ingresado todos los datos, y se ha observado como estos se plasman en el simulador, es momento de analizar la secuencia interna que realiza

el simulador en la ejecución. A partir de aquí entrarán en juego los eventos, parte fundamental en un simulador de eventos discretos como este.

3.4.4 Eventos del simulador

Estos eventos serán los encargados de enviar mensajes entre los distintos elementos del simulador. Estos mensajes son fundamentales en esta arquitectura asíncrona, por lo que se analizarán en detalle en esta sección. Para analizar la generación de eventos, primero se deben conocer cuáles son las variables de la clase `Evento`. Las variables de esta clase aparecen en la figura 3.9

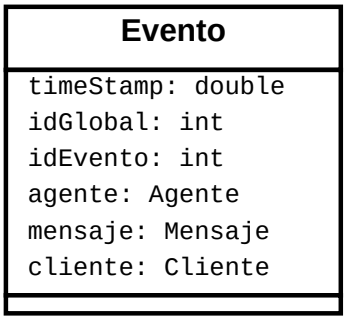

Figura 3.9. Variables de la clase Evento

Se puede apreciar la existencia de un identificador del evento (`idEvento`), este se carga automáticamente con el índice dado por la variable `idGlobal`, la cual lleva la cuenta de cuántos eventos se han creado hasta el momento, de esta manera cada nuevo evento es numerado de 1 a E siendo E la cantidad total de eventos.

Además el evento cuenta con el parámetro `timeStamp`, el cual es fundamental puesto que identifica en qué momento se debe atender el evento. Además la clase `Evento`, implementa la interfaz `Comparable`, utilizando como forma de comparar dos eventos por su variable `timeStamp`, de esta manera cuando se almacenen en una *PriorityQueue*, los eventos a atender estarán ordenados en forma creciente por el tiempo donde tiene que ser atendido. El próximo de los eventos a atender siempre será el que se encuentra en la primera posición de la cola. Este proceso será encapsulado en una nueva clase llamada `ListaDeEventos`, para de esta manera simplificar la comprensión del código para el usuario.

La variable `agente` indicará qué instancia de esta clase debe atender el evento. Recordar que las dos clases que heredan de `Agente`, son `Nodo` y `Cliente`, los cuales en el modelo son quienes se encargan de realizar pedidos o atenderlos. Además se cuenta con la variable `cliente` que indica quién realizó el pedido en primera instancia, de esta forma se puede decidir entre todos los clientes que están conectado al nodo de destino, quién debe atender el evento para finalizarlo.

Finalmente queda por discutir, cómo están estructurados estos mensajes, qué

información contienen y cómo se utiliza esta para el algoritmo de simulación. Se procede a observar las variables de la clase `Mensaje`, en la figura 3.10.

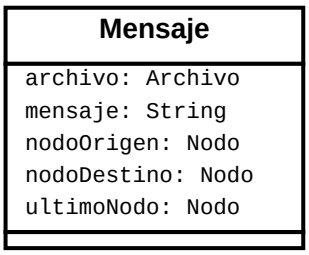

Figura 3.10. Variables de la clase Mensaje

La primera de las variables de la lista, tiene un único objetivo, identificar cuál es el archivo por el qué se está pidiendo originalmente y que se está buscando para descargar. En segundo lugar la variable `mensaje` de tipo *String*, la cual carga información de datos importantes, el primero es el momento en el que se realizó el pedido originalmente y el segundo es el indicador del tipo de evento que se lleva a cabo. El primero de estos dos datos será utilizado para medir la métrica de *delay* antes descripta. El tipo de evento servirá para indicar qué es lo que se debe realizar con este mensaje, a dónde direccionarlo, si debe modificar o no las distintas métricas del sistema. En el algoritmo clásico distinguiremos únicamente dos tipos de evento, `IDA` y `VUELTA`, que identificarán en qué momento de la descarga se encuentra el pedido. A través de los tipos de evento, se implementarán las diferentes variantes de los algoritmos que se describirán en el próximo capítulo.

En cuanto a las últimas variables son las que indican cuál es el nodo que debe alcanzar el mensaje y de dónde proviene éste. Al crearse el mensaje la variable `nodoOrigen` se carga con el nodo asociado al cliente. La variable `nodoDestino` se carga con el nodo donde se encuentra asociado el *custodian* del archivo buscado. Recordar que este nodo se halla de manera sencilla mediante el *HashMap* que mapea el archivo a descargar con el nodo donde se encuentra el *custodian*.

Por último se destaca la necesidad de la variable `ultimoNodo`, que almacena cuál fue el último salto realizado por el mensaje. Esta variable no cumple ninguna finalidad para el algoritmo clásico, pero más adelante en esta documentación se discutirá la necesidad de esta variable para la implementación del algoritmo que decidimos llamar *push/pull*.

Ahora que ya se han descripto todos los componentes de forma completa se puede proceder a comprender el algoritmo de funcionamiento básico del simulador.

3.5 EJECUCIÓN INTERNA

En esta sección se analizará el proceso a través del cual se produce efectivamente la simulación, destacando la creación de eventos y el envío de mensajes entre los distintos componentes de la red. Al igual que en la sección anterior, se utilizará un ejemplo de entrada para enmarcar la explicación y que esta resulte más amena.

Se utiliza como entrada de ejemplo la siguiente:

```
begin Topologia
    Nodo 0
    Nodo 1
    Enlace 1 0 0.05
    Nodo 2
    Enlace 2 1 0.05
    Nodo 3
    Enlace 3 2 0.05
    Nodo 4
    Enlace 4 3 0.05
end Topologia
begin Aplicaciones
    listaArchivos 12
    Custodian 0 RandomList 12
    Cache 1 2 LRU
    Cache 2 2 LRU
    Cache 3 2 LRU
    Cache 4 2 LRU
    Cliente 3 4 1.0 0.8 OrderList 10
end Aplicaciones
begin Simulacion
    Tiempo 5000
    PushAlgorithm 0
    PullAlgorithm 0
    Barrera Inercia 1
    DuplicateCopies 1
end Simulacion
```

Esta entrada genera la red que se muestra en la figura 3.11.

Al cargarse los datos se crean los distintos objetos previamente mencionados y se ejecutan los distintos algoritmos de validación de los datos. Una vez finalizado este proceso comienza la simulación con la inicialización de los clientes que agendan su primer evento, dando lugar a la simulación.

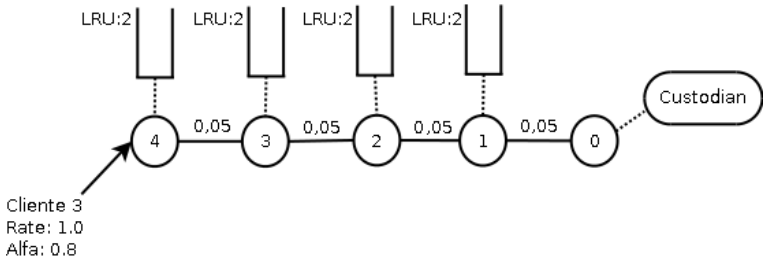

Figura 3.11. Red generada a partir de la entrada dada por el usuario.

Para el estudio de la ejecución iremos paso a paso en el pedido y descarga de un archivo, esto lo haremos con la red introducida en el ejemplo anterior que se encuentra en el estado que se ilustra en la figura 3.12

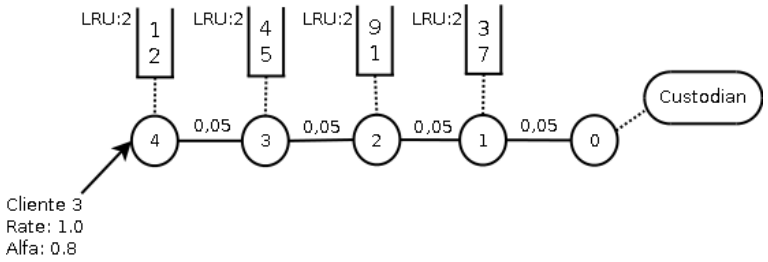

Figura 3.12. Estado actual de la red de distribución de contenidos.

Los pasos que se llevan a cabo en la ejecución son los siguientes:

1. El cliente es informado por el simulador que debe atender un evento. Para ejemplificar se asume que el evento tiene un `timeStamp` de valor de 1.45 y que el pedido es por el archivo número 3. Por esta razón el `nodoOrigen` es el nodo 4 donde se encuentra el cliente y el `nodoDestino` es el nodo 0 donde se encuentra el *custodian* del archivo 3. El cliente al tener un evento del tipo `IDA` (es decir recién se está enviando el pedido) le carga en la variable agente al evento el nodo donde está adosado el cliente. Además se agenda el próximo pedido, para continuar el flujo de arribos, al igual que en el punto anterior, sólo que ahora el `timeStamp` será aún mayor asumamos un valor de 2,5. Ambos eventos se almacenan en la lista de eventos del simulador.

2. Además el cliente realiza un nuevo pedido que implica generar un evento, que tiene como `agente` asociado y `cliente` a él mismo. Se carga como valor de `timeStamp` el tiempo actual del simulador más el resultado de un sorteo exponencial de parámetro el `rate` del cliente. El mensaje se crea sorteando un archivo mediante el generador aleatorio siguiendo la distribución $Zipf(\alpha)$ especificada, y cargando `nodoOrigen` y `nodoDestino` como antes se explicó. También se carga la variable `cliente` del evento con el cliente que está generando este pedido. Además el *string* `mensaje` almacenará el tiempo de inicio (el valor de `timeStamp`) y el tipo de evento `IDA`. Se coloca este evento en la lista de eventos del simulador.

3. El nodo 4 recibe el evento y lo atiende. El atender el evento implica analizar el tipo de evento, que en este caso es del tipo `IDA`. Al ser de este tipo el nodo verifica si el archivo buscado (número 3) se encuentra en el caché asociado. En nuestro caso el archivo no se encuentra por lo tanto aumenta en 1 la cantidad de *miss* en este caché. Luego de esto redirecciona el evento, es decir lo envía al siguiente nodo en el camino mínimo al nodo 0 (Nodo 3), y además le suma al `timeStamp` del evento el *delay* del enlace. Por lo tanto el nuevo valor de esta variable será $1,45 + 0,05 = 1,50$. El mensaje se mantiene intacto. Este evento se coloca en la lista de eventos del simulador.

4. El nodo 3 recibe el evento del tipo `IDA`. Al igual que el nodo 4, aumenta el contador de *miss* de su correspondiente caché puesto que el no almacena el archivo 3. Redirecciona el evento modificando la variable `timeStamp` la cual ahora toma un valor de 1,55. Carga en el evento que el nuevo agente a atender este evento es el nodo 2, manteniendo el resto de las variables sin modificar. Coloca este nuevo evento en la lista de eventos del simulador.

5. En el nodo 2 al igual que en los dos nodos anteriores aumenta la cantidad de *miss* del caché y le asigna como nuevo `timeStamp` el valor de 1,60. Mantiene el resto de las variables intactas y se agrega el evento a la lista del simulador.

6. El nodo 1 efectivamente tiene almacenado el archivo, así que aumenta en uno la cantidad de *hits* del caché correspondiente. Además cambia el tipo de evento indicando ahora que es del tipo `VUELTA`. También modifica las variables `nodoOrigen` y `nodoDestino`, indicando como el nodo de origen él mismo mientras que el nodo de destino es el antiguo nodo origen, ya que ahora el mensaje se envía en el sentido opuesto. Algo que es importante es que todavía no se produce el cambio en el *buffer*, eso sucede únicamente cuando el evento es de tipo `VUELTA`, por esta razón se genera un evento cuyo agente es el mismo nodo 1. Este evento tiene el mismo `timeStamp` y se almacena en la lista de eventos del simulador.

7. El evento al ser del tipo `VUELTA`, no modifica los contadores del *buffer*, sin embargo modifica los contenidos de este, en nuestro caso modifica únicamente el orden, (en el caso de que el *buffer* hubiera sido del tipo FIFO no hubieran habido cambios). Al finalizar esto se redirecciona el evento por lo que el nuevo `timeStamp` será 1,65 y el próximo agente en atender el evento será el nodo 2. Se coloca el evento en la lista del simulador al igual que en los caso anteriores.

8. El nodo 2 recibe el evento que es del tipo `VUELTA` y por esta razón modifica el *buffer* desalojando el archivo 9, almacenando únicamente el 1 y el 3. Nuevamente se producen cambios en el evento más que en el agente y en el `timeStamp` (ahora 1,70). Nuevamente se agrega a la lista.

9. El nodo 3 al no ser el nodo de destino modifica el *buffer* al igual que el nodo 2, en este caso desalojando el archivo 4. Cambia la variable `timeStamp` (ahora 1,75) y que el agente es el nodo 4.

10. El nodo 4 modifica su *buffer* puesto que es un mensaje del tipo `VUELTA`, desalojando el archivo 1. Como es el nodo de destino, el próximo agente es su cliente almacenado en la variable `cliente` del evento. Este evento es almacenado en la lista con este nuevo agente.

11. El cliente recibe el evento y puesto que ahora es del tipo `VUELTA`, y a partir del `timeStamp`, y del tiempo almacenado en el *String* `mensaje`, puede calcular el *delay* de la descarga del archivo, en este caso $1,75 - 1,45 = 0,30$.

12. Finalmente el simulador atenderá el evento creado en el punto 2 que tenía `timeStamp` de valor 2,5 y comenzará todo el proceso nuevamente. Esto continuará hasta que este valor sea mayor que el tiempo de simulación, en ese caso finalizará la simulación.

Este proceso también se puede apreciar en el diagrama de secuencia de la figura 3.13

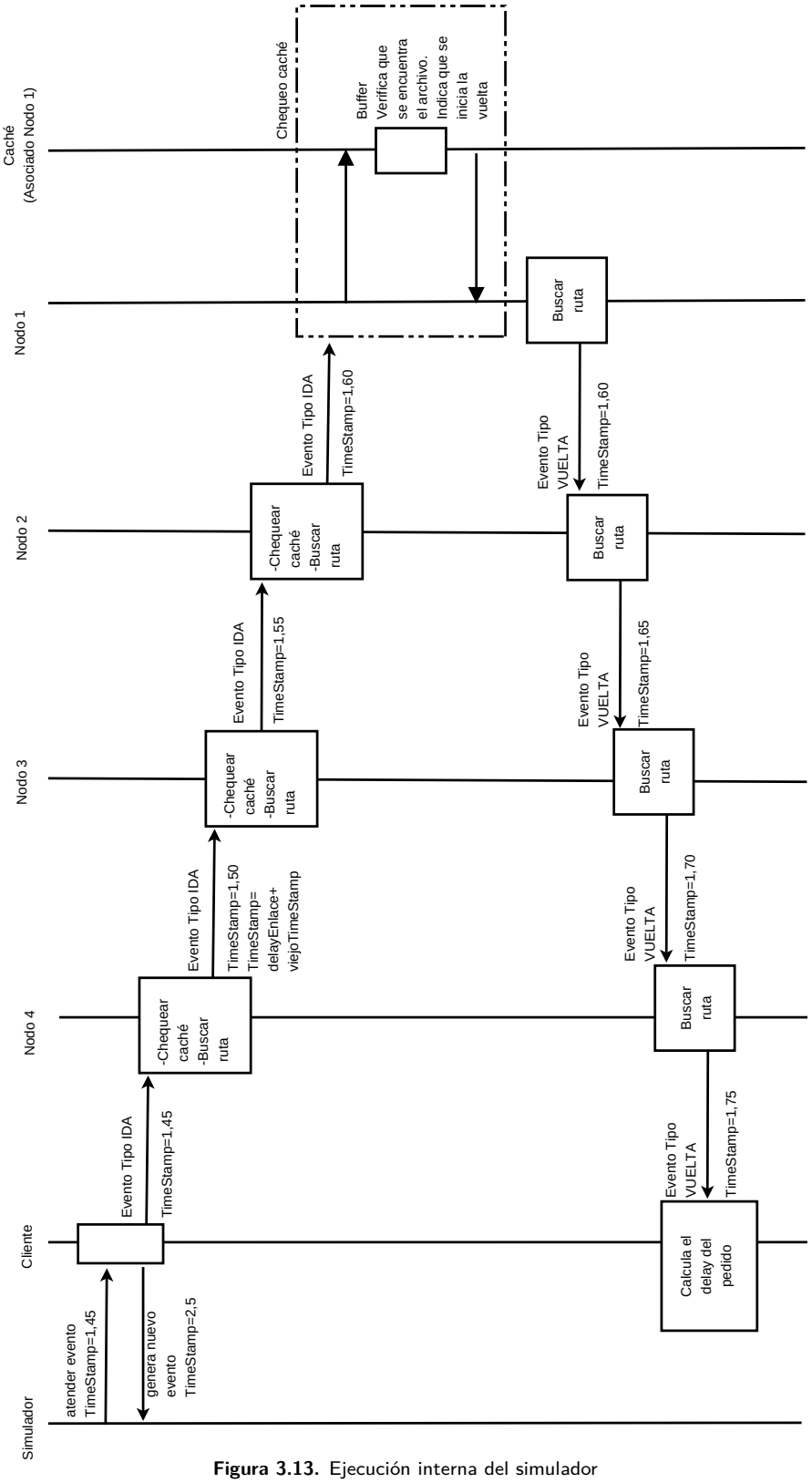

Figura 3.13. Ejecución interna del simulador

3.6 GENERADOR ALEATORIO

En el simulador se implementa una clase llamada `GeneradorAleatorio` la cual utiliza una única variable aleatoria de base, de esta manera todas la variables aleatorias generadas serán sorteadas independientemente. En esta clase se implementan los sorteos exponenciales para determinar los tiempos de los distintos pedidos de los clientes. El sorteo se realiza utilizando el método de la función inversa. Además se creó un método el cual es utilizado para realizar el sorteo del archivo que se desea descargar, en este caso se implementa la distribución $Zipf(\alpha)$.

3.7 SALIDA

En esta sección se verá en detalle la salida del simulador, donde se describirá cuál es la idea de cada uno de los componentes de este.

El simulador imprime 4 archivos en la misma dirección que la entrada con los siguientes nombres:

- `Cliente`
- `Delay`
- `SalidaCache`
- `ContenidosCache`

Analizaremos a continuación cada una de las salidas comenzando por la llamada `Cliente`. En esta salida se imprime una linea por cada pedido resuelto, es claro que si la cantidad de pedidos total es grande la diferencia entre la cantidad de pedidos realizados y resueltos, en relación al total es despreciable. Por esta razón estas estadísticas se pueden tratar también como la cantidad de pedidos realizados.

En cuanto a los datos que se imprimen son, en primer lugar el identificador que utiliza el usuario para el cliente que realizó el pedido. En segundo lugar se imprime el archivo que se pide, con estos dos datos, se puede observar si la distribución de pedidos por archivo es la correcta. Además se agrega el *delay* que se tuvo al atender el pedido, y por último se agrega un contador de pedidos realizados por el cliente, este número sirve para saber si el *rate* de pedidos simulados es el esperado.

La salida `Delay`, presenta un resumen del *delay* promedio que tuvo cada uno de los clientes para que se resolvieran sus pedidos y el *delay* promedio de todo el sistema, esto será muy práctico puesto que será la métrica que será más utilizada para comparar los distintos algoritmos.

En la salida `SalidaCache`, se tiene la información sobre el avance temporal de la cantidad de *hits* y *miss* en los distintos caché del sistema. En la salida se almacena un dato cada vez que se produce un evento cuyo agente es un caché, primero se almacena el nodo donde se encuentra el caché, luego el tiempo y después de esto dos contadores, uno con la cantidad de *hits* y el otro con la cantidad de *miss*. A su vez al final de la linea se guarda el número del archivo que generó el cambio en alguno de los contadores. Mediante estos datos se puede observar si existe algún comportamiento transitorio en el sistema y ver cuál es la tasa media de *hits* de cada uno de los caché.

Finalmente la salida `ContenidosCache`, es la que permitirá indicar cuánto tiempo en promedio se almacena cada uno de los archivos en una memoria caché. En este caso, al igual que en la salida anterior se guarda una línea cada vez que un evento es atendido por un agente del tipo `Cache`. En esta salida se guarda en primer lugar el tiempo actual del simulador, seguido por el `idUsuario` del nodo que atiende el evento. Luego se carga una tira de largo el número total de archivos. Esta tira está compuesta por 0's y 1's, indicando si cada uno de los archivos se encuentra o no almacenado en ese caché. Si la tira fuera por ejemplo 1 0 1 0 0 0, significa que la cantidad total de archivos es 6 y que en ese nodo están almacenados los archivos 0 y 2.

Mediante estas salidas es posible medir todas las métricas discutidas, y de esta manera poder discernir las ventajas y desventajas de cada uno de los algoritmos.

3.8 PRUEBAS DEL SIMULADOR

Puesto que ya se finalizó con el análisis y justificación tanto del funcionamiento del simulador como de su arquitectura, pasaremos a la verificación de su correcto funcionamiento. En este capítulo se mostrarán algunas de las pruebas de funcionamiento que se llevaron a cabo. Estas pruebas son de un carácter más macro que las pruebas unitarias, puesto que se cuenta con una gran cantidad de sorteos y una fuerte componente de aleatoriedad y estas abarcan un gran espectro de posibilidades. Para este análisis se utilizará la misma entrada que se utilizó para mostrar el paso a paso de la simulación. Por lo tanto tendremos la misma topología, aplicaciones y características del simulador.

Se comienza por verificar que los sorteos realizados tengan la distribución de probabilidad deseada y que además la cantidad de pedidos totales se corresponda con la relación dada por el tiempo total de simulación y el *rate* de pedidos. En la figura 3.14 podemos observar una gráfica con la distribución de los pedidos realizados por el cliente.

Además se contabilizó la cantidad total de pedidos, la cual fue de 5094, por lo que se verificó la relación esperada. Luego se realizó la simulación descripta en la

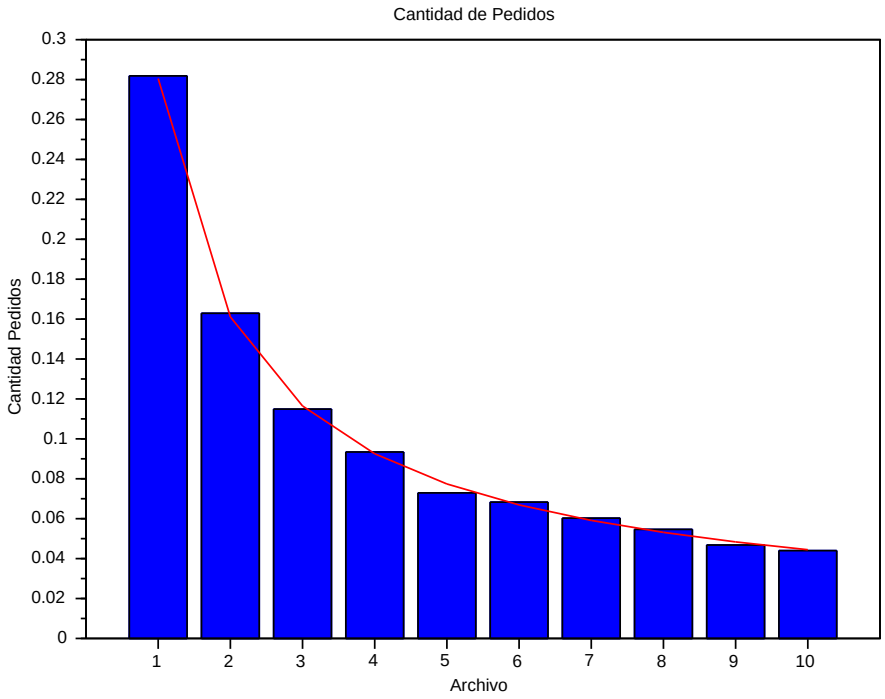

Figura 3.14. Distribución normalizada de pedidos, comparada contra la Zipf correspondiente.

sección anterior, y se imprimió el estado de los *buffers* en el momento que se saca cada uno de los eventos para ser atendidos obteniendo el siguiente resultado:

```
El cliente 3 pidio el archivo 8
C -  -  -  -
C -  -  -  -
C -  -  -  -
C -  -  -  -
C -  -  -  -
C -  -  -  -
C -  -  -  -
C -  -  -  -
C - 8  -  -  -
C - 8  - 8  -  -
C - 8  - 8  - 8  -
C - 8  - 8  - 8  - 8
El cliente 3 pidio el archivo 0
C - 8  - 8  - 8  - 8
C - 8  - 8  - 8  - 8
C - 8  - 8  - 8  - 8
C - 8  - 8  - 8  - 8
C - 8  - 8  - 8  - 8
```

```
C - 8  - 8  - 8  - 8
C - 8  - 8  - 8  - 8
C - 8 0  - 8  - 8  - 8
C - 8 0  - 8 0  - 8  - 8
C - 8 0  - 8 0  - 8 0  - 8
C - 8 0  - 8 0  - 8 0  - 8 0
El cliente 3 pidio el archivo 6
C - 8 0  - 8 0  - 8 0  - 8 0
C - 8 0  - 8 0  - 8 0  - 8 0
C - 8 0  - 8 0  - 8 0  - 8 0
C - 8 0  - 8 0  - 8 0  - 8 0
C - 8 0  - 8 0  - 8 0  - 8 0
C - 8 0  - 8 0  - 8 0  - 8 0
C - 8 0  - 8 0  - 8 0  - 8 0
C - 0 6  - 8 0  - 8 0  - 8 0
C - 0 6  - 0 6  - 8 0  - 8 0
C - 0 6  - 0 6  - 0 6  - 8 0
C - 0 6  - 0 6  - 0 6  - 0 6
```

Se puede observar como avanza el pedido en búsqueda de quién lo tenga, los distintos *buffers* se mantienen intactos. Mientras que a la vuelta, se observa la modificación de los contenidos en las distintas memorias caché.

Además se realizaron varias pruebas en distintas topologías, así como en el ingreso de los datos, para observar que las verificaciones esperadas, sean realizadas.

Capítulo 4

Algoritmos

En este capítulo se introducirán los diferentes algoritmos que se proponen como solución al problema de distribución de archivos en la red. Se analizará cuáles son las deficiencias de las diferentes estrategias y se mostrarán las posibles soluciones a dichos problemas.

Para realizar el análisis de los diferentes algoritmos se seguirán los siguientes pasos:

- Reconocer las deficiencias del último algoritmo presentado.
- Descripción de la posible solución y cuál es la idea detrás de la misma.
- Cómo se implementó la solución en el simulador.
- Analizar distintos ejemplos para verificar que efectivamente el cambio realizado cumple el objetivo.

4.1 ALGORITMO CLÁSICO

Utilizaremos como algoritmo clásico el que fue definido en [3] y utilizaremos este como punto de referencia para comparar frente a las nuevas soluciones.

4.1.1 Descripción del algoritmo

Este algoritmo fue descripto anteriormente en el Capítulo 3 cuando se realizó el análisis paso a paso de la ejecución de la simulación. A forma de repaso, los distintos clientes generan pedidos de descarga de archivos en los distintos nodos de la red.

Cada uno de estos pedidos son atendidos de la misma manera. En cada uno de los nodos a donde llega el pedido en primera instancia, éste verifica si tiene almacenado el archivo buscado. En caso de que el archivo no se encuentre en el caché que es consultado, mediante una tabla de ruteo fija, se indica cuál es el próximo nodo a ser consultado. En caso de tener el archivo se genera la descarga. La descarga lleva el archivo hallado por el mismo camino que se recorrió buscándolo hasta llegar al nodo donde el cliente realizó el pedido en un comienzo. En este algoritmo en particular, cuando se realiza la descarga, el archivo es almacenado en cada uno de los nodos del camino de vuelta. A su vez dependiendo de la política de desalojo de cada nodo se decidirá qué archivo se eliminará del *buffer*.

4.1.2 Implementación en el código

Este algoritmo se implementó utilizando únicamente dos tipos de eventos, `IDA` y `VUELTA`. El mecanismo de funcionamiento fue explicitado al momento de mostrar el algoritmo de simulación en el Capítulo 3. A modo de resumen, se diferencia la búsqueda del nodo que tenga almacenado en su memoria caché el archivo y la descarga del mismo. Es importante diferenciar ambos momentos, puesto que en la búsqueda del archivo, se deben contabilizar las métricas como *miss* y *hits* en los cachés mientras que los contenidos en éstos no deben variar. Sin embargo al momento de la descarga se deben almacenar los archivos en cada uno de los nodos intermedios pero no se altera ninguna métrica. Estas dos situaciones son las que se distinguen con cada tipo de evento para que sea claro el funcionamiento para el usuario del simulador.

4.1.3 Pruebas de performance

Se utilizarán distintas pruebas midiendo las métricas que se especificaron sobre todo el *delay*. Además se analizarán dos tipos de topología en particular. La primera es una línea donde en un extremo se encuentra el cliente que realiza los pedidos, mientras que en el otro se encuentra el *custodian*. La otra topología es un árbol de un único nivel con varias hojas, cada una con un cliente. En la raíz del árbol se encuentra el *custodian*. Se entiende que estas dos topologías son los bloques base de construcción de cualquier topología, puesto que una da sentido de profundidad (la línea), mientras que la otra da sentido de amplitud. En las siguientes pruebas se mostrará la topología donde se encuentren las principales falencias de cada algoritmo.

En este caso en particular se observará el comportamiento de algoritmo en la topología de profundidad (Figura 4.1) puesto es donde se hacen más visibles las carencias. En esta topología consideramos 4 nodos unidos por enlaces de 50 ms de retardo. En un extremo de la línea se encuentra el cliente, quien realizará pedidos con una tasa de un pedido por segundo y pedirá por 100 archivos distintos. Se tiene

que el archivo más popular será el número 0, mientras que el menos popular será el 99, y consideramos que la distribución Zipf tiene un α de valor $0,8$. En el otro extremo de la línea se encontrará el *custodian* de todos los archivos, mientras que en cada nodo se coloca una memoria caché de tamaño 10, con política de reemplazo LRU.

En este escenario la distribución óptima sería almacenar los 10 archivos más populares (los archivos del 0 al 9), en el caché del nodo 4. Luego los siguientes 10 archivos más populares en el nodo 3 y continuar de esa manera con los restantes caché.

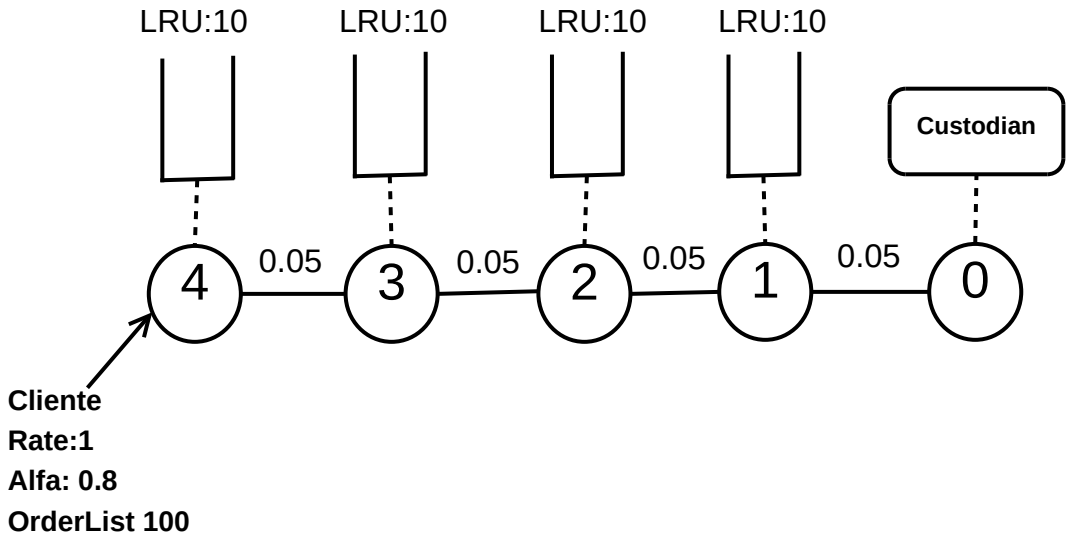

Figura 4.1. Topología en profundidad

A continuación se observa la evolución temporal de la cantidad de *hits*, solamente se grafica (Figura 4.2) para el nodo 4 que es el que concentra la mayoría de los *hits*, de todas maneras se verificó en el simulador que la cantidad de *hits* en los restantes nodos es muy baja.

En particular si se asume la hipótesis de ZDD (*Zero Download Delay*), en esta topología, pero con todos los *buffers* de tamaño 1, se produce el fenómeno de que en las distintas memorias caché, siempre está almacenado el mismo archivo. Esto se debe a que al almacenar el nuevo archivo siempre se desecha el anterior puesto que los *buffers* tienen tamaño 1. Como en el algoritmo clásico sucede que en la descarga el archivo se almacena en todos los nodos, todos los caché guardan este archivo. De todas maneras los problemas presentes no son tan extremos como en el caso que la capacidad de almacenamiento es 1, pero siguen siendo graves.

Si ahora se observan las estadísticas que indican cuánto tiempo se almacenó cada uno de los archivos, por el fenómeno antes explicado vemos que esta gráfica se repite en todos los nodos (Figura 4.3), con diferencias mínimas puesto que no se almacenan los archivos exactamente en el mismo momento. Además vemos que esta gráfica mediante una normalización adecuada concuerda con la distribución de

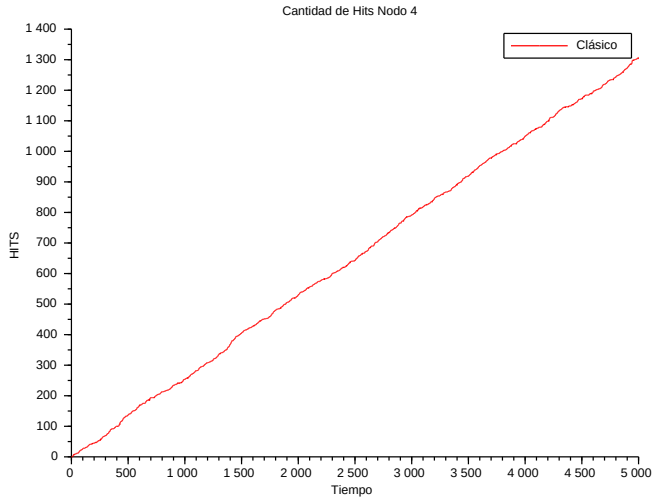

Figura 4.2. Gráfica que indica el avance temporal de los *hits* en el caché 4.

pedidos del cliente.

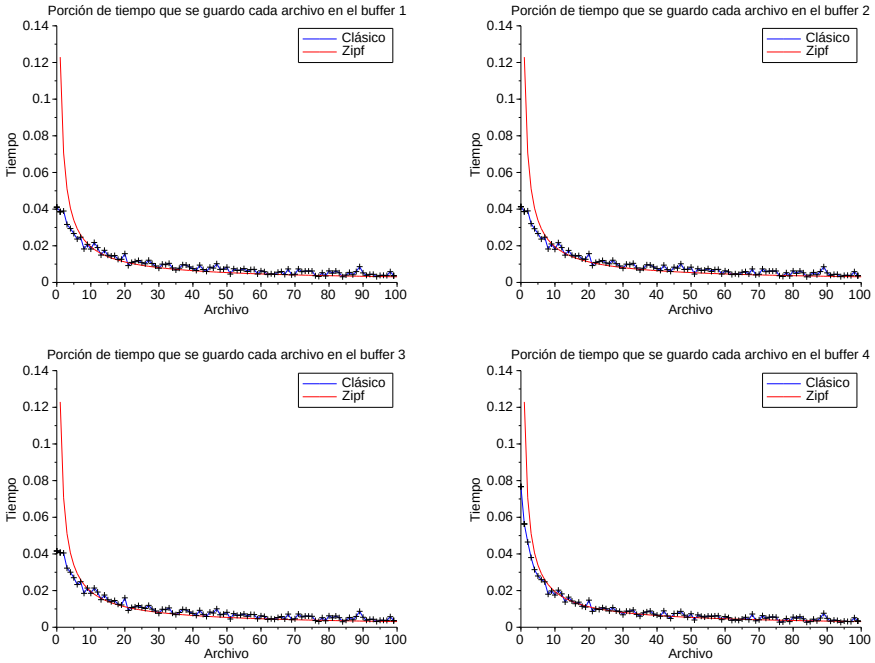

Figura 4.3. Distribución Temporal de los contenidos en cada uno de los nodos, en comparación con la ley Zipf de popularidad

Efectivamente en la figura 4.3 se puede observar que todos los nodos tienen la misma distribución de archivos. Además al comparar con el perfil de pedidos por archivo del cliente, se puede notar que la curva de distribución acompaña a la distribución de popularidades de los archivos. Es importante recordar lo visto en la introducción que las popularidades de los archivos están dadas por la distribución $Zipf(\alpha)$ la cual asigna una popularidad proporcional a $\frac{1}{i^{\alpha}}$ al archivo en la $i-$ésima

posición. Se observa que se está lejos de una distribución ideal, donde el nodo más cercano al cliente debería almacenar los archivos más populares, el siguiente nodo los segundos más populares y así sucesivamente, pero es claro que las distribuciones obtenidas no son las deseadas.

Por último se midió el *delay* total obteniendosé el valor de `DELAY MEDIO TOTAL 0.33523` recordar que esta medida se obtiene al realizar el promedio ponderado (por el *rate*) del *delay* medio de los clientes.

4.1.4 Problemas detectados

Es claro que la repetición del mismo archivo en todos los nodos es uno de los grandes problemas de este algoritmo. Si un archivo se tiene que ir a buscar al *custodian* éste será almacenado en todos los nodos del camino, cuando quizás sea uno de los archivos menos solicitados. Algo que tampoco es bueno de este algoritmo, es que los archivos desalojados son desechados por el sistema. El archivo desalojado tiene cierta información implícita; aunque no es lo suficientemente popular para ser almacenado en ese momento, era lo suficientemente popular para ser almacenado hasta hace un instante. Por lo tanto si el archivo tenía cierta popularidad, es importante no eliminarlo sino guardarlo en algún lugar cercano en la topología. A partir de estos dos conceptos surgen las ideas para los siguientes algoritmos.

4.2 ALGORITMO PUSH

4.2.1 Descripción del algoritmo

A este nuevo algoritmo lo llamaremos *push*, puesto que empujaremos los archivos, es decir que enviaremos a éstos en dirección al *custodian*. En este algoritmo lo que se hará es empujar los archivos desalojados en cada nodo.

Al hacer esto, en primer lugar, se mantiene cerca el archivo desalojado, ya que se envía al próximo nodo y se almacena ahí. También se genera el efecto de aumentar la discrepancia entre los contenidos de un nodo y el siguiente, ya que este archivo desalojado es seguro que será diferente entre ambos nodos. Esto se debe a que no está en el nodo originario puesto que fue desalojado y es almacenado en el siguiente nodo.

4.2.2 Implementación en el código

No es posible implementar este algoritmo únicamente con los tipos de eventos que se presentaron hasta ahora. Por esta razón se implementa un nuevo tipo de evento

denominado `PUSH`. Este tipo de evento tiene básicamente el mismo comportamiento que el tipo de evento `VUELTA`. Al recibir un evento `PUSH`, el nodo realiza los cambios correspondientes en su memoria caché, pero el archivo desalojado, se rutea con la misma tabla de ruteo antes utilizada. Es importante destacar que el algoritmo siempre finaliza ya que siempre o bien se alcanza el *custodian* o el archivo que es empujado se encuentra en la siguiente caché, por lo que no desaloja un nuevo archivo. El resto de los eventos se siguen manejando de la misma manera. Por último para activar este algoritmo se debe colocar el comando `PushAlgorithm 1` en la archivo de texto que ingresa al simulador.

4.2.3 Pruebas de performance

Al igual que el algoritmo clásico se estudiará en principio la topología en profundidad para ver si se pudieron realizar mejoras en este caso.

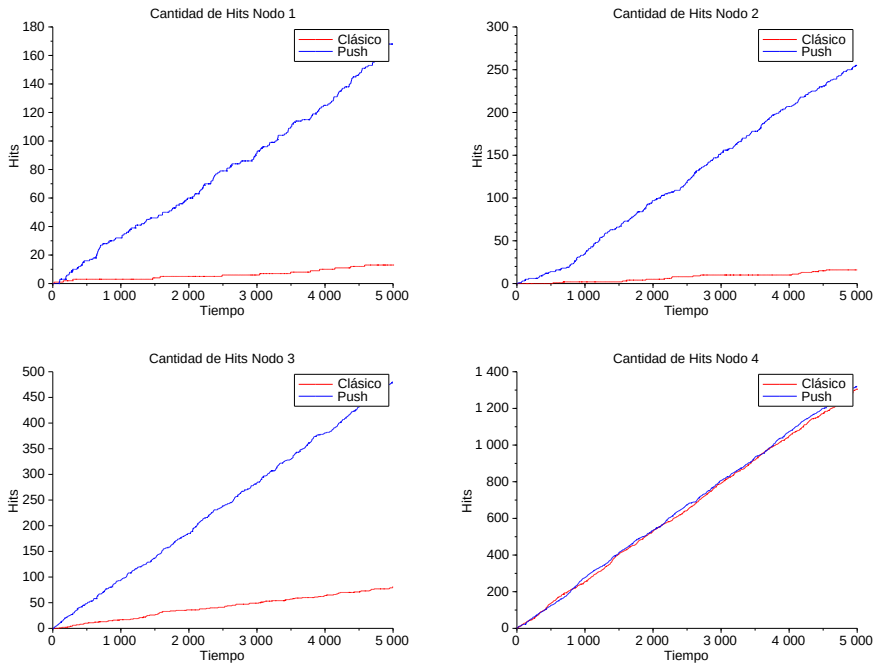

Figura 4.4. Avance temporal de los *hits* producidos en cada nodo

Observamos en primer lugar, como es de esperarse, que en el nodo 4 la diferencia es casi nula puesto que el algoritmo de *push* no afecta los contenidos en éste nodo. Sin embargo observamos que en los demás nodos, hay una gran diferencia en la proporción de *hits* que se producen, siendo una importante mejora de este algoritmo respecto al anterior. Ésta es una de las razones por las que si comparamos el *delay* medio de ambos algoritmos en las mismas condiciones obtenemos `DELAY MEDIO TOTAL 0.25136`, el cuál es significativamente menor que el obtenido aplicando el algoritmo clásico (`DELAY MEDIO TOTAL 0.33523`).

Cuando se observa si se logró aumentar la diferencia entre cuáles son los archivos

almacenados en cada uno de los nodos, se obtuvieron las gráficas de la figura 4.5

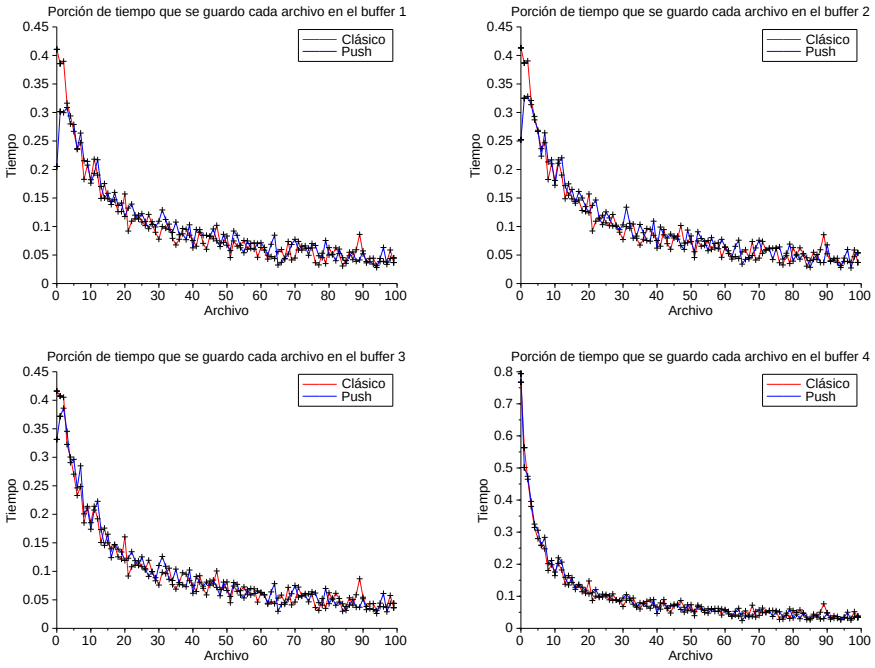

Figura 4.5. Distribución temporal de los archivos almacenados en cada nodo

No se ven grandes diferencias entre lo que se almacena en promedio en cada uno de los nodos. Sin embargo estas gráficas reflejan el promedio temporal y no muestran la discrepancia de contenidos entre nodos.

4.2.4 Problemas detectados

El problema que se observa es que la distribución de tiempo de ocupación de cada archivo en los nodos, al igual que en el algoritmo clásico no se aproxima a la distribución óptima. Esto puede deberse a que se sigue almacenando el mismo archivo en todos los nodos, por lo que se buscará una variante a este algoritmo que evite problema.

4.3 ALGORITMO PUSH + NO DUPLICATE COPIES

4.3.1 Descripción del algoritmo

Analizando los problemas de la implementación anterior es claro que se debe aumentar la discrepancia de los archivos que se almacenan en cada uno de los nodos de la línea. A partir de este problema, surge la idea de no guardar los archivos en todos los nodos, sino que únicamente se guarde en el nodo donde el cliente realizó

el pedido. Si no existiera la idea de *push* el guardar solamente en el nodo donde ser realizan los pedidos no tendría sentido. Esto se debe a que únicamente se utilizaría la primera memoria caché. Sin embargo ahora al empujar los archivos desalojados, se irán completando el resto de las memorias a través de la red.

Al evitar guardar el mismo archivo en todos los nodos, estamos aumentando la diferencia de contenidos entre nodos consecutivos. Como ya se ha discutido al aumentar esta diferencia de contenidos estamos aumentando la probabilidad que un contenido no deba ser buscado hasta el *custodian*.

4.3.2 Implementación en el código

Para implementar este algoritmo no se deben hacer grandes cambios en el simulador. Esto se debe a que en gran parte el funcionamiento de este algoritmo es muy similar al anterior. Para implementar este cambio únicamente se le deben anunciar a los *buffers* que deben almacenar el archivo, si se encuentran en el nodo donde se realizó el pedido. Para hacer esto se implementa en los algoritmos de sustitución de los *buffers* que reciban como variable el guardar o no el archivo que reciben.

Una vez implementado ese cambio en los algoritmos de desalojo de cada uno de los *buffers*, el nodo únicamente debe informarle a la memoria caché, que se encuentra en el final del camino buscado, por lo tanto se debe guardar el archivo.

4.3.3 Pruebas de performance

Primero se analizará la cantidad de *hits* en cada uno de los nodos para comprobar efectivamente si desapareció de forma completa el problema de la repetición innecesaria de contenidos. En la figura 4.6 podemos ver la comparación de la evolución temporal de la cantidad de *hits* en cada nodo.

Nuevamente, se observa una mejora en la cantidad de *hits* en cada uno de los nodos del sistema, a excepción del nodo donde se ubica el cliente, lo cual era esperable puesto que este nodo no cambia su comportamiento frente a los pedidos.

Algo que llama la atención es que al apreciar la distribución de tiempo en que los contenidos son almacenados en los distintos nodos, no se separa demasiado de los algoritmos anteriores, como se puede observar en las gráficas de la figura 4.7.

Esto lleva a la conclusión de que se debe implementar otra solución para alcanzar la distribución óptima en una línea. Aunque esta distribución óptima no fue alcanzada se observa que la métrica de *delay* nuevamente ha mejorado, en este caso obteniendo `DELAY MEDIO TOTAL 0.22431`. Nuevamente observamos una mejora, esta vez más leve, con respecto al algoritmo de *push*.

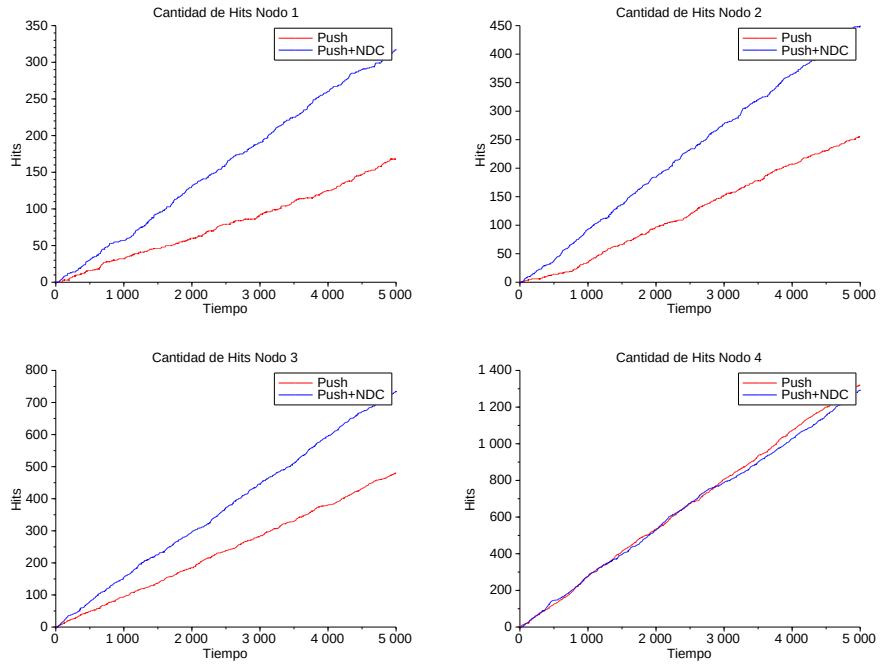

Figura 4.6. Avance temporal de los *hits* producidos en cada nodo

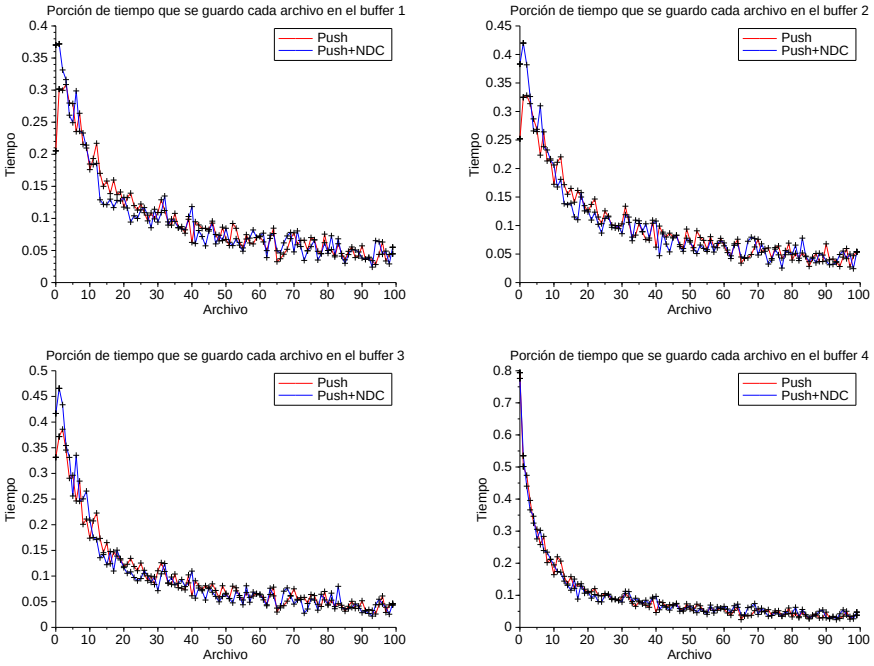

Figura 4.7. Distribución temporal de los archivos almacenados en cada nodo

4.3.4 Problemas detectados

El principal problema que se tiene con este algoritmo es que no se logra una disparidad total de contenidos, ya que siguen habiendo distintas combinaciones de pedidos que generan duplicados en la línea. Además no se logra que el tiempo que

se almacenan los contenidos en cada uno de los nodos se acerque a la distribución óptima.

4.4 ALGORITMO PUSH/PULL

4.4.1 Descripción del algoritmo

El principal problema a resolver en esta instancia es aumentar la disparidad de contenidos en la topología en profundidad, para de esta manera optimizar el uso de la memoria en la red. Como se ha analizado anteriormente un problema latente es que el archivo se guarde en varios nodos, o que haya una secuencia de pedidos que tenga como resultado que este sea almacenado repetidas veces en la red.

Cuando se analizó en profundidad el algoritmo anterior, se observó que una posible falla es que el archivo que era pedido se guardaba automáticamente en el primer caché lo cual puede no ser bueno. Por ejemplo en el caso de la topología de línea, si se realiza un pedido por el archivo menos popular, dado el algoritmo anterior este archivo se guardaría lo más cerca posible del cliente. Dados los análisis que se realizaron previamente no es de conveniencia gastar un lugar de la memoria caché más cercana al cliente en un archivo que casi no será solicitado.

A partir de este problema en concreto, surge la idea del algoritmo que denominaremos *push/pull* o de forma abreviada solamente *pull*. Se elige ese nombre porque vamos a "tirar" de los archivos para acercarlos al cliente. La idea se basa en aprovechar las rachas de pedidos por un mismo archivo, las cuales serán más comunes para los archivos más populares. Básicamente lo que hace el algoritmo es ir acercando los archivos al cliente, un nodo por vez. De esta manera cuánto más requerido sea un archivo, más rápido se acercará al cliente. También es importante destacar que se mantiene la idea del algoritmo anterior, de "empujar", los archivos que son desalojados. Al usar este algoritmo, se tiene en contrapartida, que si un archivo no es suficientemente popular, será alejado del cliente.

Finalmente, lo que es de destacar de este algoritmo, que si se aplica en la topología de profundidad, se evita que haya múltiples copias del mismo archivo en la red. Esto se debe a que una única copia viaja por la línea, ya que al no haber bifurcaciones en la topología, esta copia se puede mover únicamente hacia adelante o hacia atrás.

4.4.2 Implementación en el código

La implementación de este algoritmo no es directa sobre la base planteada por los otros algoritmos. Para poder implementar el algoritmo de *pull*, fue necesaria la

creación de dos tipos de eventos nuevos, llamados `PULLFATHER` y `PULLSON`. Cada uno indica, en qué etapa del *pull* se encuentra el archivo, si en el primer nodo donde se desaloja el archivo (el que llamaremos padre o *father*) o el nodo que recibe ese archivo (hijo o *son*). Una opción para indicar en qué situación se encuentra podría haber sido una *flag*, pero se entiende que por razones de modularidad del código, la solución que utiliza dos tipos de evento es la mejor opción.

La secuencia de funcionamiento, en gran parte, se mantiene igual. Al comienzo, el pedido viaja mediante eventos del tipo `IDA`, a través de la red hasta alcanzar el nodo que tiene almacenado el archivo. Allí se produce un *hit*. En el momento que se halla el archivo, el tipo de evento se convierte en `PULLFATHER`, puesto que ese nodo será el padre que le dará a su hijo el archivo. El mismo nodo donde se produjo el *hit*, atenderá el evento `PULLFATHER`. Al hacer esto modificará el *buffer* eliminando el archivo de la lista de contenidos. El único caso donde el evento `PULLFATHER` no elimina un archivo de la lista de contenidos es cuando el nodo que atiende el evento es el nodo donde está el cliente que realizó el pedido. Esto se debe a que ya no se puede acercar más el contenido al usuario, entonces no se debe eliminar. En el caso que se pase un archivo al hijo, se envía un evento del tipo `PULLSON`, para que el hijo lo almacene. Una vez que se almacenó el archivo en la memoria caché correspondiente, se genera el evento de tipo `VUELTA` y se procede de la misma manera que antes. A su vez se mantiene el algoritmo que genera los *push* a través del sistema.

Para activar este algoritmo se debe incluir el comando `PullAlgorithm 1`

4.4.3 Pruebas de performance

Las primeras pruebas se realizaron en la topología en profundidad, puesto que el algoritmo fue pensado para funcionar de buena manera en este tipo de topologías. Uno de los resultados que se esperaba era una mejora en la ubicación de los contenidos en la red, ya que este algoritmo evita poner múltiples copias de un archivo en la topología de profundidad. Las gráficas de distribución temporal de contenidos en cada nodo (figura 4.8) reflejan que efectivamente hubo una mejora en la distribución de los contenidos. Se puede apreciar que los archivos almacenados en cada nodo son distintos, y que en los nodos más cercanos se almacenan los archivos más populares.

Los resultados son extremadamente alentadores, ya que se logró algo que con los algoritmos antes planteados no había sucedido: que los archivos almacenados sean suficientemente disjuntos. El resultado que llama más la atención es el del nodo más lejano al cliente, donde se logró que casi no se guardara el archivo más popular y sí se guardaron los archivos que correspondían dado el orden de popularidad . Éste es el algoritmo que logró el resultado más cercano a la distribución óptima en la topología de línea o de profundidad.

Al analizar detenidamente las gráficas podemos observar que el nodo 4 guarda casi un 90% del tiempo el archivo más popular y en un porcentaje similar los archivos

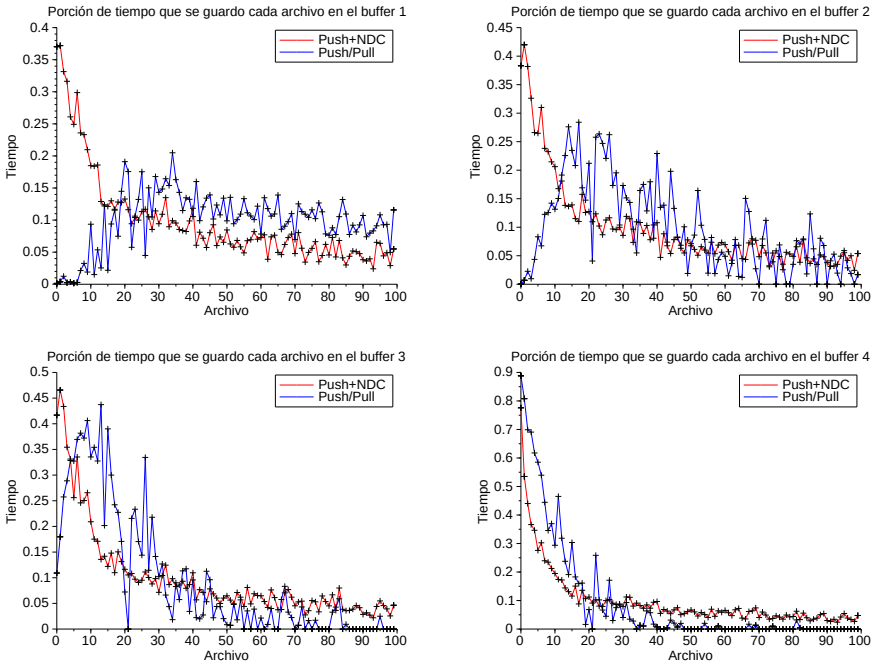

Figura 4.8. Distribución temporal de los archivos almacenados en cada nodo

más populares. Pero más importante aún es que en el resto de los nodos también se acompañó el perfil de demanda, ya que en el nodo 3, se vuelve casi nulo el tiempo de almacenamiento de los primeros archivos, dando lugar al resto de los archivos menos populares. Esta mejora significativa en la ubicación de los archivos se vio reflejada en el resto de las métricas como se presenta a continuación.

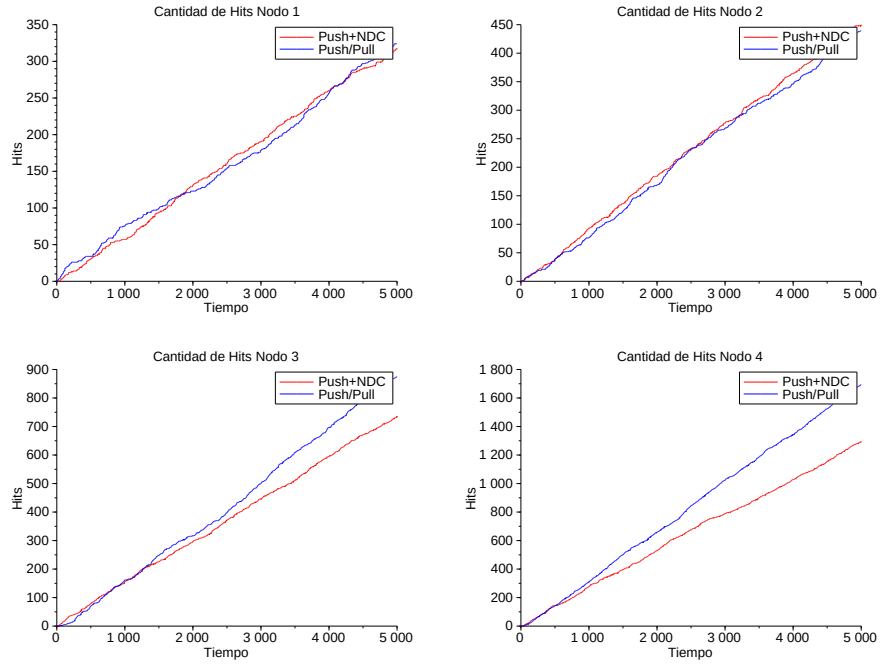

Figura 4.9. Avance temporal de los *hits* producidos en cada nodo

Si observamos las gráficas de la cantidad de *hits* en la figura 4.9, verificamos lo que esperábamos que dada la cercanía a esta distribución óptima, se traduce en una mejora en la tasa de *hits*.

Se tiene que la mayor diferencia entre tasas de *hits* se produce en los nodos más cercanos al cliente. El hecho de que no haya mejorado de gran manera la tasa de *hits* en los caché más lejanos se puede explicar debido a que llegan menos pedidos a estos nodos por lo que se producen menos *hits*. Finalmente se midió el *delay* obteniendo `DELAY MEDIO TOTAL 0.188441` el cual es el mejor resultado obtenido hasta ahora.

Dado el buen funcionamiento en la topología de la línea, resta verificar que este buen comportamiento también se traslada a las topologías de árbol. Para esto tomaremos como ejemplo la topología de la figura 4.10. En este caso será un árbol binario de 2 niveles, donde los dos hijos (nodos 2 y 3) y el padre (nodo 1), tienen memorias caché de tamaño 5. El *custodian* se encuentra en la raíz del árbol, algo muy común en estas topologías. Al igual que en la topología de profundidad todos los enlaces tendrán un retardo de 50 ms. En cuanto a los pedidos en las hojas del árbol se decidió que en cada hoja haya pedidos disjuntos (los realizados por los clientes 2 y 3), pero además se agregaron pedidos comunes (dados por los clientes 4 y 5). Todos los clientes tienen el mismo *rate*, para que sea igual de probable pedir un contenido común a ambos hijos que uno distinto.

En esta topología no es clara la distribución óptima, pero lo que si se espera es que el nodo padre dé mayor relevancia a los pedidos comunes que a los disjuntos, ya que estos llegarán con el doble de probabilidad.

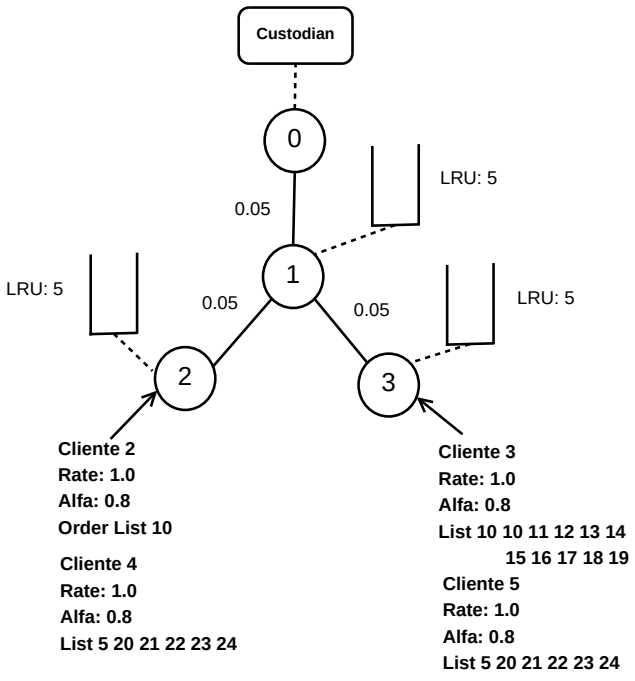

Figura 4.10. Topología en amplitud

Ahora al momento de ver la distribución que se produce por este algoritmo se tienen los resultados de la figura 4.11

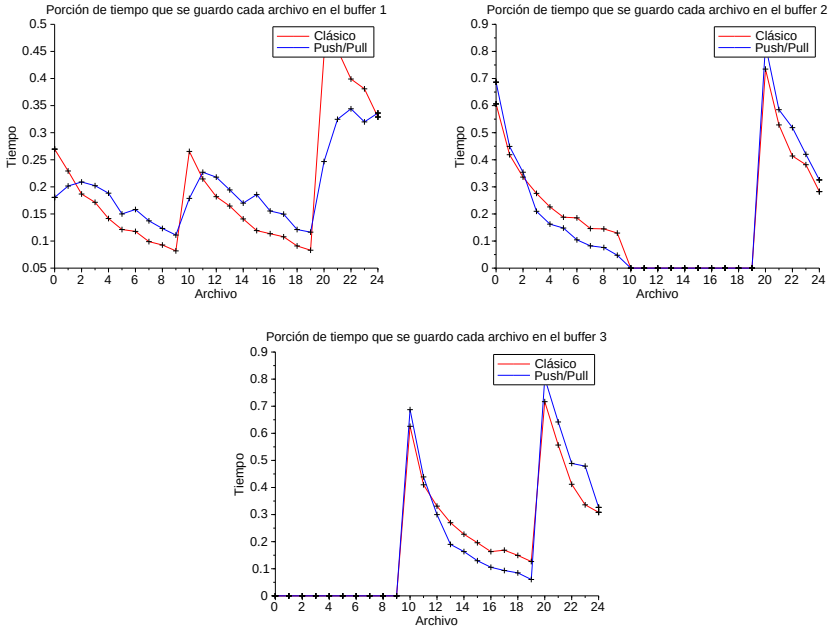

Figura 4.11. Distribución temporal de los archivos almacenados en cada nodo

Vemos que en esta distribución no hay grandes diferencia con la distribución clásica, esto se debe a que no hay ningún mecanismo que evite que los archivos al ser pedidos en el padre, bajen de inmediato a los hijos. Por esta razón no se logra almacenar una porción importante del tiempo los archivos comunes a ambos hijos, en el nodo padre.

Al analizar la cantidad de *hits* que se presentan en la figura 4.12, se puede ver que en los hijos (nodos 2 y 3) no se observa gran diferencia entre ambos algoritmos. Sin embargo al observar el nodo 1, el algoritmo de *pull* supera de forma amplia el algoritmo clásico, lo cual es un resultado muy alentador.

Este resultado además viene acompañado por una mejora en el *delay*. Al medir el *delay* se obtuvo, utilizando el algoritmo clásico:

```
DELAY MEDIO CLIENTE 2 es 0.12786
DELAY MEDIO CLIENTE 3 es 0.12508
DELAY MEDIO CLIENTE 4 es 0.07953
DELAY MEDIO CLIENTE 5 es 0.08104
```

mientras que utilizando el algoritmo de *Pull* se obtuvo:

```
DELAY MEDIO CLIENTE 2 es 0.10886
```

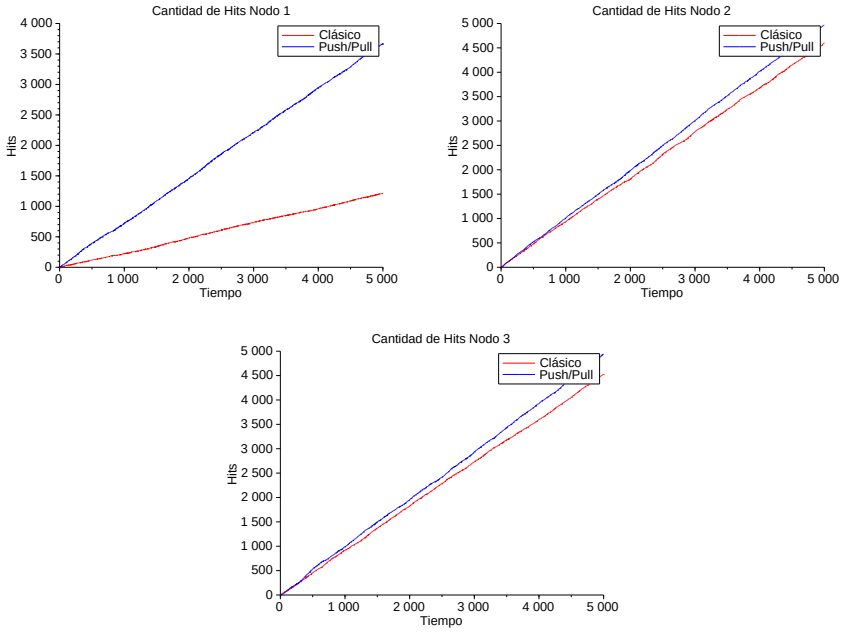

Figura 4.12. Avance temporal de los *hits* producidos en cada nodo

```
DELAY MEDIO CLIENTE 3 es 0.11109
DELAY MEDIO CLIENTE 4 es 0.05671
DELAY MEDIO CLIENTE 5 es 0.05985
```

Se puede confirmar que efectivamente no hay grandes diferencias entre este algoritmo y los anteriores, para las topologías amplias como la de la figura 4.10.

4.4.4 Problemas detectados

En cuanto a la topología de profundidad no se detectaron grandes falencias por lo que se piensa que el resultado obtenido es satisfactorio. El problema detectado es en la topología de amplitud, ya que no se logró mejorar la distribución de los contenidos. Se identifica que este problema se genera puesto que el archivo, que es descargado desde el padre, se almacena de forma inmediata en el hijo. Esto se puede observar como una falta de conocimiento de los pedidos que se realizan al nodo padre, ya que este no registra la información de donde viene cada uno de los pedidos a los que responde.

4.5 ALGORITMO DE INERCIA

4.5.1 Descripción del algoritmo

En el algoritmo anterior se detectó como problema el hecho de que no se alcanza la distribución óptima en la topología en amplitud. Se entiende que este problema se debe principalmente en que se baja el archivo a los hijos de manera demasiado rápida. En este tipo de topología se debe mantener algún tipo de información extra de la red local para poder alcanzar la distribución óptima. Por esta razón el algoritmo que utiliza cada nodo debe utilizar más información de los pedidos, para tener un mayor conocimiento de cómo se está comportando la red. A partir de esta necesidad de usar más información de los pedidos surge el algoritmo de inercia. Este algoritmo lo que implementa es que para que un padre le pase un archivo a su hijo, debe suceder que el mismo hijo sea quien realice los últimos P pedidos por este archivo. Llamaremos barrera de inercia al valor de esta variable P. Al incorporar este concepto se espera que el archivo que está en el padre, y es requerido por sus hijos de igual manera, sea más difícil que pase a éstos. Para que suceda debe haber una racha de P pedidos de un nodo antes que se genere un pedido desde el otro hijo. Si un archivo es pedido muchas veces por un único hijo, entonces este, luego de P pedidos resueltos por el nodo padre, se descarga al hijo.

4.5.2 Implementación en el código

Para mantener la información que se obtiene de los pedidos, en cada *FileManager* se generan dos *arrays* de largo la cantidad total de archivos del sistema. El primero de ambos *arrays* lleva el nombre de `contadorPedidos`, que como lo dice su nombre lleva la cuenta de cuántos pedidos se realizaron desde el mismo nodo. El otro *array* es denominado `nodoProvenientePedido`, que almacena cuál es nodo de donde proviene el pedido. Para saber de dónde proviene cada uno de los mensajes se debió agregar la variable `ultimoNodo` en la clase mensaje. La variable se carga cada vez que se genera un nuevo evento, cargando en la variable el nodo que genera el evento puesto que es el último en tener el pedido antes de redireccionarlo. En cuanto al contador de pedidos, cada vez que se desaloja el archivo de la memoria caché, este se vuelve a 0. Cuando el archivo es solicitado por un nodo distinto del almacenado, el contador se actualiza a 1.

Al mantener esta dinámica de actualizaciones del contador, se logró que todo archivo que no es almacenado cargue en la variable `contadorPedidos` un valor nulo. Esto es importante puesto que al saber que los archivos no almacenados no tienen información sólo se deben manejar B datos, siendo B el tamaño del *buffer*. La implementación realizada en el simulador utiliza *arrays* de tamaño fijo K el total de archivos, que facilita actualizar los distintos contadores, aunque en una futura versión se pueden mantener los contadores con una *ArrayList* que nunca supere en largo el tamaño del *buffer* B.

En cuanto a la barrera de inercia P, ésta se puede modificar mediante la entrada del simulador. Para modificar este valor se debe ingresar en la entrada el comando `BarreraInercia`, acompañado por el valor de P. Es importante notar que este algoritmo se utilizará únicamente si está activado el algoritmo de *push/pull*. En el caso de que el valor de la barrera de inercia sea 1, el resultado es equivalente a directamente no tener activado el algoritmo de inercia.

4.5.3 Pruebas de performance

Este algoritmo en una primera instancia será evaluado en la topología en profundidad para verificar que este cambio no haya distorsionado los buenos resultados conseguidos hasta el momento. Además esta primera prueba se realizará para distintas barrera de inercia, las cuales se adoptaran los valores de $P = 2, P = 3$ y $P = 4$.

Primero se observarán los tiempos de almacenamiento de cada uno de los archivos en los distintos nodos en la figura 4.13.

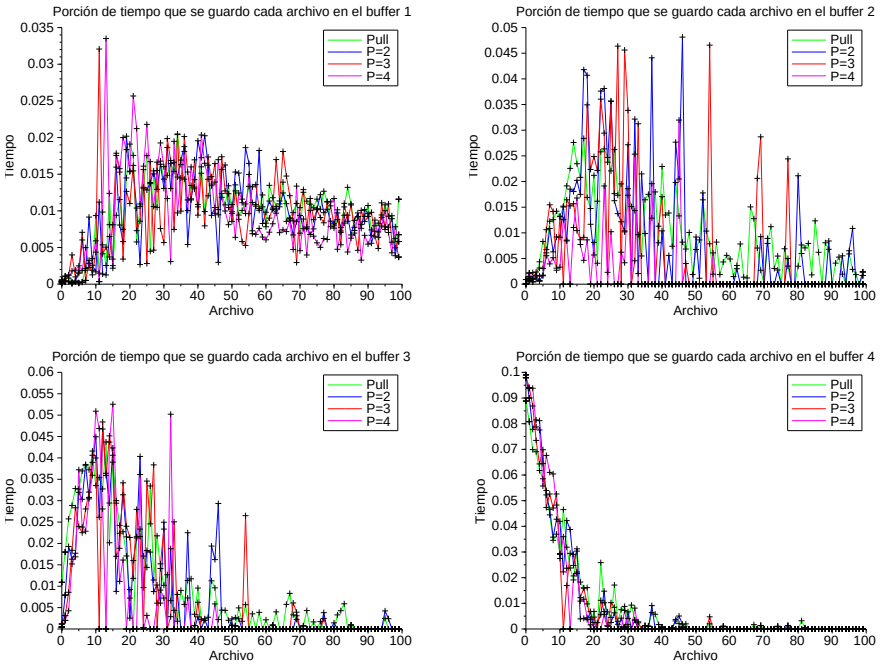

Figura 4.13. Distribución temporal de los archivos almacenados en cada nodo

Se puede ver que no hay grandes diferencias entre el algoritmo hasta ahora planteado con o sin inercia. Este resultado se considera muy bueno ya que el algoritmo de *pull* generaba una distribución de contenidos muy cercana a la óptima en la topología de profundidad y la inercia no modifica esto.

Sin embargo al observar la cantidad de *hits* en la figura 4.14 vemos que se producen importantes cambios.

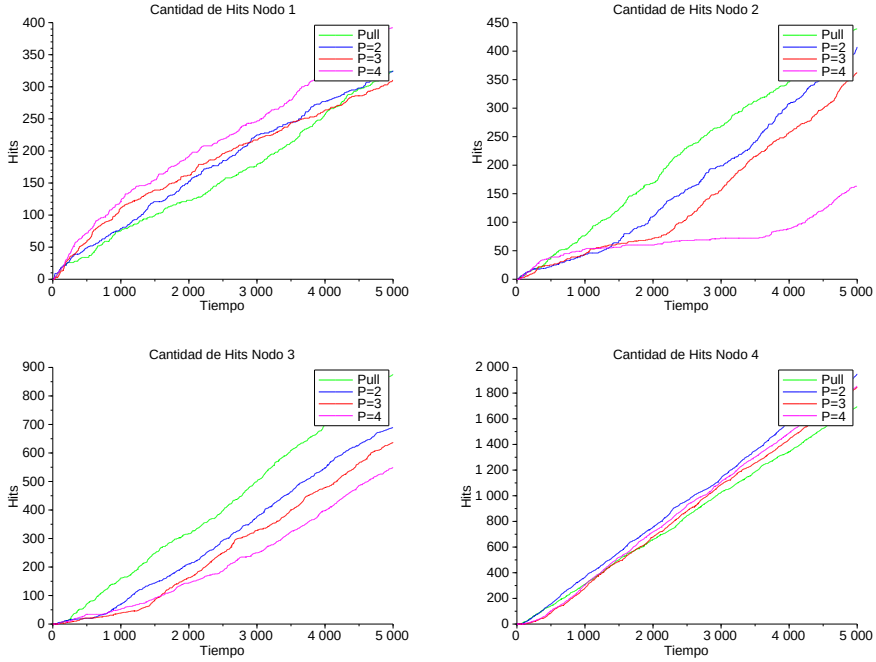

Figura 4.14. Avance temporal de los *hits* producidos en cada nodo, topología en profundidad

Las principales diferencias se pueden observar cuando la barrera de inercia toma un valor de 4. En este caso se puede observar que se producen algunos transitorios debido a la lentitud que tiene el algoritmo en llenar todas las memorias de la red. Este transitorio tan importante repercute en el valor del *delay* total. Se obtiene que para $P = 2$ el `DELAY MEDIO TOTAL 0.1816`, mientras que para $P = 3$ el `DELAY MEDIO TOTAL 0.1926` y para $P = 4$ se tiene que el `DELAY MEDIO TOTAL 0.2044`. En el algoritmo de *pull* se había obtenido un `DELAY MEDIO TOTAL 0.1841`.

Luego de analizar el efecto de estas barreras de inercia en una topología en profundidad, es momento de estudiar el caso que da origen a esta idea.

Analizando en la topología de amplitud primero observamos la cantidad de *hits* que se muestran en la figura 4.15. Se observa que las diferencias en los *hits* no son mayores, y las pequeñas diferencias son a favor de esta nueva implementación en el algoritmo. Esta afirmación se justifica debido a que hay una mayor cantidad de *hits* en los nodos hijos. Esto también es una explicación de por qué hay menos *hits* en el nodo padre, puesto que se filtran más pedidos en los hijos. Si se analiza el *delay* se obtiene para $P = 2$ `DELAY MEDIO TOTAL 0,06519`, mientras que para $P = 3$ `DELAY MEDIO TOTAL 0,062465` y para $P = 4$ con un `DELAY MEDIO TOTAL 0,06156` En todos los casos mejorando el `DELAY MEDIO TOTAL 0,06955` del algoritmo de *pull*.

La distribución de contenidos se muestran en la figura 4.16. Se observa que los archivos comunes (del 20 al 24) que no son los más populares se mantienen más tiempo en el nodo padre. Es importante destacar que los archivos comunes más consultados son tan populares que la ubicación óptima se da en los nodos hijos.

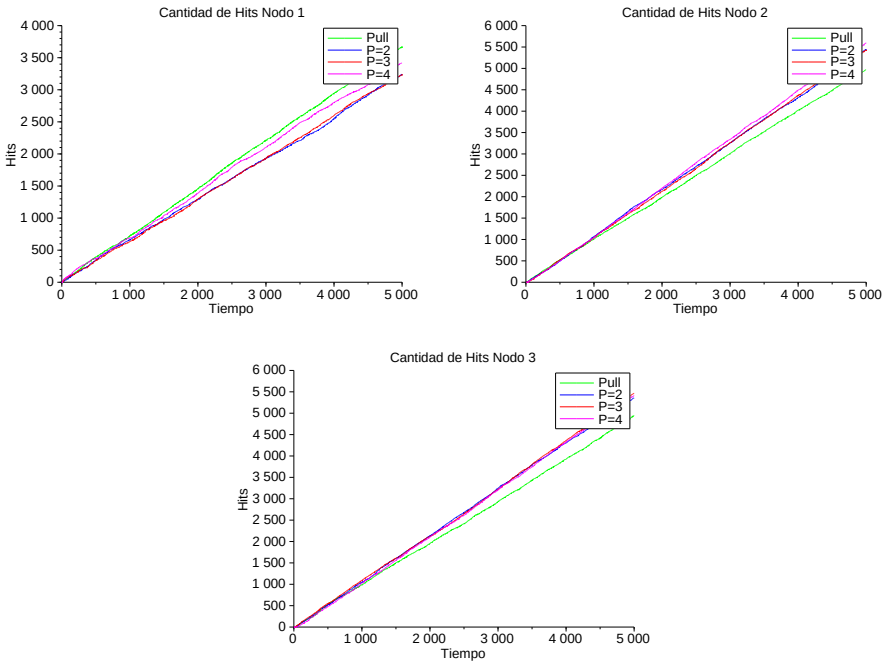

Figura 4.15. Avance temporal de los *hits* producidos en cada nodo, topología en amplitud

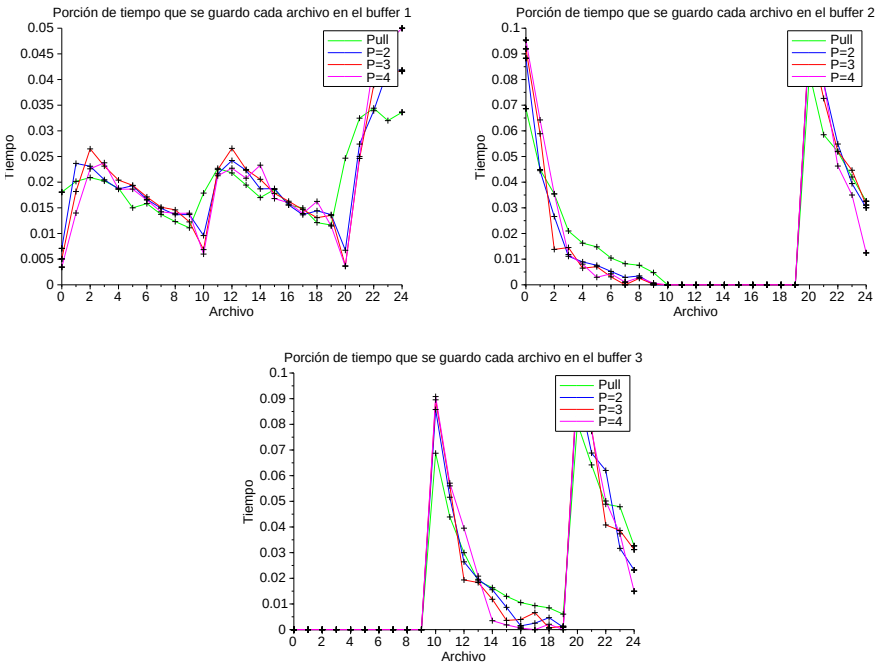

Figura 4.16. Distribución temporal de los archivos almacenados en cada nodo

Se puede concluir que la distribución temporal de contenidos de este algoritmo se separara de la distribución del algoritmo *pull*.

Es importante destacar que no se define un valor fijo para el valor de la barrera de inercia, sino que se deja como un parámetro de diseño para que el usuario lo

modifique para obtener el mejor resultado posible. A partir del estudio realizado se verifica que hay cierto compromiso de proyecto a la hora de elegir esta barrera. Si se elige una barrera muy alta, se producen grandes transitorios que afectan los resultados globales. Mientras que en topologías de amplitud esta barrera de inercia acerca la distribución de contenidos a su forma óptima.

Como cierre de este capítulo se puede concluir que se halló un algoritmo que produce una gran mejora en las topologías de profundidad. En estas topologías se logra una distribución de contenidos cercana a la óptima. En las topologías de amplitud se alcanzan buenos resultados pero no tan indiscutibles como en el caso anterior. En el próximo capítulo se mostraran las pruebas que se realizaron para validar el uso de este nuevo algoritmo.

Capítulo 5

Pruebas

En este capítulo se realizarán distintas pruebas para comparar el algoritmo que se ha tratado como clásico y la última versión del algoritmo desarrollado. Para comparar ambos algoritmos se simularán en las topologías antes discutidas en diferentes formatos y en nuevas topologías. Se estudiarán las diferencias entre ambos algoritmos al variar la profundidad de la red, al agregar clientes en puntos intermedios y finalmente variando la cantidad de hijos en la topología de amplitud. Además se plantearán topologías más complejas que las vistas en el Capítulo 4, para que de esta forma se combinen los problemas de las topologías de línea y las topologías de árbol.

5.1 PROFUNDIDAD VARIABLE

Se comienza por la topología de profundidad variable. Asumimos que en dicha topología existe un único cliente. Este cliente se encontrará en un extremo mientras que en el otro se encontrará el *custodian*. Además tendremos que en total habrán 100 contenidos en la red, los cuales tendrán como perfil de popularidad una *Zipf*, con un valor de $\alpha = 1,2$, uno de los valores más comunes según el estudio realizado en [18] y asumiremos que el cliente tiene un *rate* de 1 pedido por segundo. En todos los nodos habrá un caché que podrá almacenar hasta 10 archivos. Por esta razón se realizarán pruebas con hasta 10 nodos intermedios, puesto que en este caso ya se podrían almacenar todos los contenidos. Asumiremos por último que todos los enlaces tendrán un retardo de $50ms$, mientras que el cliente realiza un pedido por segundo en promedio. Es importante notar que a mayor profundidad, tiene menos sentido la hipótesis de ZDD, ya que el tiempo que demorará en ser resuelto un pedido puede alcanzar 1 segundo.

La medida que utilizaremos para comparar ambos algoritmos, será el *delay*. Se dividirá este *delay* entre 50 ms de forma que la medida obtenida sea la cantidad de

saltos promedio que debe realizar un pedido para ser atendido.

En la figura 5.1 se observa la comparación entre ambos algoritmos.

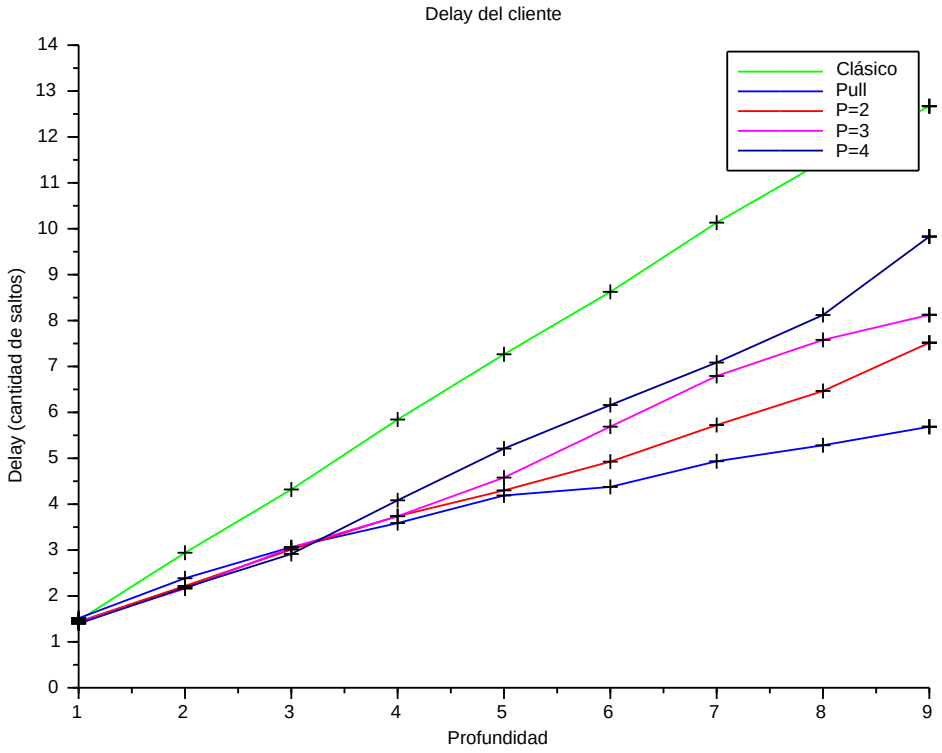

Figura 5.1. Comparación de los algoritmos en profundidad

Se varían los diferentes valores de P (la barrera de inercia) puesto que en el capítulo anterior no se alcanzó ningún resultado concluyente sobre el valor de la misma, sino que se incluye como un parámetro más para que utilice el usuario.

Se observa que el algoritmo de *pull* es el de mejor performance para las topologías más profundas. En los últimos casos sucede incluso que mejora a la mitad el *delay* percibido por el cliente respecto al algoritmo clásico. Es importante notar, que para profundidades bajas, el algoritmo de inercia se comportó de buen modo, pero para mayores profundidades el *delay* fue mayor. Este fenómeno se explica por los grandes transitorios que tiene este algoritmo para barreras de inercia muy grandes.

5.2 AMPLITUD VARIABLE

En este caso estudiaremos las topologías de árbol, de 2 niveles, pero de múltiples hijos. Al variar la cantidad de hijos se podrá observar cómo se comportan ambos algoritmos ante la presencia de múltiples pedidos en el padre desde diferentes hijos.

Al suceder esto el nodo padre debe decidir con la información que obtiene de sus hijos qué archivos almacenar.

Se plantean dos escenarios antagónicos, el primero donde todos los hijos piden exactamente los mismos contenidos y un segundo escenario donde la popularidad de los contenidos es seleccionada al azar. Primero en la figura 5.2, se pueden ver los resultado si todos los clientes realizan los mismos pedidos.

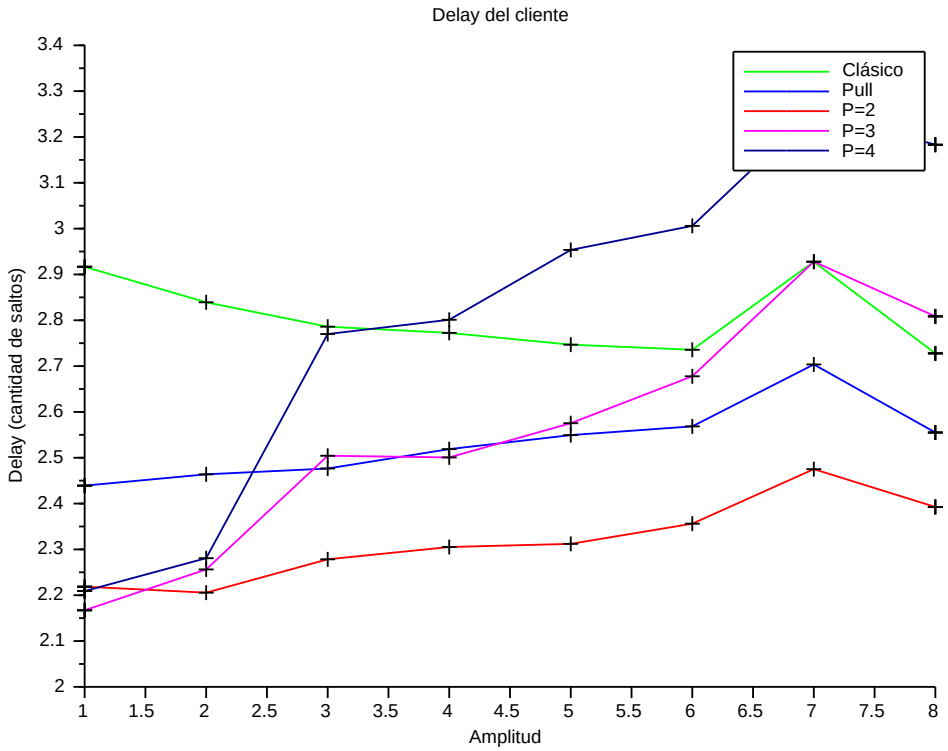

Figura 5.2. Comparación de los algoritmos en amplitud con iguales pedidos en los hijos

Se observa en este caso que el algoritmo de mejor performance es el de barrera de inercia $P = 2$. Esta simulación justifica en parte el uso de la barrera de inercia, puesto que logra una mejora sustancial no sólo respecto del algoritmo clásico sino también respecto del algoritmo de *pull*. Sin embargo el algoritmo de mayor inercia resulta el de peor performance.

Ahora al analizar el caso en que las popularidades de los archivos se distribuyen al azar en cada uno de los clientes, se tiene el resultado de la figura 5.3

Vemos que los resultados son similares al caso anterior. La única diferencia es que en general todos los algoritmos tienen un mayor *delay* para responder a las consultas. Sin embargo el algoritmo de inercia con barrera $P = 2$ es el que se comporta de mejor manera. Por esta razón se concluye que en este tipo de redes

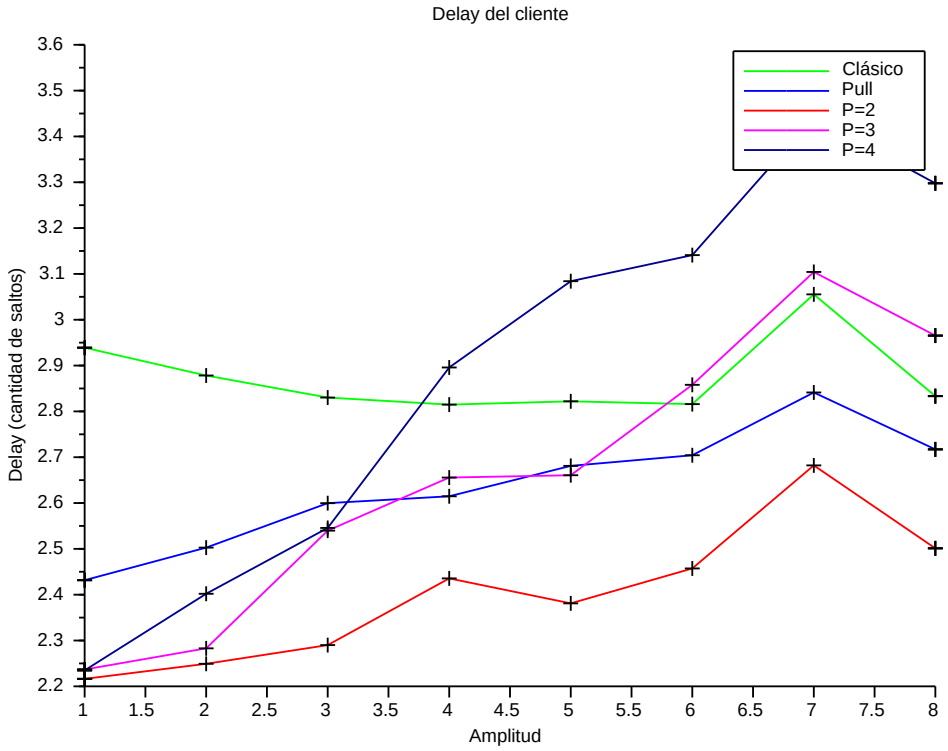

Figura 5.3. Comparación de los algoritmos en amplitud con distintos pedidos en los hijos

puede ser útil la implementación de este algoritmo.

5.3 PROFUNDIDAD VARIABLE CON CLIENTES INTERMEDIOS

La idea detrás de esta prueba es observar el comportamiento de ambos algoritmos cuando se produce una "interferencia" entre pedidos. Para hacer esto se colocan clientes en todos los nodos de la línea, primero todos los clientes tendrán el mismo perfil de popularidad para que esta "interferencia" de pedidos cruzados sea mínima y luego se realizará otra prueba donde el orden de prioridad de los archivos sea al azar.

Algo que se utilizará es que el *rate* de pedidos de los clientes es cada vez más bajo a medida que estos están ubicados más cerca del *custodian*. La idea detrás de esta hipótesis es que esta topología se puede ver como la rama de una topología de árbol. Al estar más cerca de la raíz llegan los pedidos no resueltos por los hijos, es decir que ya se filtraron varios de los pedidos realizados en las hojas, por esta razón el *rate* visto desde ese nodo es menor que el visto desde las hojas.

Entonces en el caso que todos los nodos piden los mismos archivos se tienen las gráficas de cantidad media de saltos de la Figura 5.4.

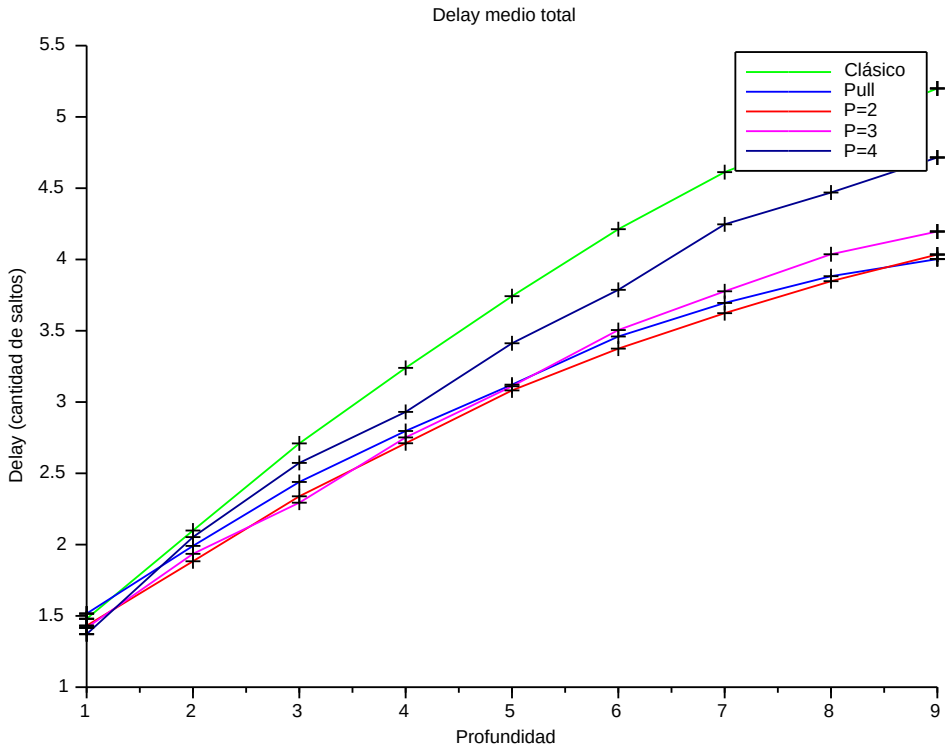

Figura 5.4. Comparación de los algoritmos en profundidad con varios clientes con los mismos pedidos

Nuevamente se observa una mejor performance del algoritmo de *pull* y de el algoritmo de barrera de inercia 2.

A continuación se realiza la misma prueba pero ahora los clientes piden por 100 archivos cuyo orden es seleccionado al azar. Los resultados se pueden observar en la figura 5.5

Se observa que los resultados no varían demasiado del caso anterior, de lo que se puede deducir que las mejoras que introduce el nuevo algoritmo son inmunes a las "interferencias" antes descriptas.

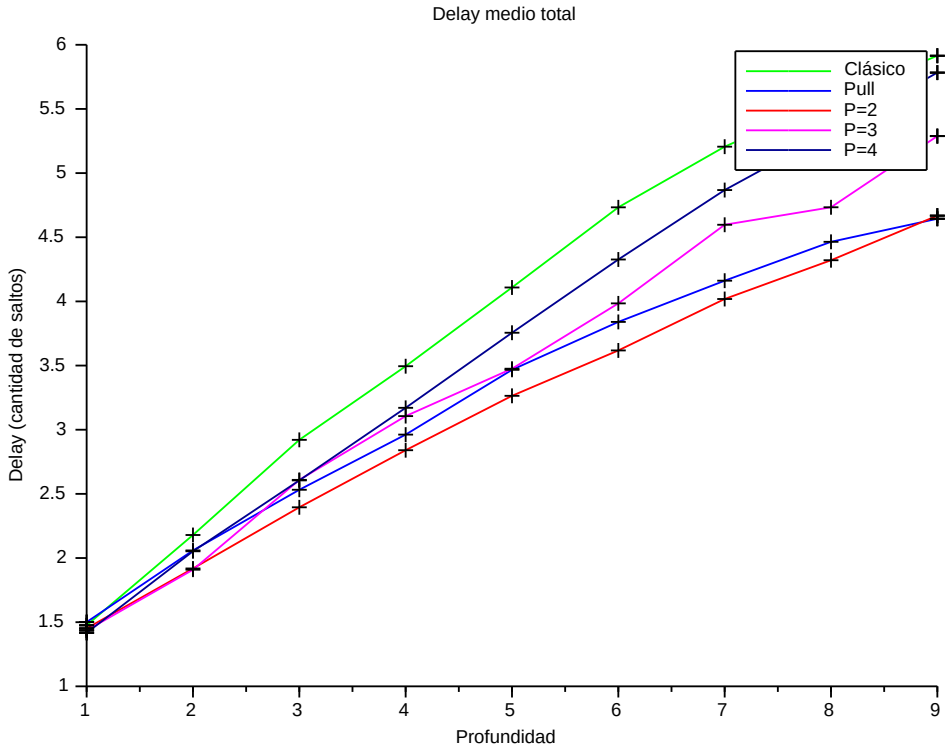

Figura 5.5. Comparación de los algoritmos en profundidad con varios clientes con distintos pedidos

5.4 CASO GENERAL

Para cerrar este capítulo de pruebas, se propone una topología más general. En esta topología se incluyen clientes únicamente en un extremo de la topología mientras que los *custodians* se encuentran en el otro extremo. Se tomó la decisión de colocar los clientes en un extremo puesto que en la mayoría de la redes de contenidos distribuidos se asemejan a árboles con los clientes en las hojas y con los *custodians* en la raíz.

La topología que se utilizó para comparar los algoritmos es la presentada en la Figura 5.6. Para evitar que la figura quede sobrecargada, no se indican los valores de *delay* de los enlaces, los cuales son todos iguales con un valor de 0.05.

En este caso además se agregaron clientes que realizan pedidos por los mismos cinco contenidos comunes en todos los nodos y otros clientes que piden por contenidos que no mantienen ningún orden en especial. Se agregan además *buffers* con políticas FIFO y LRU para mostrar de esta manera que el algoritmo funciona sin importar la política de sustitución seleccionada. Por último se observa que en la figura hay dos *custodians* que almacenan contenidos disjuntos, algo común en las redes de distribución.

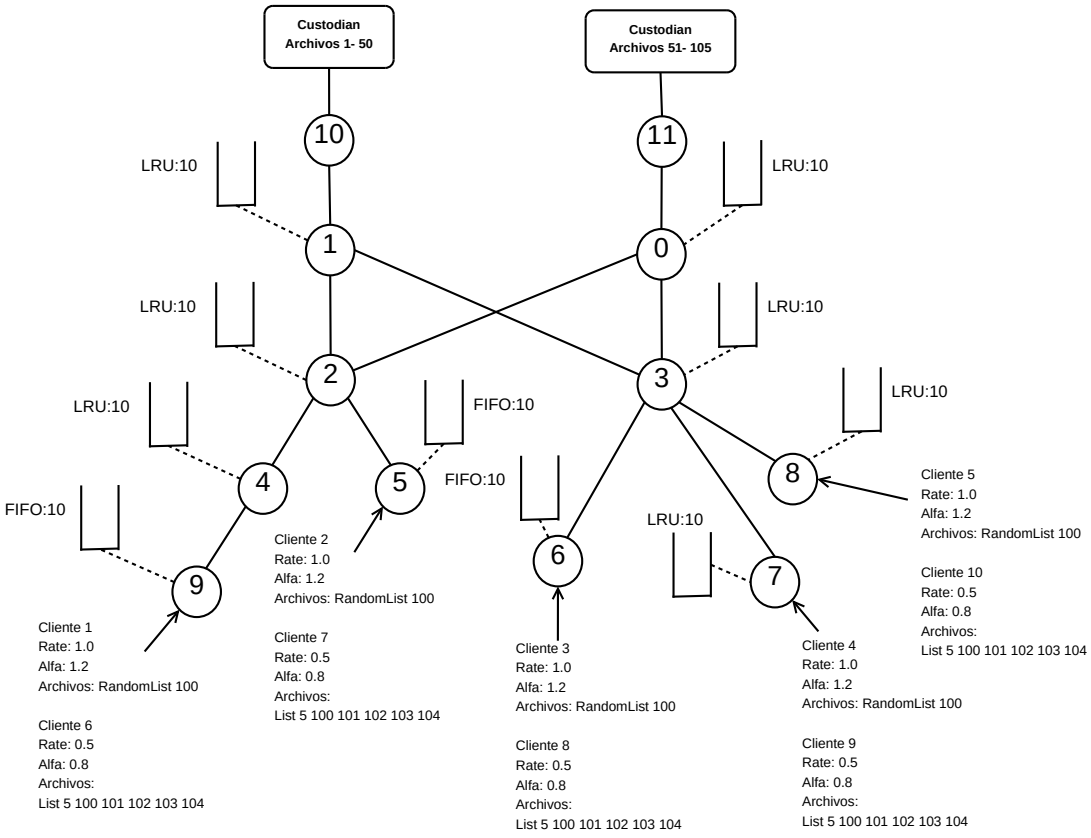

Figura 5.6. Topología genérica donde se ensayan los algoritmos

Al simular los algoritmos en esta topología se obtuvieron los resultados que se muestran en la tabla 5.1.

Cliente	**Clásico**	**Pull**	**P = 2**	**P = 3**	**P = 4**
Cliente 1	0,29634	0,20564	0,19585	0,21069	0,22912
Cliente 2	0,22118	0,17140	0,15963	0,17301	0,18027
Cliente 3	0,21839	0,18146	0,16865	0,16595	0,16707
Cliente 4	0,21891	0,16930	0,17181	0,16233	0,17861
Cliente 5	0,20858	0,17502	0,16796	0,16590	0,17018
Cliente 6	0,13120	0,02387	0,02225	0,02986	0,04190
Cliente 7	0,09216	0,02390	0,01886	0,01453	0,02389
Cliente 8	0,09188	0,03320	0,01427	0,01921	0,03652
Cliente 9	0,08830	0,02461	0,02966	0,02650	0,02119
Cliente 10	0,08281	0,03811	0,01990	0,02189	0,02299
Total	**0,140657**	**0,097466**	**0,091636**	**0,093387**	**0,099849**

Tabla 5.1. *Delays* por cliente en cada uno de los algoritmos

Se puede concluir entonces que el algoritmo de inercia, particularmente con una

barrera de inercia 2 ($P = 2$) funciona de buena manera en diferentes situaciones. Es de destacar que en ninguno de los casos estudiados sucedió que el algoritmo nuevo tuviera una performance más baja que la del algoritmo clásico. Este hecho, unido a la sencilla implementación del algoritmo en redes ya instaladas, hace pensar que puede ser una buena modificación a los métodos utilizados en la actualidad.

Capítulo 6

Conclusiones y trabajo a futuro

En este capítulo se presentará un resumen de los resultados obtenidos en este proyecto y las posibles líneas de trabajo a futuro.

6.1 CONCLUSIONES

En este proyecto se han alcanzado distintos resultados, para exponer estos de forma ordenada, se presentarán las conclusiones que se desprenden de cada uno de los capítulos.

En el capítulo 2, se logró escribir una formulación del problema de distribución de contenidos. Utilizando dicha formulación se pudo resolver el problema de ruteo de los pedidos por archivos. Se concluyó que en un ruteo óptimo, las rutas a seguir siempre son las menos costosas (en nuestro análisis las de menor retardo) hasta donde se almacena el contenido buscado. Como es imposible saber de antemano dónde está almacenado cada uno de los archivos, se toma la decisión de utilizar el camino con menos retardo hacia el *custodian* del contenido deseado. Se probaron diferentes cambios en la formulación del problema utilizando distintas ideas pero no se pudo alcanzar una formulación resoluble mediante optimización convexa. Por esta razón se concluyó que no es sencillo dar una formulación convexa al problema.

En el capítulo 3, se lograron alcanzar los distintos objetivos planteados para el simulador. Primero se diseñó un sistema que permite modelar topologías de forma arbitraria. Además se logró que este simulador fuera de eventos discretos, lo que permite resolver pedidos de forma asíncrona. Esto permite emular las redes de contenidos reales de una manera mucho más fidedigna. Se pudieron implementar las diferentes políticas de reemplazo de los *buffers*, y al haber utilizado programación orientada a objetos es muy sencillo implementar nuevas políticas. En conclusión al capítulo se logró crear un simulador flexible y modular que permite reflejar de

manera fiel el comportamiento de las redes que se han estudiado a lo largo de esta documentación.

En el capítulo 4, es donde aparecen los resultados más importantes de este trabajo. Aquí se plantearon nuevos algoritmos los cuales tuvieron una mejor *performance* que los utilizados en la actualidad, se llegó a esto mediante distintas modificaciones que fueron mejorando poco a poco las deficiencias de los algoritmos anteriores. Finalmente se alcanza el algoritmo de inercia (en particular con $P = 2$) que funciona de excelente forma en topologías con pocas bifurcaciones mientras que en las topologías de árbol con muchos hijos, se comporta de mejor manera que los algoritmos clásicos.

Por último en el capítulo 5, se logró comprobar el buen funcionamiento del algoritmo desarrollado en el capítulo anterior. Las conclusiones que se desprenden de estos resultados, unido a la sencilla implementación de estos algoritmos en el simulador, únicamente se realizan intercambios de mensajes entre nodos, hace concluir que estas modificaciones son perfectamente aplicables a redes reales.

6.2 TRABAJO A FUTURO

Al igual que en el desarrollo de las conclusiones, esta sección se desarrollará capítulo a capítulo.

Comenzaremos con el capítulo 2, el cual no deja claro que línea seguir a futuro. El principal inconveniente es que al no haberse podido convexificar el problema, no se pueden realizar demasiadas aproximaciones a la solución mediante estos métodos.

En el capítulo 3 una línea de trabajo natural es generar una interfaz de usuario gráfica, la cual sea más agradable. Sería muy cómodo para el usuario poder generar las topologías de las redes si la interfaz fuera del tipo *Drag and Drop*. En esta interfaz los elementos se colocan de forma natural siendo colocados e interconectados con el mouse.

En cuanto al capítulo 4, los resultados obtenidos fueron muy alentadores, por lo que se cree que no hay mucho lugar para seguir trabajando en búsqueda de más mejoras. Algo que podría ser interesante hacer en un futuro es definir de forma analítica el valor de la barrera de inercia para decidir cuál es el mejor valor en cada red.

Finalmente en el capítulo 5, lo natural sería poder realizar las pruebas del algoritmo desarrollado en redes reales. Para poder trabajar en redes reales se deberían implementar los protocolos de mensajes utilizados en el simulador, en los *routers* físicos de las redes.

Bibliografía

[1] Wikipedia. (2015) Red de entrega de contenidos — wikipedia, the free encyclopedia. [Online; accessed 3-January-2015]. [Online]. Disponible: http://es.wikipedia.org/wiki/Red_de_entrega_de_contenidos

[2] S. Borst, V. Gupta, and A. Walid, "Distributed caching algorithms for content distribution networks," in *INFOCOM, 2010 Proceedings IEEE*. IEEE, 2010, pp. 1–9.

[3] E. J. Rosensweig, D. S. Menasché, and J. Kurose, "On the steady-state of cache networks," in *INFOCOM, 2013 Proceedings IEEE*. IEEE, 2013, pp. 863–871.

[4] N. Laoutaris, H. Che, and I. Stavrakakis, "The lcd interconnection of lru caches and its analysis," *Performance Evaluation*, vol. 63, no. 7, pp. 609–634, 2006.

[5] S. Tewari and L. Kleinrock, "Proportional replication in peer-to-peer networks." in *INFOCOM 2006, IEEE*. IEEE, 2006.

[6] G. Carofiglio, M. Gallo, L. Muscariello, and D. Perino, "Modeling data transfer in content-centric networking," in *Proceedings of the 23rd international teletraffic congress*. International Teletraffic Congress, 2011, pp. 111–118.

[7] S. Ioannidis and P. Marbach, "On the design of hybrid peer-to-peer systems," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 1. ACM, 2008, pp. 157–168.

[8] S. Narayana, W. Jiang, J. Rexford, and M. Chiang, "Joint server selection and routing for geo-replicated services," in *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*. IEEE Computer Society, 2013, pp. 423–428.

[9] A. Sharma, A. Venkataramani, and R. K. Sitaraman, "Distributing content simplifies isp traffic engineering," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 1. ACM, 2013, pp. 229–242.

[10] M. Gallo, B. Kauffmann, L. Muscariello, A. Simonian, and C. Tanguy, "Performance evaluation of the random replacement policy for networks of caches," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1. ACM, 2012, pp. 395–396.

[11] A. Dan and D. F. Towsley, "An approximate analysis of the LRU and FIFO buffer replacement schemes," in *SIGMETRICS*, 1990, pp. 143–152. [Online]. Disponible: http://doi.acm.org/10.1145/98457.98525

[12] E. J. Rosensweig, J. Kurose, and D. Towsley, "Approximate models for general cache networks," in *INFOCOM, 2010 Proceedings IEEE*. IEEE, 2010, pp. 1–9.

[13] P. Barford, A. Bestavros, A. Bradley, and M. Crovella, "Changes in web client access patterns: Characteristics and caching implications," *World Wide Web*, vol. 2, no. 1-2, pp. 15–28, 1999.

[14] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 1. IEEE, 1999, pp. 126–134.

[15] H. Che, Z. Wang, and Y. Tung, "Analysis and design of hierarchical web caching systems," in *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3. IEEE, 2001, pp. 1416–1424.

[16] S. Ioannidis and P. Marbach, "Absence of evidence as evidence of absence: A simple mechanism for scalable p2p search," in *INFOCOM 2009, IEEE*. IEEE, 2009, pp. 576–584.

[17] K. Stamos, G. Pallis, A. Vakali, D. Katsaros, A. Sidiropoulos, and Y. Manolopoulos, "Cdnsim: A simulation tool for content distribution networks," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 20, no. 2, p. 10, 2010.

[18] C. Fricker, P. Robert, and J. Roberts, "A versatile and accurate approximation for lru cache performance," in *Proceedings of the 24th International Teletraffic Congress*. International Teletraffic Congress, 2012, p. 8.

Anexo A

Manual de Usuario

Este simulador se puede utilizar tanto en Windows como en Unix pero se asume que se tiene instalado JRE 1.7 o versiones más actuales.

Comenzaremos por la descripción de la entrada del simulador la cual es mediante un archivo de texto plano. Se debe tener en cuenta que cada comando del que se hablará en este manual de usuario, debe escribirse en una línea distinta.

El cuerpo de esta entrada se divide en 3 bloques, `Topologia`, `Aplicaciones` y `Simulacion`, los cuales se deben ingresar en ese orden.

Comencemos por el bloque `Topologia`, este comienza con el comando de inicio `begin Topologia` y finaliza con el comando `end Topologia`. En este bloque se agregan los elementos referentes a la parte física de la red que se desea modelar. En particular aquí se pueden ingresar nodos o enlaces, a continuación veremos cómo ingresar cada uno de estos elementos.

Comenzaremos por ingresar un nodo, para esto se utiliza el siguiente comando

```
Nodo 1
```

Es importante que se debe escribir exactamente "`Nodo`" respetando mayúsculas y minúsculas, y separado por un único espacio el entero con el que se identificará el nodo, este identificador no se tiene que haber utilizado antes. El simulador comunicará por la salida estándar si el nodo no es agregado ya sea porque existe un nodo con ese número o porque los parámetros no son del tipo adecuado.

En cuanto al ingreso de enlaces, este se realiza mediante el siguiente comando:

```
Enlace 1 2 0.05
```

Se comienza por el ingreso de la palabra "`Enlace`" donde se deben respetar mayúsculas y minúsculas nuevamente. Luego todos los datos deben ser separados únicamente por un espacio. Estos datos son en primer lugar los dos nodos que une el enlace y por último se carga el costo del enlace. Los primeros dos datos deben ser enteros, mientras que el último debe ser un real positivo.

El programa informa, si no existe alguno de los nodos de extremos, si ya existe un enlace que una estos nodos, si el costo no es positivo o si el formato no es el adecuado. Es importante que el grafo que se forme sea conexo, puesto que de caso contrario al finalizar el ingreso de los datos se informará de que no fue posible realizar la simulación por esta razón.

Un ejemplo de este bloque es:

```
begin Topologia
Nodo 0
Nodo 1
Enlace 1 0 0.05
Nodo 2
Enlace 2 1 0.05
Nodo 3
Enlace 3 2 0.05
Nodo 4
Enlace 4 3 0.05
end Topologia
```

Si se desea más información de este bloque se puede recurrir a la sección 3.4.1 del capítulo 3.

El siguiente bloque a analizar es el bloque de `Aplicaciones`. Este bloque será el encargado de colocar a los clientes, caché y *custodians* en la red. Además es el encargado de definir la cantidad total de archivos del sistema. Este bloque se inicia con la línea `begin Aplicaciones` y se finaliza con la línea `end Aplicaciones` .

El primer comando que se debe ingresar en este bloque es el que indica la cantidad total de archivos en la red. Esto se hace ingresando la siguiente línea:

```
listaArchivos 10
```

Donde el entero que se introduce es la cantidad de archivos que hay en la red. Nuevamente se debe ingresar el comando respetando mayúsculas y minúsculas y dejando un único espacio para ingresar el siguiente dato. El simulador comunica que hay un error en el caso de que alguna de las variables no tenga el formato adecuado o si la cantidad de archivos no es válida. Es importante señalar que los archivos creados se numerarán de 0 hasta el número ingresado menos 1. En el ejemplo se generarían los archivos de 0 al 9.

Continuando con las aplicaciones, si se desea ingresar un caché, se debe hacer mediante el siguiente comando

```
Cache 7 2 LRU
```

Donde el primer entero indica en qué nodo se ubicará el caché, y las otras dos variables indican tamaño y política de reemplazo del nodo. El tamaño debe ser un entero positivo, mientras que la políticas de reemplazo debe ser un String que sea exactamente "`LRU`", "`FIFO`" o "`Random`". Se debe respetar nuevamente mayúsculas y minúsculas al ingresar este último dato.

En primer lugar se verifica que el formato de la entrada sea el adecuado. Además el programa verifica que efectivamente exista el nodo donde se ubica el caché. También se verifica que el tamaño del *buffer* sea positivo y que exista la política que se indica. En el caso de que se agreguen varios caché en el mismo nodo, sólo se tendrá en cuenta el último que se ingresó.

Siguiendo con el bloque de aplicaciones, se indica cómo crear un cliente. Esto se realiza mediante el siguiente comando:

```
Cliente 3 4 1.0 0.8 OrderList 10
```

El comando comienza con la palabra Cliente, luego es acompañado por un entero que es el identificador del cliente. Este identificador es único para cada cliente, por lo que no se permite crear dos clientes con este mismo valor. El siguiente parámetro es el número de nodo en el que se ubicará el cliente. Luego aparecen dos parámetros reales, el primero es el *rate* y el segundo es el valor de α de la $Zipf(\alpha)$.

Finalmente se agrega el comando que indica cuáles serán los archivos que pedirá el cliente y el orden de estos. Recordar que el orden es importante, ya que el primer archivo de la lista será el más popular, y el último por lo tanto el menos popular. Hay tres tipos de orden, `OrderList X`, `RandomList X` y `List Y A B ...`.

Para utilizar estos, se deben respetar mayúsculas y minúsculas, y tener en cuenta que todos los enteros deben estar separados por un único espacio.

Comenzaremos por `OrderList X`, donde X es un entero positivo, genera una lista de archivos ordenados, desde 0 hasta $X-1$. Por lo que el archivo 0 será el más popular y el archivo número $X-1$ el menos popular.

En cuanto a `RandomList X`, donde X es un entero positivo, genera una lista de archivos de 0 hasta $X-1$, pero ahora el orden es decidido al azar.

Finalmente `List Y A B ...`, es el único comando que no tiene una única variable. La primera variable es un entero positivo Y, que indica el largo de la lista de contenidos. Luego los siguiente Y enteros, indican los índices de los archivos que se

desean agregar en ese mismo orden. Por ejemplo si se ingresa `List 4 3 7 5 9`, el cliente pedirá por 4 archivos, siendo el 3 el más popular y el 9 el menos popular.

El simulador validará que los datos ingresado tengan el formato correcto, que el *rate* sea positivo, que el valor de alfa no sea negativo. Además se validará que la cantidad de archivos por los que se pide sea positiva, y que los archivos que se pidan existan en la lista de archivos global del sistema.

Para finalizar con este bloque de Aplicaciones se explica cómo ingresar un *custodian*. Para hacer esto se utiliza el comando

`Custodian 3 RandomList 7`.

Este comando se compone por la palabra `Custodian` donde se debe respetar mayúsculas y minúsculas, separado por un único espacio el número del nodo donde se desea ubicar el *custodian*. Por último se ingresan con la misma convención que en el cliente, los archivos que se desean almacenar. En este caso el orden de los archivos es indiferente.

Es importante que al finalizar este bloque, se haya colocado al menos un cliente, todos los nodos tengan un caché o un *custodian* y que todos los archivos de la lista sean custodiados en algún nodo. En el caso de que no se cumpla alguna de estas condiciones, no se efectuará la simulación y esto se comunicará por la salida estándar.

Un ejemplo de este bloque es:

```
begin Aplicaciones
listaArchivos 10
Custodian 0 RandomList 10
Cache 1 1 LRU
Cache 2 1 LRU
Cache 3 1 LRU
Cache 4 1 LRU
Cliente 3 4 1.0 0.8 OrderList 10
end Aplicaciones
```

Si se desea más información de este bloque se puede recurrir a la sección 3.4.2 del capítulo 3.

Finalmente en cuanto al último bloque `Simulador`, al igual que los bloques anteriores tiene un comando de comienzo `begin Simulador` y uno de fin `end Simulador`. En este bloque se cargarán los distintos parámetros del simulador.

En primer lugar se debe cargar el tiempo que dura la simulación mediante el

comando

```
TiempoSimulacion 5000
```

En este comando se debe respetar al igual que en los otros casos mayúsculas y minúsculas. Luego se debe ingresar un real positivo, que indicará cuántas unidades de tiempo dura la simulación. Si no se ingresa este comando, quedará el tiempo de simulación por defecto que es 1. El simulador notifica en el caso de que el comando no tenga el formato adecuado o si el número ingresado no es positivo. Los comandos que restan por analizar tienen el mismo formato. Estos comandos son los que indican cuáles de los algoritmos presentados en esta documentación se utilizarán en el simulador y cuáles no. Se ingresan un comando por línea

```
PushAlgorithm 0

DuplicatesCopies 1

PullAlgorithm 0
```

Nuevamente se deben respetar mayúsculas y minúsculas en el nombre del comando, dejar un único espacio en blanco, y luego incluir un 1 o un 0. Se activará el algoritmo si se ingresa un 1 y no se implementará este si se ingresa un 0. Si no incluyen estos comandos, la configuración predeterminada es la mostrada en el ejemplo. El simulador informará si alguno de estos comandos no tiene el formato correcto, o si no se ingresó una opción que no sea 0 o 1. Por último se analiza el comando referente a la barrera de Inercia, la cual sólo tiene efecto si está activado el algoritmo *Push/Pull*. El comando es de la forma

```
BarreraInercia 1
```

Al igual que en los casos anteriores se deben respetar mayúsculas y minúsculas para el nombre del comando, el cual viene acompañado por un entero positivo que será el valor de la barrera de inercia. En el caso de que no se ingrese este comando, el valor por defecto de dicha variable es 1. El simulador notifica en caso de que alguna de las variables no tenga el formato adecuado, o si el valor que se ingresa para la barrera de inercia no es positivo.

Un ejemplo de este bloque podría ser:

```
begin Simulacion
Tiempo 5000
PushAlgorithm 0
PullAlgorithm 0
DuplicateCopies 1
BarreraInercia 1
```

```
end Simulacion
```

Si se desea más información de este bloque se puede recurrir a la sección 3.4.3 del capítulo 3.

En conclusión la entrada al simulador tendría la siguiente estructura:

```
begin Topologia
Nodo 0
Nodo 1
Enlace 1 0 0.05
Nodo 2
Enlace 2 1 0.05
Nodo 3
Enlace 3 2 0.05
Nodo 4
Enlace 4 3 0.05
end Topologia
begin Aplicaciones
listaArchivos 10
Custodian 0 RandomList 10
Cache 1 1 LRU
Cache 2 1 LRU
Cache 3 1 LRU
Cache 4 1 LRU
Cliente 3 4 1.0 0.8 OrderList 10
end Aplicaciones
begin Simulacion
Tiempo 5000
PushAlgorithm 0
PullAlgorithm 0
DuplicateCopies 1
BarreraInercia 1
end Simulacion
```

Ahora asumamos que este archivo de entrada se almacenó como `input.txt` en el directorio `C:\Simulacion` Para realizar la simulación lo primero que se debe hacer es guardar el archivo Simulador.jar, que se encuentra en el medio óptico entregado, en algún directorio de su dispositivo. A modo de ejemplo se almacenará en disco C:

A continuación se debe ejecutar en la consola, estando en el directorio donde se almacenó el archivo `SimuladorCDN.jar` el siguiente comando

```
java -jar SimuladorCDN.jar  :ubicacionEntrada:
```

En nuestro caso se debe ingresar `C:\Simulacion\input.txt`, que es la dirección del archivo de texto que se utilizará como entrada en la simulación, en lugar de `:ubicacionEntrada:`.

IMPORTANTE: La dirección que se ingresa debe ser el *path* **absoluto** y también se debe tener en cuenta que este archivo tenga permiso de lectura. Además el directorio donde se encuentra el archivo de entrada debe tener permiso de escritura, puesto que es allí donde se generarán los archivos de texto con las medidas realizadas.

Si el ingreso de los datos es correcto, se podrán observar en el directorio de la entrada cuatro archivos de texto nuevos. Estos archivos de texto llevan los nombres `Cliente`, `Delay`, `ContenidosCache` y `SalidaCache`.

IMPORTANTE: Si ya existía un archivo de texto con alguno de estos nombres en el directorio de la entrada, este será sobrescrito.

Si todo salió bien, se debe ver en la consola, el porcentaje de avance de la simulación. Al finalizar la simulación se imprimirá por pantalla, la lista de contenidos por los que pidieron los clientes. Esto es de utilidad cuando se ingresa el orden al azar, para saber cuál fue el resultado de los sorteos.

Finalmente en el mismo directorio donde se encuentra la entrada en nuestro ejemplo (`C:\Simulacion`), se crearon los 4 archivos de texto descriptos en la sección Salida del capítulo Simulador.