Universidad Zaragoza
1542

Trabajo Fin de Grado

Análisis de prestaciones de un sistema de control distribuido COSME

Autor

Julián Guillén Ros

Director

Carlos Catalán Cantero

Codirector

Félix Serna Fortea

Universidad de Zaragoza / Escuela Universitaria Politécnica de Teruel

2015

Análisis de prestaciones de un sistema de control distribuido COSME

Resumen

El siguiente trabajo constituye un proyecto de análisis de prestaciones de un sistema distribuido COSME.

COSME (*COntrol System and Modeling Enviroment*), es una plataforma software de control industrial, basada en el estándar IEC 61499, que permite diseñar, construir y ejecutar aplicaciones de control distribuido.

El objetivo es optimizar la parte referida a la comunicación de COSME en modo distribuido. En la cual intervienen dos elementos, La Pasarela y la red. Para poder optimizar la Pasarela, implementada en Java, es necesario realizar *profiling* sobre el código de la misma, conocer el funcionamiento interno y las posibles configuraciones de la JVM (*Java Virtual Machine*). Para poder optimizar la parte referida a la red se debe conocer que posibilidades ofrece en términos de QoS (*Quality of Service*) el *switch* que la interconecta.

Las pruebas para determinar la configuración óptima tanto del *switch* como de la JVM se han realizado sobre una plataforma de pruebas. Dicha plataforma está compuesta por cuatro Raspberrys y un *switch* comunicados mediante una red *Ethernet*, donde cada Raspberry ejecuta una aplicación COSME llamada Sumador Distribuido.

El estudio realizado en la plataforma de pruebas se ha enfocado a la JVM, al *switch* y al código de la Pasarela. Las pruebas han consistido en la medición de los tiempos de ejecución y *profiling* sobre COSME.

La JVM se ha ejecutado con diferentes configuraciones para determinar cuales ofrecen mejor tiempo de ejecución. En cada configuración se ha elegido una política de recolección de basura, un umbral de compilación y unos parámetros de rendimiento diferentes.

Se ha cambiado la política de recolección de basura en la JVM para determinar cómo afecta cada una de ellas a la "`función normal`" de COSME ejecuta bajo restricciones de tiempo real.

Se ha incrementado gradualmente la carga de trabajo en la "`función normal`", que se ejecuta bajo tiempo real, para determinar cómo afecta al tiempo de ciclo de COSME distribuido.

En la parte referida a la red, se ha configurado la QoS del *switch* para determinar cómo afecta a la latencia de red. Las pruebas se han realizado con y sin carga de trabajo en el *switch* y activando y desactivando la QoS en cada caso, dando más prioridad a los puertos dónde estén conectadas las Raspberrys.

Con el *progiling* sobre el código de la Pasarela se ha determinado si existen cuellos de botella y creación anómala de objetos, para a continuación aplicar las optimizaciones necesarias.

Contenido

1. Introducción

1.1. Objetivos

Objetivo general:

Analizar y optimizar las diferentes partes de un sistema distribuido COSME, donde el aspecto relevante es la comunicación.

Objetivos específicos:

1. Entender el funcionamiento de un sistema distribuido COSME.
2. Poner en marcha un sistema distribuido COSME.
3. Entender los diferentes tipos de recolectores de basura de la JVM de Java.
    a. Diseñar y realizar las pruebas para determinar que recolector de basura es el más óptimo.
4. Entender los diferentes tipos de parámetros de rendimiento de la JVM de Java.
    a. Diseñar y realizar las pruebas para determinar que parámetros de rendimiento son los más óptimos.
5. Determinar cómo los diferentes tipos de recolectores de basura afectan al tiempo de ejecución de la "`función normal`" de COSME.
6. Determinar como el tiempo de ejecución de la "`función normal`" de COSME afecta al tiempo de ejecución del ciclo distribuido.
7. Determinar la configuración óptima del *switch* para tener un tiempo de latencia de red mínimo y estable.
8. Realizar *profiling* sobre COSME para detectar cuellos de botella y anomalías en la creación de objetos.

1.2. Plataforma COSME

COSME (*COntrol System and Modeling Enviroment*), es una plataforma software de control industrial desarrollada en la EUPT, basada en el estándar IEC 61499, que permite diseñar, construir y ejecutar aplicaciones de control distribuido.

COSME está basada en redes de bloques funcionales (FB), cada bloque representa un tipo de componente tanto software como hardware.

Los FB permiten separar las diferentes partes de una aplicación y así poder reutilizarlos. En la Figura 1, se puede apreciar la estructura de un bloque funcional.

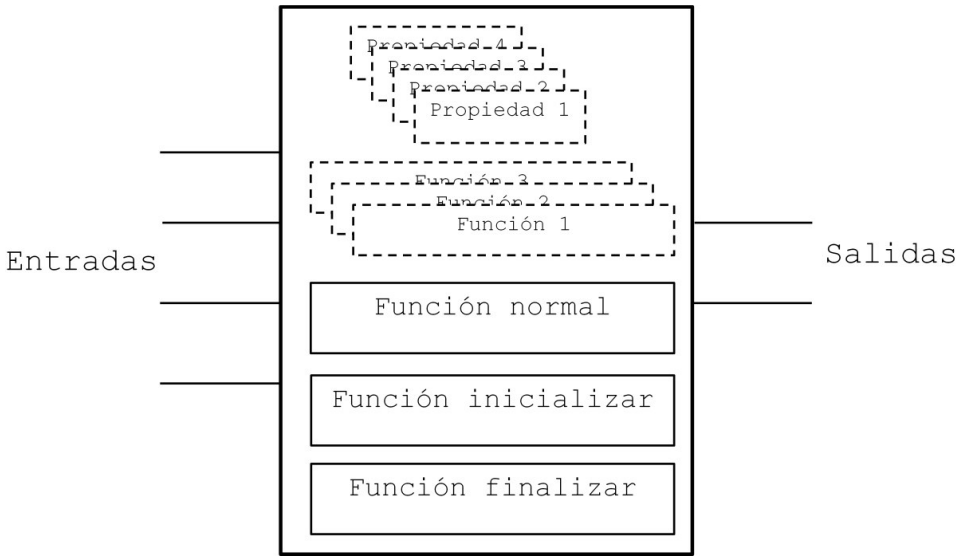

Figura 1: Bloque Funcional (FB)

Los bloques se agrupan formando redes, y se ejecutan según la secuencia definida por el diseñador, Figura 2.

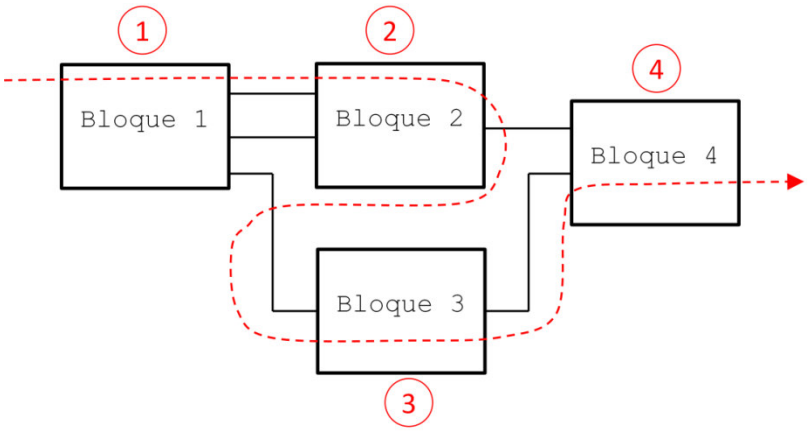

Figura 2: Orden de ejecución de los FB

COSME se puede dividir en dos partes, la parte de ***runtime*** y la parte de **comunicación**.

1.2.1 Runtime

El *runtime* está implementado en C y se encarga de ejecutar de forma periódica las funciones de cada FB.

Un FB está compuesto básicamente por *n* entradas, *n* salidas, y cuatro funciones; "`función normal`, `función normal NoRT`, `función inicializa` y `función finaliza`".

La "`función normal`**"** es una función periódica que se ejecuta bajo la restricción de tiempo real, es decir, cuando esta función tenga el turno para ejecutarse tendrá prioridad sobre todas las demás funciones. Hasta que la función no termine, el sistema operativo no podrá dedicar tiempo a otros procesos. El ciclo de ejecución de esta función se especifica con la variable `SISTEMA.tiempo_ciclo_ms`.

La "`función normal NoRT`**"** es también una función periódica pero no se ejecuta en tiempo real por lo tanto tiene menos prioridad que la "`función normal`". El tiempo de periodo se determina con la variable `SISTEMA.tciclo_E_CYCLE`.

La "`función inicializa`**"** se ejecutan un sola vez una al inicio de la aplicación y la "`función finaliza`**"** al final de la misma.

El *runtime* también se encarga de enviar a la Pasarela el valor de las variables que el diseñador de aplicaciones COSME ha definido. Estas variables pueden ser agrupadas en lo que se denomina Cestas, con un periodo de refresco determinado.

1.2.2. Comunicación

La comunicación en COSME se realiza a través de cadenas de texto, llamados Telegramas, con una nomenclatura concreta. Para que COSME pueda trabajar de **modo distribuido**, se necesitan dos Telegramas:

- `Publish`: es enviado por el runtime a la Pasarela para comunicarle que las variables que lo componen son para la siguiente Raspberry.

- `Exec_seq`: es enviado por la Pasarela a la siguiente Raspberry con las variables del Telegrama `publish`. Cuando la Raspberry recibe este telegrama comienza la ejecución de la "`función normal NoRT`".

En la Figura 3, se puede observar una secuencia distribuida de COSME.

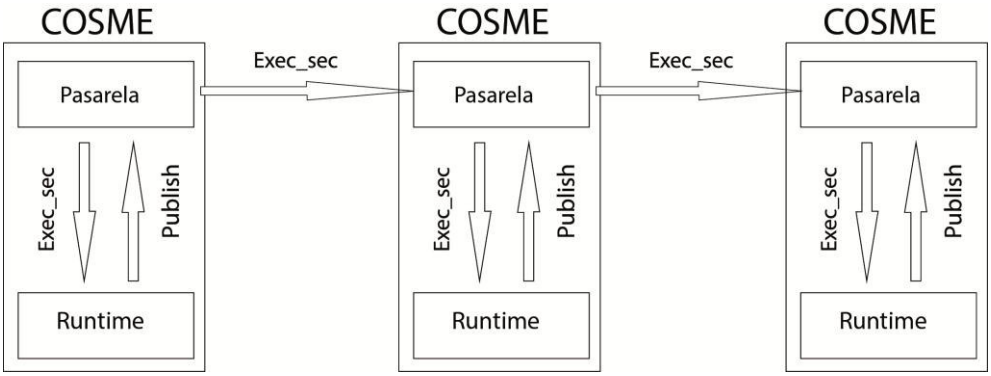

Figura 3: Comunicación en COSME distribuido

En la comunicación de COSME intervienen dos elementos, la Pasarela y la red.

Pasarela

La Pasarela está implementada en Java, se encarga de comunicar el *runtime* con el exterior y viceversa, a través de dos FIFOS, una de escritura y otra de lectura.

La Pasarela acepta y establece conexiones, a través de la librería Arcadio, bien con otras instancias de COSME o bien con otros clientes Arcadio. Además de gestionar los Telegramas entre ellos y el *runtime*.

La librería Arcadio se encarga del envío y la recepción de Telegramas, además de mantener activa la conexión mediante el envío periódico de un Telegrama de tipo ping.

Las posibles optimizaciones a realizar pueden ir referidas al código en sí de la Pasarela o a la JVM.

- Si son referidas al código de la Pasarela, se utilizará la herramienta *Profiler* de NetBeans para detectar cuellos de botella o creación anómala de objetos.

- Si son referidas a la JVM, se deberá conocer el funcionamiento de la JVM (cómo está distribuida la memoria, las diferentes políticas del recolector de basura y los parámetros de JVM). Tal funcionamiento se explica en el apartado 1.4. Gestión de los parámetros de la JVM.

Red

Se trata de una red *Ethernet* interconectada mediante un *switch*. Con el objetivo de optimizar el funcionamiento del *switch*, se debe conocer que posibilidades ofrece en términos de QoS (Quality of Service). En el apartado 1.4. Configuración del *switch* se explica más en detalle esta característica.

1.3. Configuración de la ejecución de JVM

La configuración de la JVM se gestiona a partir de los parámetros que se le indican al ejecutarla. De estos parámetros dependerá el rendimiento final. Los parámetros pueden ir relacionados con la asignación de memoria, con el *garbage collector* o con el rendimiento.

1.3.1. Gestión de la memoria en JVM

El *heap* es la zona de memoria dinámica que almacena los objetos que se crean, en un principio tiene un tamaño fijo asignado por la JVM, pero según sea necesario se va añadiendo más espacio. Está dividido en tres partes *Young Generation*, *Old Generation* y *Permanent Generation*, como se puede apreciar en la Figura 4*:*

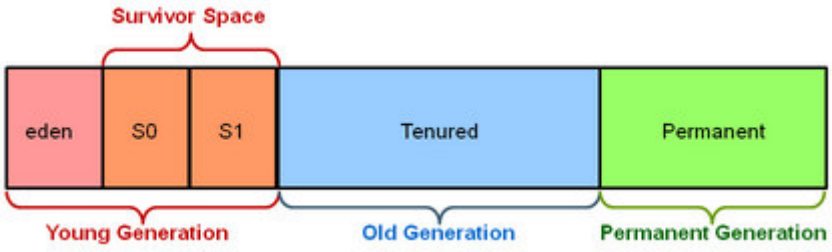

Figura 4: Estructura del *heap* en HotSpot

- ***Young Generation***:
    - *Eden Space*: Esta es el área inicial donde se inicializan la mayoría de los objetos.
    - *Survivor Space*: En esta área se almacenan los objetos que han sobrevivido a la recolección de basura en el *Eden*. En general esta área está dividida en dos partes *From* y *To*.
- ***Old Generation***:
    - *Tenured Space*: Contiene los objetos que han existido por un largo tiempo y que han pasado por el *Survivor Space*.
- ***Permanent Generation***: contiene metadatos requeridos por la JVM para describir las clases y métodos utilizados en la aplicación. El espacio *Permanent* es rellenado por la JVM en tiempo de ejecución basándose en las clases que hay en uso por la aplicación.

Cuando se crea un objeto nuevo en Java con la instrucción *new*, éste inicialmente se encuentra en el espacio *Eden*. Conforme se van ejecutando varios ciclos de recolección de basura o se van creando nuevos objetos, éstos van migrando a través de los espacios de supervivencia, *Survivor Spaces*. Los objetos que sobreviven pasan al espacio *Tenured Space*.

El espacio *Tenured* se utiliza para almacenar objetos que han sobrevivido durante un periodo de tiempo largo. Eventualmente este espacio necesita ser recogido por el recolector de basura.

Parámetros de gestión de memoria en JVM

El tamaño de cada zona de memoria es configurable por el usuario, en la Figura 5, se puede apreciar el parámetro específico para cada zona.

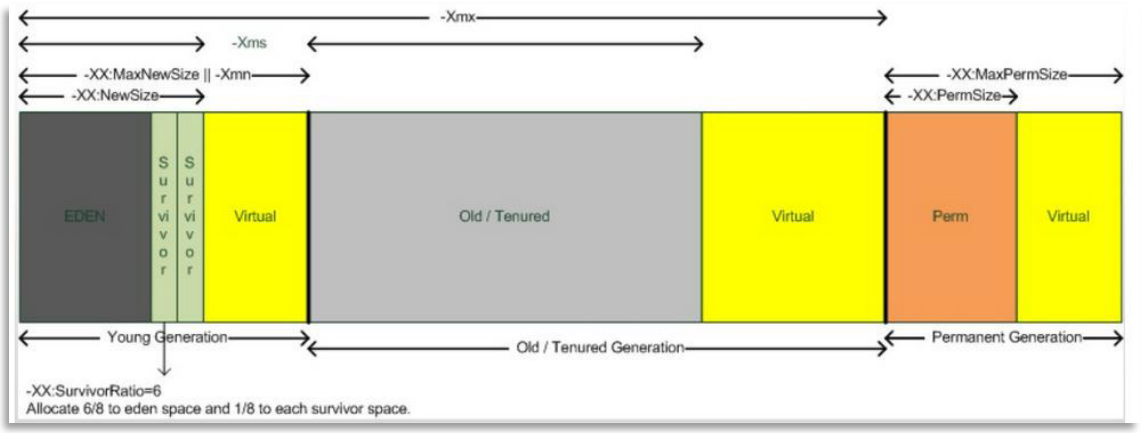

Figura 5: Parámetros de configuración de las zonas de memoria en JVM

`Xms` y `Xmx`:

Indican el tamaño mínimo y máximo del *heap*, ejemplo:

- `-Xms1024M`. Tamaño mínimo del *heap* en 1024 MB.
- `-Xmx1800M`. Tamaño máximo del *heap* en 1800MB.

`NewSize` y `MaxNewSize`

Indican el tamaño mínimo y máximo de la zona *Young Generation*, ejemplo:

- `-XX:NewSize=128M`. Tamaño mínimo de la zona *Young Generation*.
- `-XX:MaxNewSize=256M`. Tamaño máximo de la zona *Young Generation*.

`PermSize` y `MaxPermSize`

Indican el tamaño mínimo y máximo de la zona *Permanent Generation*, ejemplo:

- `-XX:PermSize=128M`. Tamaño mínimo de la zona *Permanent Generation*.
- `-XX:MaxPermSize=256M`. Tamaño máximo de la zona *Permanent Generation*.

`SurvivorRatio`

Indica el ratio entre *Eden* y *Survivor*, ejemplo:

- `-XX:SurvivorRatio=3`

`TargetSurvivorRatio`

Indica que porcentaje se puede llenar la zona *Survivor* antes de mover a *Old Generation,* ejemplo:

- `-XX:TargetSurvivorRatio=50`

1.3.2. Parámetros relacionados con el GC

El recolector de basura, del inglés *garbage collector*, es un proceso automático de baja prioridad que se ejecuta dentro de la JVM. Se encarga de limpiar aquella memoria del *heap* que ya no se utiliza y por tanto, podría ser utilizada por otros programas. Un objeto podrá ser borrado cuando no sea referenciado por otro.

Desde una aplicación Java se puede invocar al recolector de basura con `System.gc`, pero esto no es aconsejable porque cada recolector tiene su propia política y podría afectar al rendimiento de la aplicación. Se puede deshabilitar las llamadas explicitas al recolector de basura con el siguiente parámetro de JVM, `-XX:-DisableExplicitGC`.

Para saber cuándo se ejecuta el GC se debe utilizar el parámetro `-Xloggc:gc.txt` al ejecutar la aplicación Java. Este parámetro guarda información sobre la ejecución del recolector de basura en el fichero `gc.txt`.

Existen dos tipos de recolecciones de basura:

- **Recolección de basura menor**: se ejecuta cuando se llena la zona *Eden* o antes de incrementar su espacio.

- **Recolección de basura mayor o completa**: se ejecuta cuando se llena la zona *Tenured* o antes de incrementar su espacio.

Políticas de funcionamiento del garbage collector

Existen cuatro políticas de *garbage collectors* en Java 7.0:

- Serie
- Paralela
- Concurrente
- Garbage-First (G1)

Serie

La política serie está diseñada para aplicaciones que requieran un *heap* de hasta 100MB, en equipos con un solo procesador. Esta política es la que utiliza por defecto la JVM en las Raspberrys.

Corre en un solo hilo y usa el algoritmo de Copia. Cuando se ejecuta para limpiar la memoria la pausa que realiza es del tipo STW (*Stop The World*), es decir, toda la aplicación se paraliza y no se reactiva hasta que el recolector acaba, como se puede ver en la Figura 6.

Esta política se habilita con la opción de la JVM, `-XX:+UseSerialGC`. Se puede utilizar desde la versión de Java 5.0

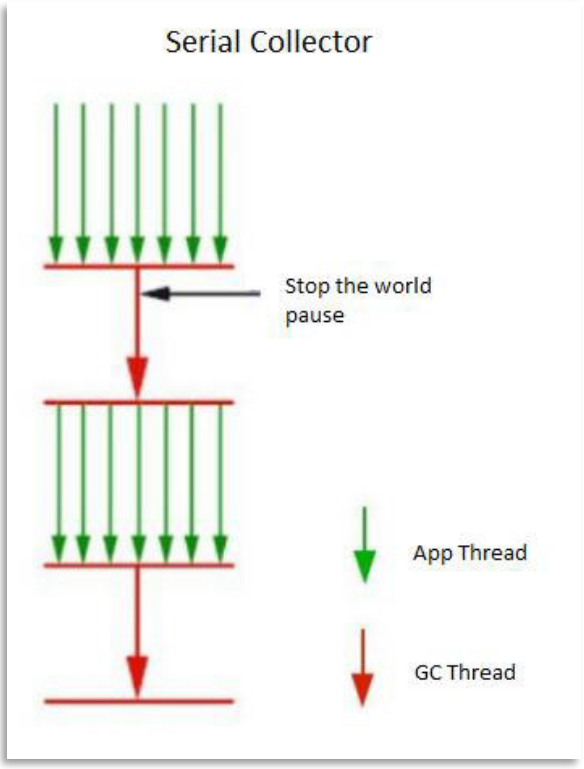

Figura 6: Política de recolección de basura serie

Paralela

La política paralela corre en múltiples hilos y las pausas que realiza son del tipo STW, es decir, toda la aplicación se paraliza.

Esta política se habilita con la opción de la JVM `-XX:+UseParallelGC`.

En una máquina con *N* procesadores la política paralela utiliza *N* hilos para recolectar la basura. Sin embargo, este número se puede ajustar con la opción `-XX:ParallelGCThreads=n`.

Con el comando `-XX:+UseParallelGC`, la política de recolección no compacta la memoria pero se puede activar con la opción `-XX:+UseParallelOldGC`. Esta opción está disponible desde la versión: 1.4.1

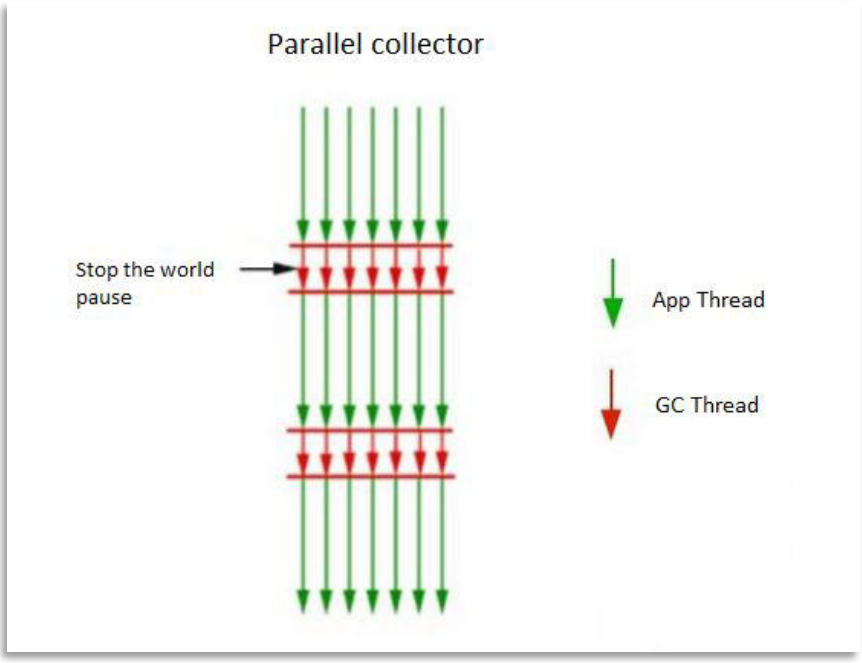

Figura 7: Política de recolección paralela

En la Figura 7, se puede observar cómo el recolector de basura corre en varios hilos y las pausas que realiza son del tipo STW.

Concurrente

La política concurrente está diseñada para aplicaciones que precisen pausas más cortas en la recolección de basura y que corren en máquinas con dos o más procesadores.

Esta política se habilita con la opción de la JVM, `-XX:+UseConcMarkSweepGC`, disponible desde la versión: 1.4.1

Su funcionamiento es el siguiente:

1. Marcado inicial: En esta fase se produce una pequeña pausa del tipo STW, que paraliza toda la aplicación, donde todos los objetos alcanzables (vivos) son marcados.
2. Marcado Concurrente: busca los objetos vivos mientras la aplicación se ejecuta en otro hilo.
3. Remarcado: busca los objetos que no fueron encontrados durante la fase 2.
4. Barrido concurrente: elimina los objetos que son inalcanzables. Y no compacta los objetos alcanzables.
5. Restablecimiento: se prepara para la siguiente ejecución limpiando las estructuras usadas.

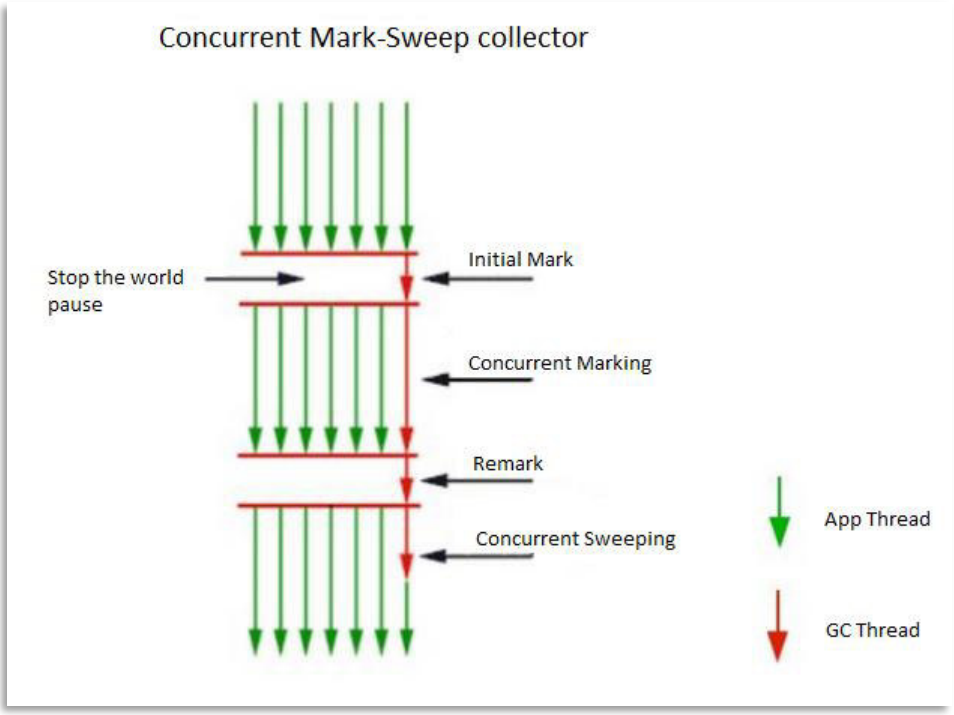

Figura 8: Recolector de basura CMS

En la Figura 8, se puede observar el funcionamiento de la política de recolección concurrente.

Garbage-First G1

La política de recolección de basura de Garbage-First (G1) es la más reciente, diseñada para reemplazar al recolector CMS. G1 es una política de estilo servidor, pensado para máquinas multi-procesador con grandes cantidades de memoria.

Este recolector está disponible desde la versión 1.7.4 de Java.

G1 es un recolector compactador. Compactar es un proceso por el cual los objetos vivos se mueven sobre la memoria libre hacia el final del *heap* de manera que se logra un área contigua de memoria libre. Esto es importante para las aplicaciones que se ejecutan durante mucho tiempo porque es inevitable que el *heap* se fragmente con el paso del tiempo. G1 evita los potenciales problemas de la fragmentación.

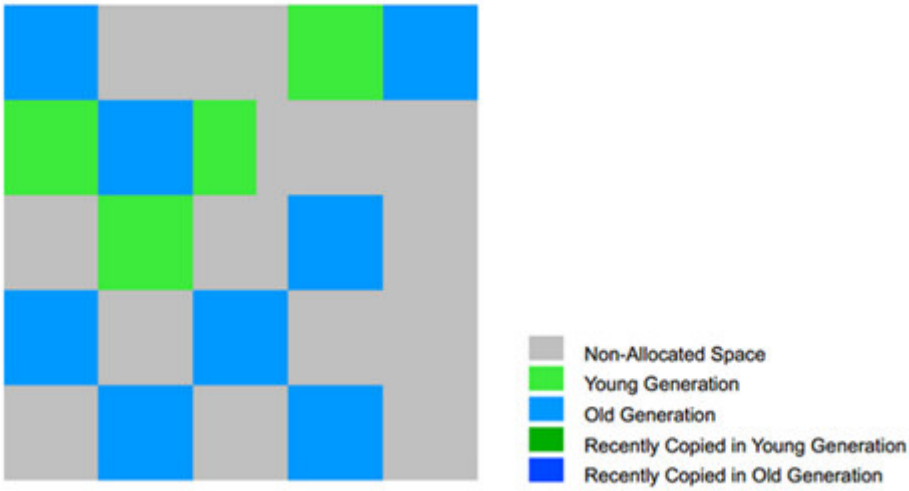

Figura 9: Distribución del *heap* con G1

El *heap* se divide en un conjunto de regiones de igual tamaño, cada una en un intervalo contiguo de memoria virtual. Estos conjuntos de regiones usan el mismo sistema de Generaciones que las anteriores políticas de recolección, pero no hay un tamaño fijo para ellas. Esto proporciona una mayor flexibilidad en el uso de memoria. En la Figura 9, se puede ver un ejemplo de la distribución de las Generaciones con el recolector G1.

1.3.3. Parámetros relacionados con el rendimiento de la JVM

Server y Client

La JVM HotSpot incluye dos modos de ejecución, cliente y servidor. Las dos soluciones comparten el código base de ejecución, pero utilizan diferentes compiladores que se adaptan a las características de rendimiento de los clientes y servidores.

Aunque el modo servidor y el modo cliente son similares, cada uno es específico para un propósito.

El modo servidor está diseñado para maximizar la velocidad de operación y destinado a la ejecución de aplicaciones de servidor de larga duración. El compilador de este modo aplica al código optimizaciones complejas, por eso la puesta en marcha de aplicaciones en modo servidor requieren más tiempos y memoria.

Pero el parámetro –`server` no está disponible para la arquitectura ARMv6 que es la que está implementada en las Raspberrys utilizadas para este proyecto. Solo está disponible para versiones ARMv7+ por lo tanto no se puede hacer uso de él.

El modo cliente, al compilar el código no ejecuta optimizaciones complejas, por lo tanto se requiere menos tiempo para analizar y compilar el código. Esto significa que la aplicación cliente se puede poner en marcha más rápido y requiere un menor consumo de memoria.

El modo cliente es el que se ejecuta por defecto en las Raspberrys.

CompileThresHold

Java es independiente de la plataforma, significa que programas escritos en Java pueden ejecutarse igualmente en cualquier tipo de hardware. De ahí proviene el famoso eslogan *write once, run anywhere*.

En las versiones actuales de la JVM se ejecutan los programas combinando la interpretación de *bytecodes* y la compilación JIT (*Just-In-Time*).

La JVM analiza *bytecodes* a medida que se van interpretando e identifica los *bytecodes* que se ejecutan con más frecuencia. Estos *bytecodes* son traducidos a lenguaje máquina correspondiente por el compilador JIT. Cuando la JVM encuentre de nuevo estos puntos activos, ejecutará directamente el código máquina disminuyendo el tiempo de ejecución del programa y aumentando el rendimiento.

En la JVM HotSpot, se puede definir el número de invocaciones antes de que un trozo de código se compile al lenguaje máquina correspondiente. Ese número se define con el parámetro `CompileThreshold`. Por defecto en aplicaciones cliente este parámetro tiene un valor de 1500.

Ejemplo: `java -XX:CompileThreshold=10000 –jar programa.java`

AggressiveOpts

El parámetro `AggressiveOpts`, aplica las mejoras que se esperan sean por defecto en siguientes versiones de la JVM. Introducido en la versión 5.0.

Ejemplo: `java -XX:+AggressiveOpts -jar programa.java`

UseFastAccessorMethods

El parámetro `UseFastAccessorMethods`, utiliza versiones optimizadas del método *get().*

Ejemplo: `java -XX:+UseFastAccessorMethods -jar programa.java`

UseStringCache

El parámetro `UseStringCache`, permite el almacenamiento en caché de las cadenas más usadas.

Ejemplo: `java -XX:+UseStringCache -jar programa.java`

UseCompressedStrings

El parámetro `UseCompressedStrings`, utiliza vectores de bytes, *byte[]*, cuando una cadena de texto, *String*, puede representarse como ASCII puro. Introducido en Java 6 *update* 21. Pero fue eliminado en la versión de Java 7.0, por lo tanto no se puede hacer uso de él.

Ejemplo: `java -XX:+UseCompressedStrings -jar programa.java`

1.4. Configuración del *switch*

Una de las partes que intervienen en la comunicación de COSME es el *switch*. Con el objetivo de optimizar este elemento hay que conocer que opciones ofrece el *switch* Cisco SLM 2008 en términos de QoS (*Quality of Service*). A continuación se describen las posibilidades de configuración que presta.

La configuración de la QoS en el *switch*, consta de tres modos, basada en los puertos, en el estándar 802.1p y en *DSCP* (*Differentiated Services Code Point*).

- Basada en los **puertos**: permite establecer cuatro niveles de prioridad, *low, normal, médium y high,* para cada puerto. Siendo *low* la prioridad más baja y *high* la más alta.

- **Estándar 802.1p**: proporciona priorización de tráfico, por medio de 3 bits en el campo prioridad de usuario (*user_priority*) de la cabecera IEEE 802.1Q, asignando a cada paquete un nivel de prioridad entre 0 y 7. Solo puede ser soportado por una red LAN.

- **DSCP**: hace referencia al segundo byte en la cabecera de los paquetes IP que se utiliza para diferenciar el nivel de servicio de cada paquete.

El modo de encolar los paquetes, es decir, la manera con la que le *switch* reparte los paquetes por los diferentes puertos puede configurarse como Prioridad Estricta (*Strict Priority*) o WRR (*Weighted Round-Robin*).

- **Prioridad Estricta**: transmite primero los paquetes de las colas con prioridad más alta antes que las colas con prioridad menor.

- **WRR**: comparte le ancho de banda de los puertos de salida, usando varios pesos 1, 2, 4 y 8 asignados a 4 colas (*low, normal, médium, high*).
  Este planificador envía paquetes de cada cola en turnos, siendo la cantidad en función de los pesos configurados.
  WRR previene de la inanición a la que podría dar lugar la Prioridad Estricta en los casos en que a la cola de mayor prioridad llegara una gran cantidad de tráfico. Mediante este planificador no existe el concepto de que una cola sea de mayor prioridad que otra, todas las colas tienen ocasión de enviar paquetes aunque tengan un peso pequeño.

Para realizar las pruebas se ha elegido la **QoS basada en los puertos**, ya que en los demás casos se requieren cabeceras adicionales en los datagramas.

2. Plataforma de pruebas

El análisis de prestaciones se ha realizado sobre un entorno distribuido, consta de cuatro ordenadores de placa reducida Raspberry Pi, de ahora en adelante Raspberry, comunicados por una red *Ethernet* mediante un *switch* Cisco SLM2008.

El *switch* permanecerá conectado a un *router*, por donde se accederá a las Raspberrys desde el ordenador con el que se supervisará el trabajo.

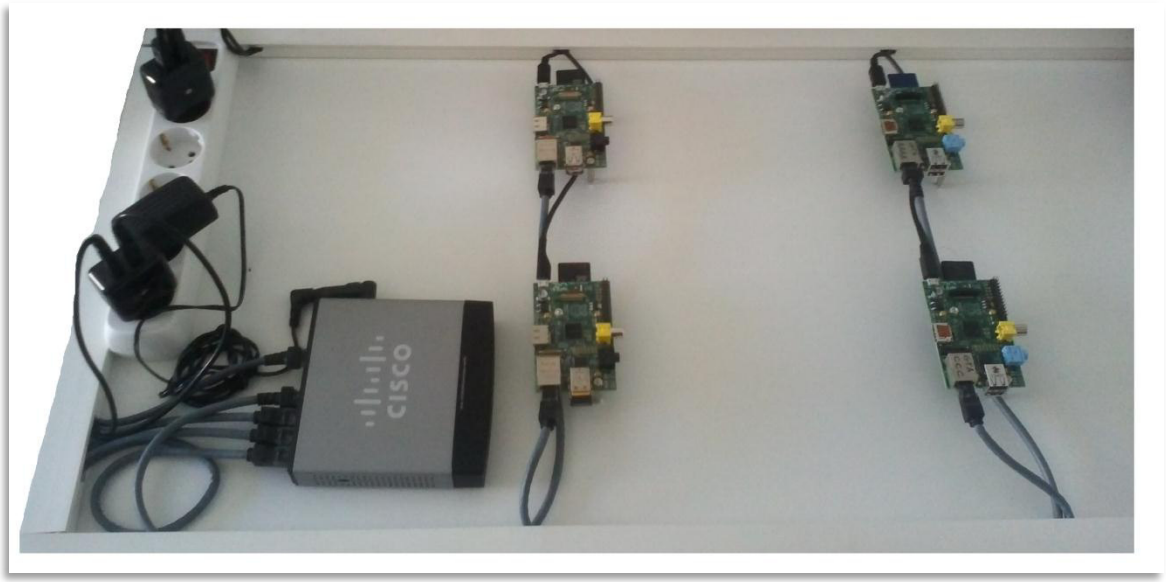

Figura 10: Plataforma de pruebas

En la Figura 10 se puede apreciar la plataforma compuesta del *switch* y las cuatro Raspberrys.

Se ha trabajado con el sistema operativo Raspbian, instalado en cada Raspberry, y parcheado con Xenomai, para dotarle de características de tiempo real.

A su vez, sobre COSME se ha ejecutado una aplicación sintética, llamada Sumador Distribuido, diseñada especialmente para este proyecto.

La versión de Java utiliza ha sido la 1.7.0_40 con la JVM HotSpot.

2.1. Aplicación sintética

La funcionalidad de la aplicación sintética, Sumador Distribuido, consistirá en que cada Raspberry pase a la siguiente un valor numérico. Cada una tendrá que actualizar una posición de ese valor.

La Raspberry1 actualizará las unidades de millar, la Raspberry2 las centenas, la Raspberry3 las decenas y la Raspberry4 las unidades. Como se puede apreciar en la siguiente figura.

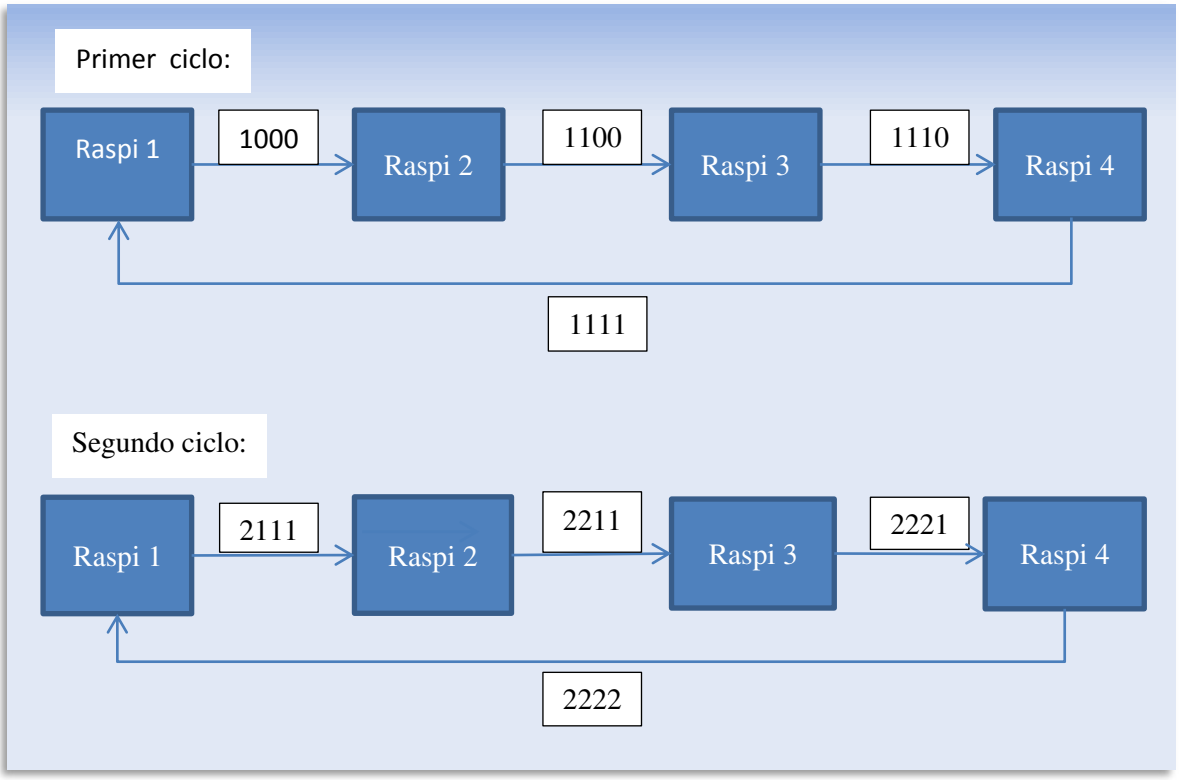

Figura 11: Esquema de funcionamiento de Sumador Distribuido

En la Figura 1*1*, se puede apreciar un ejemplo del funcionamiento de la aplicación sintética Sumador Distribuido.

3. Análisis de prestaciones

3.1. Medidas de los tiempos de ejecución

El análisis de los tiempos de ejecución de la plataforma COSME, se ha llevado a cabo midiendo los tiempos de ejecución de la Pasarela, cambiando los parámetros de la JVM en cada ejecución.

Tales parámetros van referidos a la máquina virtual de Java, por lo tanto solo afectan a la Pasarela, y han sido los siguientes:

- Referidos al recolector de basura
- Referidos al rendimiento
- Referidos a la memoria

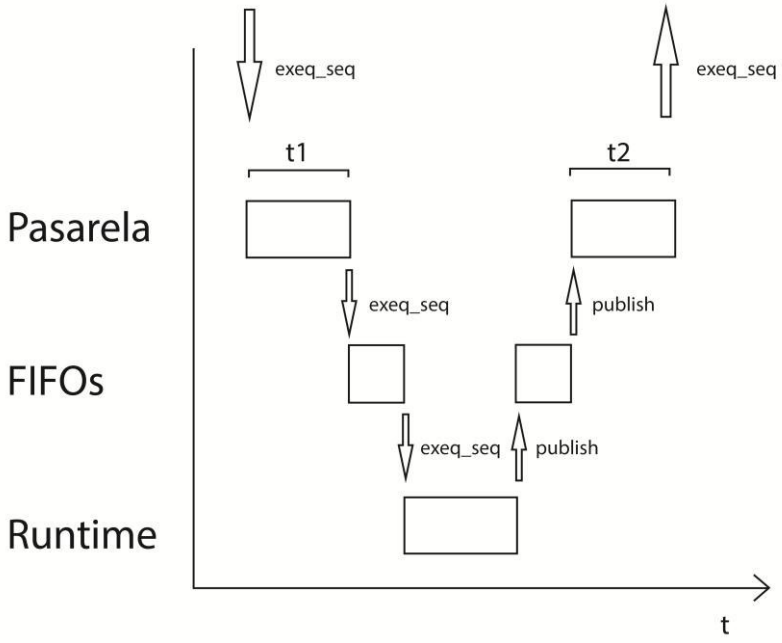

Figura 12: Esquema de tiempos a medir

Más concretamente como se puede apreciar en la Figura 12, los tiempos que se han medido han sido **t1**, tiempo de ejecución desde que llega el telegrama `exec_seq` a la Pasarela, hasta que lo mete en la FIFO para enviárselo al *runtime*, y **t2**, tiempo empleado desde que la Pasarela recibe el telegrama `publish` desde el *runtime* hasta que envía el telegrama `exec_seq` a la siguiente Raspberry.

Para realizar las mediciones se ha creado una clase específica en Java, ver Anexo 1. Y un *shell script*, ver Anexo 2, para poner en funcionamiento COSME cada cierto tiempo con diferentes parámetros de la JVM al ejecutar la Pasarela.

3.1.1. Recolector de basura y umbral de compilación

En esta sección se muestran los tiempos de ejecución obtenidos, combinando las diferentes tipos de políticas de recolección de basura y el umbral de compilación.

Cada configuración de JVM se ha ejecutado durante 5 minutos con un tiempo de ciclo, `SISTEMA.tciclo_E_CYCLE`, de 50 ms. Con lo que se han obtenido 5900 muestras de **t1** y **t2**. Los cuales se han sumado para obtener el tiempo global de ejecución de la Pasarela y a continuación calcular la media, la desviación típica y el valor menos desfavorable al 95% y el valor al 100%.

El objetivo de esta prueba es hallar que política de recolección de basura y que valor de `Compilethreshold` obtienen el menor valor para la media, la desviación típica y el valor menos desfavorable al 95% y al 100%.

El criterio de clasificación para determinar la configuración óptima, se establecido en primer lugar por el menor valor al 95%, en segundo lugar por el menor valor de la media y en tercer lugar por el menor valor de la desviación típica.

Estos datos se corresponden al archivo del Anexo 3: *TiemposDiferentesGCyCompilacion.txt*

Recolector de basura concurrente y umbral de compilación

En este apartado se comparan los tiempos obtenidos entre una ejecución de COSME con el recolector de basura concurrente, parámetro `ConcMarkSweepGC`, y el umbral de compilación, parámetro `CompileThreshold`, con los valores 1, 100, 1000, 10000 y 100000.

Configuraciones de JVM:

1. `UseConcMarkSweepGC, CompileThreshold=1`
2. `UseConcMarkSweepGC, CompileThreshold=100`
3. `UseConcMarkSweepGC, CompileThreshold=1000`
4. `UseConcMarkSweepGC, CompileThreshold=10000`
5. `UseConcMarkSweepGC, CompileThreshold=100000`

Configuraciones JVM	Media	Desviación Típica	95%	100%
1	15,11	23,84	49,04	659,96
2	8,49	8,24	20,48	337,81
3	7,66	6,74	12,10	140,66
4	9,4	4,37	14,26	188,13
5	17,53	12,37	45,03	207,81

Tabla 1: Tiempos de ejecución GC concurrente (valores en ms)

En la Tabla 1, se puede apreciar que la configuración más óptima, es decir, la que ha dado un tiempo de ejecución menos desfavorable al 95% es la política de recolección concurrente y el umbral de compilación (`CompileThreshold`) a 1000 con valor de 12,10 ms. La media, la

desviación típica y el 100% de dicha configuración también son los valores más pequeños de esta prueba.

La configuración que peores resultados ha obtenido, es decir, que ha dado un tiempo de ejecución más desfavorable al 95% es la política de recolección concurrente y el umbral de compilación (`CompileThreshold`) con un valor de 100000.

Recolector de basura paralelo y umbral de compilación

En este apartado se comparan los tiempos obtenidos, entre una ejecución de COSME con el recolector de basura paralelo, parámetro `ParallelGC`, y el umbral de compilación, parámetro `CompileThreshold` con los valores 1, 100, 1000, 10000 y 100000.

Configuraciones de JVM:

6. `UseParallelGC, CompileThreshold=1`
7. `UseParallelGC, CompileThreshold=100`
8. `UseParallelGC, CompileThreshold=1000`
9. `UseParallelGC, CompileThreshold=10000`
10. `UseParallelGC, CompileThreshold=100000`

Configuraciones JVM	Media	Desviación Típica	95%	100%
6	8,44	11,28	16,20	304,08
7	8,16	8,18	16,75	199,64
8	9,9	12,16	21,58	359,95
9	13,18	10,81	25,42	195,79
10	22,31	15,19	37,06	214,26

Tabla 2: Tiempos de ejecución GC paralelo (valores en ms)

En la Tabla 2, se puede apreciar que la configuración más óptima, es decir, la que ha dado un tiempo de ejecución menos desfavorable al 95% es la política de recolección paralela y el umbral de compilación (`CompileThreshold`) a 1 con valor de 16,2 ms.

La configuración que peores resultados ha obtenido, es decir, que ha dado un tiempo de ejecución más desfavorable al 95% es la política de recolección paralela y el umbral de compilación (`CompileThreshold`) con un valor de 100000.

Recolector de basura paralelo compactador y umbral de compilación

En este apartado se comparan los tiempos obtenidos, entre una ejecución de COSME con el recolector de basura paralelo compactador, parámetro `ParallelOld`, y el umbral de compilación, parámetro `CompileThreshold` con los valores 1, 100, 1000, 10000 y 100000.

Configuraciones de JVM:

11. `UseParallelOldGC, CompileThreshold=1`
12. `UseParallelOldGC, CompileThreshold=100`
13. `UseParallelOldGC, CompileThreshold=1000`
14. `UseParallelOldGC, CompileThreshold=10000`
15. `UseParallelOldGC, CompileThreshold=100000`

Configuraciones JVM	Media	Desviación Típica	95%	100%
11	7,91	9,59	14,39	303,74
12	7,94	7,47	15,85	197,01
13	9,44	9,76	20,63	289,26
14	12,99	14,68	24,77	503,38
15	21,58	16,21	36,77	352,05

Tabla 3: Tiempos de ejecución GC paralelo compactador (valores en ms)

En la Tabla 3, se puede apreciar que la configuración más óptima, es decir, la que ha dado un tiempo de ejecución menos desfavorable al 95% es la política de recolección paralela compactadora y el umbral de compilación (`CompileThreshold`) a 1 con valor de 14,39 ms.

La configuración que peores resultados ha obtenido, es decir, que ha dado un tiempo de ejecución más desfavorable al 95% es la política de recolección paralela compactadora y el umbral de compilación (`CompileThreshold`) con un valor de 100000.

Recolector de basura serie y umbral de compilación

En este apartado se comparan los tiempos obtenidos, entre una ejecución de COSME con el recolector de basura serie, parámetro `SerialGC` y el umbral de compilación, parámetro `CompileThreshold` con los valores 1, 100, 1000, 10000 y 100000.

Configuraciones de JVM:

16. `UseSerialGC, CompileThreshold=1`
17. `UseSerialGC, CompileThreshold=100`
18. `UseSerialGC, CompileThreshold=1000`
19. `UseSerialGC, CompileThreshold=10000`
20. `UseSerialGC, CompileThreshold=100000`

Configuraciones JVM	Media	Desviación Típica	95%	100%
16	8,41	9,38	16,32	313,47
17	7,31	6,5	13,2	337,97
18	9,61	12,74	19,91	639,29
19	9,91	5,79	17,71	199,85
20	14,46	8,64	28,96	188,02

Tabla 4: Tiempos de ejecución GC serie (valores en ms)

En la Tabla 4, se puede apreciar que la configuración más óptima, es decir, la que ha dado un tiempo de ejecución menos desfavorable al 95% es la política de recolección serie y el umbral de compilación (`CompileThreshold`) a 100 con valor de 13,2 ms.

La configuración que peores resultados ha obtenido, es decir, que ha dado un tiempo de ejecución más desfavorable al 95% es la política de recolección serie y el umbral de compilación (`CompileThreshold`) con un valor de 100000.

Recolector de basura G1GC y umbral de compilación

En este apartado se comparan los tiempos obtenidos, entre una ejecución de COSME con el recolector de basura G1, parámetro Use`G1GC` y el umbral de compilación, parámetro `CompileThreshold` con los valores 1, 100, 1000, 10000 y 100000.

Configuraciones de JVM:

21. `UseG1GC, CompileThreshold=1`
22. `UseG1GC, CompileThreshold=100`
23. `UseG1GC, CompileThreshold=1000`
24. `UseG1GC, CompileThreshold=10000`
25. `UseG1GC, CompileThreshold=100000`

Configuraciones JVM	Media	Desviación Típica	95%	100%
21	8,44	9,02	15,55	294,09
22	7,35	4,4	13,6	180,71
23	10,35	11,28	25,01	200,91
24	18,79	13,82	41,62	228,3
25	16	10,96	32,98	323,04

Tabla 5: Tiempos de ejecución GC G1 (valores en ms)

En la Tabla 5, se puede apreciar que la configuración más óptima, es decir, la que ha dado un tiempo de ejecución menos desfavorable al 95% es la política de recolección G1 y el umbral de compilación (`CompileThreshold`) a 100 con valor de 13,6 ms. La media, la desviación típica y el 100% de dicha configuración también son los valores más pequeños de esta prueba.

La configuración que peores resultados ha obtenido, es decir, que ha dado un tiempo de ejecución más desfavorable al 95% es la política de recolección G1 y el umbral de compilación (`CompileThreshold`) con un valor de 10000.

Análisis de los resultados

En este apartado se comparan los tiempos obtenidos, entre la configuración por defecto y la configuración más óptima de cada uno de los apartados anteriores.

Configuraciones JVM	Media	Desv.Típica	95%	100%
Por defecto	7,76	12,27	19,18	725,19
Concurrente, CompileThreshold = 1000	7,66	6,74	12,10	140,66
Paralelo, CompileThreshold = 1	8,44	11,28	16,20	304,08
Paralelo Compactador, CompileThreshold = 1	7,91	9,59	14,39	303,74
Serie, CompileThreshold = 100	7,31	6,5	13,2	337,97
G1, CompileThreshold = 100	7,35	4,4	13,6	180,71

Tabla 6: Comparativa tiempos de ejecución Pasarela (valores en ms)

En la Tabla 6, se puede observar que el mejor resultado al 95% ha sido para el recolector de basura concurrente y el umbral de compilación, `CompileThreshold=1000`, con un tiempo de 12,10 ms, seguido del recolector de basura serie con un tiempo de 13,2 ms.

Comparando la configuración del recolector de basura concurrente y `CompileThreshold=100` con el modo por defecto, 12,10 ms y 19,18 ms respectivamente, se obtiene una mejora del **36,91%**.

Por lo tanto, la configuración más óptima de la JVM para ejecutar la Pasarela es usando la política de recolección de basura concurrente y el umbral de compilación con un valor de 1000.

De ahora en adelante se llamará Configuración 1 a los parámetros `UseConcMarkSweepGC` y `CompileThresHold=1000`.

3.1.2. Parámetros de rendimiento

Las siguientes tablas muestran los tiempos de ejecución obtenidos, combinando la Configuración 1, parámetros `SerialGC` y `CompileThreshold=1000`, con otros parámetros de rendimiento.

Cada configuración de JVM se ha ejecutado durante 5 minutos con un tiempo de ciclo, `SISTEMA.tciclo_E_CYCLE`, de 50 ms. Con lo que se han obtenido 5900 muestras de **t1** y **t2**. Los cuales se han sumado para obtener el tiempo global de ejecución de la Pasarela y calcular la media, la desviación típica y el valor menos desfavorable al 95% y al 100%.

El objetivo de esta prueba es hallar si la Configuración 1 junto con algún parámetro de rendimiento mejora los tiempos de ejecución de la Pasarela.

El criterio de clasificación para determinar la configuración óptima, se establecido en primer lugar por el menor valor del 95%, en segundo lugar por el menor valor de la media y en tercer lugar por el menor valor de la desviación típica.

Estos datos se corresponden al archivo del Anexo 3: *TiemposParametrosRendimiento.txt*

Configuración 1 y AggressiveOpts

En el siguiente tabla se comparan los tiempos obtenidos, entre una ejecución de COSME por defecto, con la Configuración 1 y con la Configuración 1 con el parámetro `AgressiveOpts`.

Configuraciones de JVM:

1. Por defecto
2. `UseConcMarkSweepGC CompileThreshold=1000`
3. `UseConcMarkSweepGC, CompileThreshold=1000 y AggressiveOpts`

Configuraciones JVM	Media	Desviación Típica	95%	100%
1	7,76	12,27	19,18	725,19
2	7,66	6,74	12,10	140,66
3	9,70	9,29	24,26	200,16

Tabla 7: Comparativa tiempos de ejecución Pasarela (valores en ms)

En el Tabla 7 se puede observar que la ejecución con el parámetro `AggressiveOpts` ha dado una media, una desviación típica, un valor al 95% y al 100% peor que la Configuración 1. Por lo tanto la configuración óptima de la Pasarela seguirá siendo con los parámetros `UseConcMarkSweepGC` y `CompileThreshold=1000`.

Configuración 1 y UseStringCache

En el siguiente apartado se comparan los tiempos obtenidos, entre una ejecución de COSME por defecto, con la Configuración 1 y con la Configuración 1 con el parámetro `UseStringCache`.

Configuraciones de JVM:

1. Valor por defecto
2. `UseConcMarkSweepGC, CompileThreshold=1000`
3. `UseConcMarkSweepGC, CompileThreshold=1000 y UseStringCache`

Configuraciones JVM	Media	Desviación Típica	95%	100%
1	7,76	12,27	19,18	725,19
2	7,66	6,74	12,10	140,66
3	8,53	6,23	15,97	194,49

Tabla 8: Comparativa tiempos de ejecución Pasarela (valores en ms)

En el Tabla 8 se puede observar que la ejecución con el parámetro `UseStringCache` ha dado una media, un valor al 95% y al 100% peor que la Configuración 1. Por lo tanto la configuración óptima de la Pasarela seguirá siendo con los parámetros `UseConcMarkSweepGC` y `CompileThreshold=1000`.

Configuración 1 y UseStringCache, AggressiveOpts, UseFastAccessorMethods

En el siguiente gráfico se comparan los tiempos obtenidos, entre una ejecución de COSME sin parámetros, la Configuración 1 y la Configuración 1 con los parámetros `UseStringCache`, `AggressiveOpts` y `UseFastAccessorMethods`.

Configuraciones de JVM:

1. Valor por defecto
2. `UseConcMarkSweepGC, CompileThreshold=100`
3. `UseConcMarkSweepGC, CompileThreshold=1000, UseFastAccessorMethods`
4. `UseConcMarkSweepGC, CompileThreshold=1000, UseStringCache, UseFastAccessorMethods`
5. `UseConcMarkSweepGC, CompileThreshold=1000, UseStringCache, UseFastAccessorMethods, AggressiveOpts`

Configuraciones JVM	Media	Desviación típica	95%	100%
1	7,76	12,27	19,18	725,19
2	7,66	6,74	12,10	140,66
4	8,84	7,21	15,39	201,90
5	7,83	4,62	15,72	163,24
6	8,79	7,95	17,29	335,97

Tabla 9: Comparativa tiempos de ejecución Pasarela (valores en ms)

En el Tabla 9 se puede observar que la ejecución con la combinación de los parámetros `UseStringCache`, `UseFastAccessorMethods`, `AggressiveOpts` ha dado una media, un valor al 95% y al 100% peor que la Configuración 1. Por lo tanto la configuración óptima de la Pasarela seguirá siendo con los parámetros `UseConcMarkSweepGC` y `CompileThreshold=1000`.

Análisis de los resultados

Los resultados con los parámetros de rendimiento anteriores ha dado peores resultados que con la Configuración 1, por lo tanto, la configuración óptima para ejecutar la Pasarela será con la política de recolección basura concurrente y el umbral de compilación a 1000, parámetros `UseConcMarkSweepGC` y `CompileThreshold=1000`.

3.1.3. Tamaño del *heap*

El propósito de estas medidas es observar cuantas ejecuciones realiza y el tiempo que dedica a cada una de ellas el recolector de basura, variado el tamaño del *heap*.

Estos datos se corresponden al archivo del Anexo 3: *TrazasGC.txt*

Pruebas

Heap de 32 Mb

Con un tamaño de *heap* de 32 megabytes y con los parámetros más óptimos del apartado anterior `CompileThreshold=1000, UseConcMarkSweepGC`. Se ha obtenido las siguientes trazas del recolector de basura durante 60 minutos de ejecución de COSME, como se puede apreciar en la Tabla 10.

Memoria ocupada antes del GC (Kb)	Memoria ocupada después del GC (Kb)	Tiempo dedicado a la recolección (s)
8832	833	0,0827640
9665	795	0,0719320
9627	793	0,0433940
9625	793	0,0230970
9625	794	0,0205180

Tabla 10: Trazas GC

Se ha elegido 32 Mb por aumentar el tamaño del *heap*, ya por defecto la JVM le asigna 6,75 Mb.

Heap de 6,75 Mb

Con un tamaño de *heap* de 6,75 megabytes, que es que la JVM asigna por defecto en la Raspberry, y con los parámetros más óptimos para ejecutar la Pasarela `CompileThreshold=100, UseSerialGC`. Se ha obtenido un número elevado de trazas que en el apartado anterior por lo que se ha decido colocarlas en el Anexo 3 en el fichero *TrazasGC.txt*.

.

Análisis de los resultados

Comparando las trazas del recolector de basura entre una ejecución con el tamaño del *heap* por defecto, 6,75 megabytes, y otra con 32 megabytes, se puede observar que el recolector de basura se ejecuta obviamente menos veces con un tamaño de *heap* mayor pero tarda más tiempo en cada recogida.

Con el tamaño de *heap* por defecto, 6,75 megabytes, el tiempo medio de cada entrada del recolector de basura es de 21,56 ms. Entrando en 30 minutos 48 veces.

Con el tamaño del *heap* de 32 megabytes, el tiempo medio de cada ejecución del recolector de basura es de 41,45 ms. Ejecutándose en 30 minutos 5 veces. Como se puede apreciar en la Tabla 11.

Tamaño del *heap* (Mb)	Ejecuciones del GC	Tiempo medio por entrada (ms)
6,75	48	21,56
32	5	41,45

Tabla 11: Ejecuciones del GC según tamaño del geap

En este proyecto hay requisitos de tiempo real, por lo que lo más óptimo es que el recolector de basura entre más veces pero poco tiempo. Por lo tanto, el tamaño del *heap* conviene que sea ajustado al requerimiento de memoria de la Pasarela.

3.2. Medidas de los tiempos de ejecución de la "función normal"

El propósito de estas medidas es probar si los diferentes tipos de recolectores de basura afectan al tiempo de ejecución de la "función normal". Para ello se ha ejecutado durante 60 minutos con cada recolector.

Los datos se corresponden al archivo del Anexo 4: *Prueba con diferentes GC el RT.txt*

3.2.1 Pruebas

Las pruebas se han realizado ejecutando COSME, con la aplicación sintética Sumador Distribuido, durante 60 minutos con cada política de recolección de basura y utilizando la aplicación MiniBlas para obtener el tiempo máximo y mínimo de ejecución de la "función normal".

MiniBlas es una aplicación que permite consultar y modificar el valor de las variables de la aplicación que se ejecuta en COSME.

Los tiempos de ciclo se han configurado de la siguiente manera:

- `SISTEMA.tiempo_ciclo_ms = 10 ms`
- `SISTEMA.tciclo_E_CYCLE = 50 ms`

Para generar carga en la "función normal" se ha introducido un bucle que calcula *n* senos de una variable. Las iteraciones de este bucle se controlan con la variable: `sumador.num_senos_RT`.

A continuación en el siguiente gráfico se puede observar el tiempo máximo y mínimo de la "función normal" para cada política de recolección de basura.

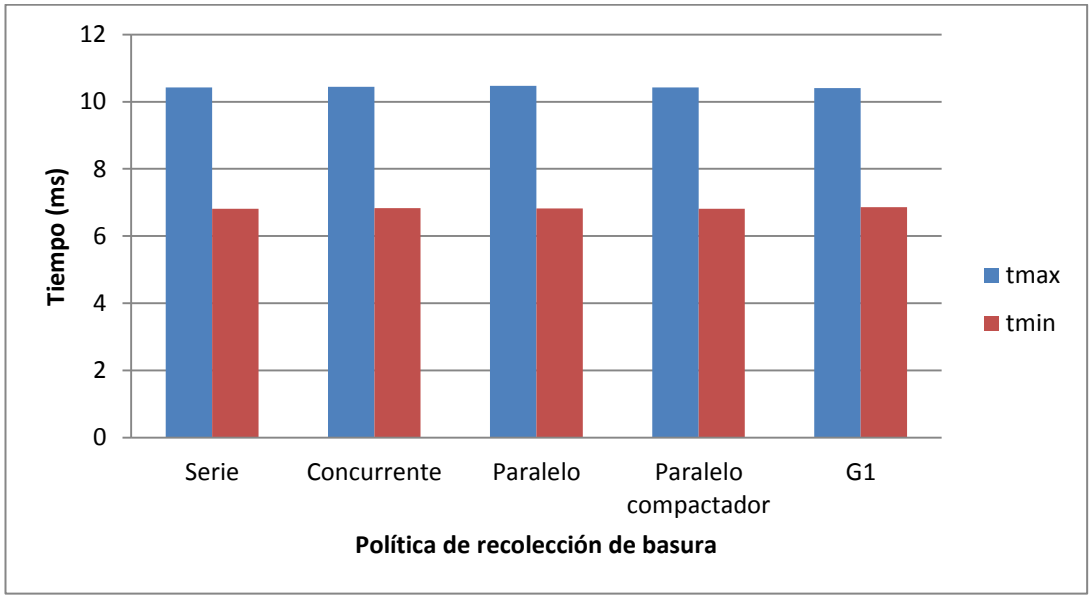

Gráfico 1: Tiempos de ejecución "función normal"

3.2.2 Análisis de los resultados

En el Gráfico 1, se puede ver que los resultados son prácticamente iguales, esto quiere decir que el tipo de recolector de basura **no afecta** al tiempo de ejecución de la "`función normal`".

3.3. Medidas de los tiempos de ejecución de la secuencia distribuida COSME

El propósito de estas pruebas es ver cómo afecta el tiempo de ejecución de la "`función normal`" al tiempo de ejecución de COSME distribuido en la plataforma de pruebas. Teniendo en cuenta que la "`función normal`" se ejecuta bajo la restricción de tiempo real. Cada prueba se ha ejecutado durante 60 minutos.

Los datos se corresponden a los archivos del Anexo 5.

3.3.1 Pruebas

Las pruebas han consistido en ejecutar COSME en modo distribuido, para medir el tiempo desde que el telegrama `exec_seq` sale de la Raspberry1 pasa por las Raspberry2, 3 y 4 hasta que llega de nuevo a la Raspberry1. Aplicando diferentes niveles de carga en la "`función normal`" de la Raspberry2 utilizando la variable `sumador.num_senos_RT`, explicada en el apartado anterior. Con un tiempo `SISTEMA.tciclo_E_CYCLE` de 500 ms.

En el siguiente gráfico se puede observar los tiempos obtenidos con los diferentes niveles de carga.

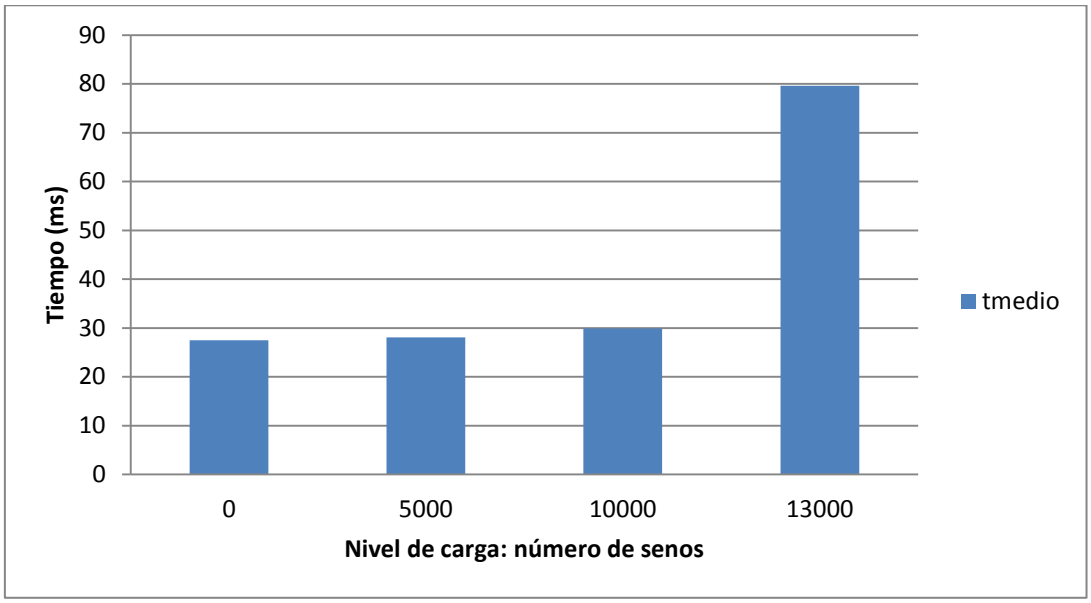

Gráfico 2: Tiempos de ejecución COSME distribuido

3.3.2 Análisis de los resultados

Se puede apreciar en el Gráfico 2, que obviamente cuanta más carga de trabajo hay en la "`función normal`" más tarda en ejecutarse un ciclo completo de la aplicación Sumador Distribuido. Porque dicha función tienen más prioridad, al ejecutarse bajo la restricción de tiempo real.

Si el tiempo de ejecución de la "`función normal`" fuera igual o superior al tiempo de ciclo de la misma, el procesador solo podría dedicar tiempo a ejecutar esta función. Ya que al ser periódica la diferencia entre el tiempo de ejecución y el tiempo de ciclo de la misma es el tiempo que el procesador puede dedicar a otros procesos.

3.4. Medidas de latencia de red

El propósito de estas pruebas es sobrecargar el *switch* y configurar de manera diferente la Calidad de Servicio para observar como varia la latencia de red entre las Raspberrys.

Cada configuración del *switch* se ha ejecutado durante 5 minutos con un tiempo de ciclo, `SISTEMA.tciclo_E_CYCLE`, de 50 ms. Con lo que se han obtenido unas 5900 muestras.

Los datos se corresponden a los archivos del Anexo 6:

3.4.1. Pruebas

Las pruebas se han realizado conectando al *switch* cuatro PCs para intercambiar dos archivos entre ellos, de 20 *GB* cada uno, el PC1 se intercambia la información con el PC2 y el PC3 con el PC4.

A su vez la Raspberry1 y la Raspberry4 ejecutan la aplicación sintética, Sumador Distribuido, para determinar la latencia entre las dos. La Raspberry1 medirá el tiempo desde que envía el telegrama `exec_seq` a la Raspberry4 hasta que la Raspberry4 le responda con el telegrama `escribir`.

La Raspberry4 medirá el tiempo que le cuesta ejecutarse desde que recibe el telegrama de la Raspberry1 hasta que le responde.

Posteriormente a los tiempos medidos por la Raspberry1 se le restarán los tiempos medidos por la Raspberry4, todo esto dividido entre 2, así se obtendrá la latencia de la red. De la cual se calculará la media, desviación típica y el valor menos desfavorable al 95% y al 100%.

El criterio de clasificación para determinar la configuración óptima del *switch*, se establecido en primer lugar por el menor valor del 95%, en segundo lugar por el menor valor de la media y en tercer lugar por el menor valor de la desviación típica.

A continuación, en la Figura 13, se describe la conexión de los puertos del *switch*.

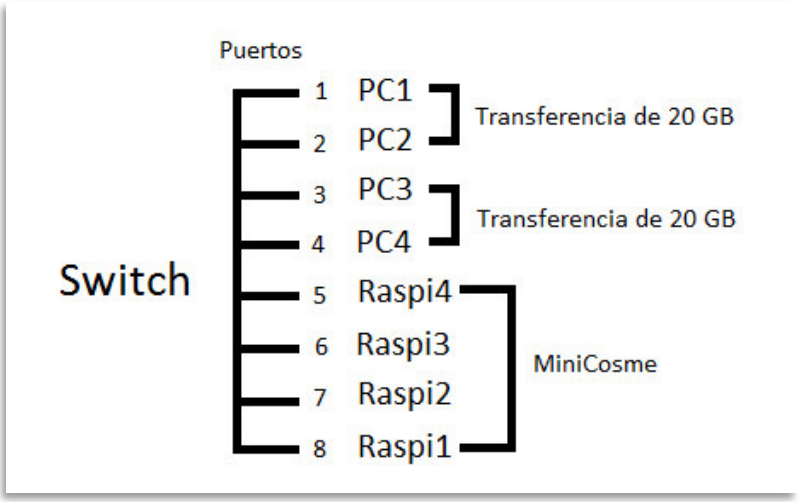

Figura 13: Esquema de conexiones del *switch*

Las siguientes pruebas se han realizado con la QoS basada en los puertos y con los modos de encolar los paquetes, Prioridad Estricta y *WRR*.

Prioridad Estricta y QoS basada en los puertos, configuraciones:

QoS se ha configurado basada en los puertos y el modo de cola con prioridad estricta.

1. **QoS desactivada y sin carga para el *switch***

Esta prueba, se ha realizado con la QoS desactivada y sin tráfico de datos en los puertos 1, 2, 3 y 4 del *switch*.

2. **QoS desactivada y con carga para el *switch***

Esta prueba, se ha realizado con la QoS desactivada y con tráfico de datos en los puertos 1, 2, 3 y 4 para sobrecargar el *switch*.

3. **QoS activada y con carga para el *switch***

Esta prueba, se ha realizado con la QoS activada dando prioridad *high*, la más alta, a los puertos conectados a las Raspberrys, y con tráfico de datos en los puertos 1, 2, 3 y 4 para sobrecargar el *switch*.

4. **QoS activada y sin carga para el *switch***

Esta prueba, se ha realizado con la QoS activada dando prioridad *high*, la más alta, a los puertos conectados a las Raspberrys, y sin tráfico de datos en los puertos 1, 2, 3 y 4.

Configuraciones *Switch*	Media	Desviación Típica	95%	100%
1-QoS desactivada y sin carga	1,11	1,61	2,47	29,18
2-QoS desactivada y con carga	0,59	1,61	3,04	20,11
3-QoS activada y con carga	0,51	1,05	2,88	30,13
4-QoS activada y sin carga	1,07	1,45	2,36	25.18

Tabla 12: Latencias de red con Prioridad Estricta (valores en ms)

En la Tabla 12, se puede observar que comparando el valor al 95% de la configuración con la QoS activada y con carga en el *switch* con la configuración con la QoS desactivada y con carga en el *switch*, se obtiene un **mejor resultado activando la QoS**, 2,88 ms frente a los 3,04 ms respectivamente. Esto supone una mejora del 5,26 %.

Comparando de nuevo el valor al 95% de la configuración con la QoS activada y sin carga en el *switch* con la configuración con la QoS desactivada y sin carga en el *switch*, se obtiene también

un **mejor resultado activando la QoS**, 2,36 ms frente a los 2,47 ms respectivamente. Esto supone una mejora del 4,45 %.

WRR y QoS basada en los puertos, configuraciones:

QoS se ha configurado basada en los puertos y el modo de cola con WRR. Asignando los pesos 1, 1, 2 y 4 a las colas *low, normal, médium y high* respetivamente. Es decir, la cola *high* estará cuatro veces más tiempo transmitiendo paquetes que la cola *low*.

1. QoS desactivada y sin carga para el *switch*

Esta prueba, se ha realizado con la QoS desactivada y sin tráfico de datos en los puertos 1, 2, 3 y 4 del *switch*.

2. QoS desactivada y con carga para el *switch*

Esta prueba, se ha realizado con la QoS desactivada y con tráfico de datos en los puertos 1, 2, 3 y 4 para sobrecargar el *switch*.

3. QoS activada y con carga para el *switch*

Esta prueba, se ha realizado con la QoS activada dando prioridad *high*, la más alta, a los puertos conectados a las Raspberrys, y con tráfico de datos en los puertos 1, 2, 3 y 4 para sobrecargar el *switch*.

4. QoS activada y sin carga para el *switch*

Esta prueba, se ha realizado con la QoS activada dando prioridad *high*, la más alta, a los puertos conectados a las Raspberrys, y sin tráfico de datos en los puertos 1, 2, 3 y 4.

Configuraciones *Switch*	Media	Desviación Típica	95%	100%
1-QoS desactivada y sin carga	1,11	1,61	2,47	29,18
2-QoS desactivada y con carga	0,58	1,48	3,09	19,93
3-QoS activada y con carga	0,5	1,37	2,5	14,86
4-QoS activada y sin carga	0,97	1,10	1,73	17,07

Tabla 13: Latencias de red con WRR (valores en ms)

En la Tabla tal, se puede observar que comparando el valor al 95% de la configuración con la QoS activada y con carga en el *switch* con la configuración con la QoS desactivada y con carga en el *switch*, se obtiene un **mejor resultado activando la QoS**, 2,5 ms frente a los 3,09 ms respectivamente. Esto supone una mejora del 19,09 %.

Comparando de nuevo el valor al 95% de la configuración con la QoS activada y sin carga en el *switch* con la configuración con la QoS desactivada y sin carga en el *switch*, se obtiene también un **mejor resultado activando la QoS**, 1,73 ms frente a los 2,47 ms respectivamente. Esto supone una mejora del 29,96 %.

3.4.2. Análisis de los resultados

El peor resultado al 95% se obtiene con la QoS desactivada y con carga, ya que el *switch* tiene que dedicar más tiempo a procesar los paquetes de los puertos con mayor tráfico. Activando la calidad de servicio se consigue un mejor resultado porque los puertos donde están conectadas las Raspberrys tienen más prioridad sobre los demás.

Comparando los dos modos de encolar los paquetes, Prioridad Estricta y WRR, se ha obtenido un mejor resultado al 95% con WRR tanto con carga como sin carga en el *switch*. Además este modo evita el problema de inanición en los puertos.

Con la QoS activada y el modo de encolar los paquetes WRR se consigue una mejora del 19,09 con carga el *switch* y del 29,96 sin carga.

En conclusión, la **QoS** se tendrá que **activar** y configurar con más prioridad los puertos donde estén conectadas las Raspberrys. El modo en el que se tratan las colas de paquetes se tendrá que configurar a **WRR**.

3.5 *Profiling* sobre la Pasarela

El objetivo de esta prueba es usar la herramienta Profiler de Netbeans 8.0.1, para detectar anomalías, como por ejemplo cuellos de botella, creación excesiva de objetos, etc. Y llevar a cabo las pertinentes optimizaciones.

Esta prueba sea ha realizado durante 60 minutos sobre la ejecución de la aplicación sintética Sumador Distribuido con un tiempo de ciclo, `SISTEMA.tciclo_E_CYCLE`, de 1000 ms.

El *profiling* se ha llevado a cabo sobre dos pates; sobre la memoria y sobre la CPU de la Pasarela.

Estos datos corresponden a los archivos del Anexo 7.

3.5.1 Memoria

Con el *profiling* de la memoria se desea obtener si el número de objetos creados es el adecuado. En la siguiente tabla se pueden observar los resultados:

Class Name - Live Allocated Objects	Total Alloc. Obj. ▼
gta.cosme.common.**ListaVariables**	25.173
gta.cosme.gateway.**Telegram**	16.209
gta.cosme.gateway.**TelegramTokenizer**	16.209
gta.cosme.common.**ItemVariable**	12.597
gta.cosme.gateway.**TelegramTypes**	35
gta.cosme.gateway.loader.**ExecSequenceName**	8
gta.cosme.gateway.**AccessLevels**	3
gta.cosme.gateway.loader.**ItemConnection**	2
gta.cosme.gateway.loader.**FontaneroXML**	2
gta.cosme.gateway.**FIFOListener**	2
gta.cosme.gateway.**TelegramTypes[]**	1
gta.cosme.gateway.loader.**AppInfo**	1
gta.cosme.gateway.loader.**AppLoader**	1
gta.cosme.gateway.loader.**ExecSequenceName[]**	1
gta.cosme.gateway.loader.**FBN_Info**	1
gta.cosme.gateway.loader.**ItemExecSequenceInfo**	1
gta.cosme.gateway.loader.**ItemInstance**	1
gta.cosme.gateway.loader.**ItemProperty**	1
gta.cosme.gateway.**ExecSeqController**	1
gta.cosme.gateway.**AccessLevels[]**	1
gta.cosme.gateway.**Gatera**	1
gta.cosme.gateway.**GateraClient**	1
gta.cosme.gateway.**PasarelaAlta**	1
gta.cosme.gateway.**PasarelaAlta$1**	1
gta.cosme.gateway.**RTPartner**	1
gta.cosme.gateway.**RemoteConnectionsLauncher**	1

Tabla 11: *Profiling* sobre la memoria utilizada por la Pasarela

En la Tabla 14 se puede ver que el número de objetos creados durante 60 minutos de ejecución.

De la clase `PasarelaAlta` y de la clase `RTPartner` se crea un solo objeto, los cuales concuerdan con el valor teórico.

Los objetos creados de la clase `FIFOListener`, 2 en total, también concuerdan. Uno para la FIFO de escritura en el *runtime* de COSME y otro para la lectura.

De la clase `GateraClient` solo se crea un objeto porque solo hay una Raspberry conectada.

De la clase `Telegram` se crean 16209 objetos. Este número de objetos **no** concuerda con el valor teórico:

- La duración de la prueba es de 60 minutos.
- Se recibe un telegrama `exec_seq` cada segundo, es decir, se reciben 3600, más el eco que genera el *runtime* de cada uno, **7200**.
- De cada telegrama `exec_seq` el *runtime* manda a la Pasarela un telegrama `publish`, en total **3600** telegramas.
- Del telegrama `ping`, se recibe uno cada dos segundos, 1800 telegramas, más el eco que genera el *runtime* de cada uno, **3600**.
- En total 7200 + 3600 + 3600 = **14400** valor teórico de objetos `Telegram` creados.

El valor teórico, 14400 objetos `Telegram`, no concuerda con el valor real medido 16209. Por lo tanto, se están creando telegramas de más, en concreto 16209 – 14400 = **1809** telegramas.

Tras concluir que el valor teórico y el real no concuerdan, se procede a examinar el código fuente para encontrar el error:

- El método `GateraClient.escucharTelegrama()` lee un *String* del Socket y crea un objeto `Telegram`, como la duración de la prueba es de 60 minutos, se crean 3600 para el telegrama `exec_seq` y 1800 para el `ping`.
- A continuación si es un telegrama para la Pasarela llama al método `PasarelaAlta.escribirTelegrama(String _txt)` donde recibe un *String* y vuelve a crear otro objeto telegrama, como la duración de la prueba es de 60 minutos se crean 1800 para el telegrama `ping`, para después invocar al método `PasarelaAlta.escribirTelegrama(Telegrama _tlg)` que escribe el telegrama en la FIFO. Por lo tanto cuando se recibe el telegrama ping se crean dos telegramas del mismo *String*.
- Al leer de la FIFO con el método `PasarelaALta.leerTelegrama(String _txt)` se crea otro objeto telegrama, como la duración de la prueba es de 60 minutos se crean 3600 objetos para el telegrama `publish`, 1800 para el eco del telegrama `ping` y 3600 para el eco del telegrama `exec_seq`.

Si se suman los telegramas creados 3600(`exec_seq`) + 1800(`ping`) + 1800(`ping`) + 3600(`publish`) + 1800(eco `ping`) + 3600(eco `esec_seq`) = 16200 objetos `Telegram`. Este valor 16200 es muy cercano al valor real medido por el *profiler* 16209.

El error se produce porque se duplican los objetos telegrama de tipo `ping`, por lo tanto el valor teórico y real no concuerda.

El error se corrige llamando al método `PasarelaAlta.escribirTelegrama (Telegrama _tlg)` en vez de al método `PasarelaAlta.escribirTelegrama (String _txt)` desde `GateraClient.escucharTelegrama()` cuando llega un telegrama para la Pasarela.

Optimizaciones

Dado que en las clases `Telegram` y `TelegramTokenizer` se emplean numerosos objetos de tipo `String`. Una optimización que se puede llevar a cabo es sustituir los objetos `String` por `StringBuffer`. Porque a la hora concatenar dos `String` con el operador suma, se crea un nuevo `String` resultante, con lo que constantemente se está creando objetos nuevos. El objeto `StringBuffer` permite concatenar dos cadenas de texto sin tener que crear otro objeto, por lo tanto el uso de `StringBuffer` es más eficiente.

3.5.2 CPU

Con el *profiling* de la CPU se desea averiguar si existe algún cuello de botella en el código. Detectar cuáles son los métodos que más veces se ejecutan para poder optimizarlos. En la siguiente tabla se muestran los resultados obtenidos:

Hot Spots - Method	Self Time (CPU) ▾
gta.cosme.gateway.Telegram.**<init>** (String)	46,7 ms
gta.cosme.gateway.RTPartner.**run** ()	46,0 ms
gta.cosme.common.ListaVariables.**<init>** ()	26,6 ms
gta.cosme.gateway.Gatera.**aceptarConexiones** ()	25,6 ms
gta.cosme.gateway.Gatera.**arrancarProyecto** ()	24,8 ms
gta.cosme.gateway.PasarelaAlta.**escribirTelegrama** (gta.cosme.gateway.Telegram)	23,7 ms
gta.cosme.gateway.loader.AppLoader.**initApplication** (String, String)	23,2 ms
gta.cosme.gateway.PasarelaAlta.**leerTelegrama** (String)	22,8 ms
gta.cosme.gateway.TelegramCreator.**getTlg__execSequence** (gta.cosme.gateway.loader.E	20,5 ms
gta.cosme.gateway.TelegramTypes.**<clinit>**	16,5 ms
gta.cosme.gateway.loader.ItemExecSequenceInfo.**<init>** ()	14,8 ms
gta.cosme.gateway.RTPartner.**<init>** (gta.cosme.gateway.Gatera, String)	14,0 ms
gta.cosme.gateway.Gatera.**<init>** ()	12,9 ms
gta.cosme.gateway.loader.AppLoader.**loadRuntimePartners** (String)	12,4 ms
gta.cosme.gateway.loader.ItemInstance.**setOrden** (int)	10,2 ms
gta.cosme.gateway.TelegramTokenizer.**nextToken** ()	10,1 ms
gta.cosme.gateway.RemoteConnectionsLauncher.**run** ()	8,52 ms
gta.cosme.common.ItemVariable.**getNombre** ()	8,27 ms
gta.cosme.gateway.RTPartner.**publishOutputs** (gta.cosme.gateway.loader.ExecSequenceNa	7,95 ms
gta.cosme.common.ListaVariables.**getLista** ()	7,52 ms
gta.cosme.gateway.loader.AppLoader.**initApplications** ()	7,29 ms
gta.cosme.gateway.GateraClient.**<init>** (gta.cosme.gateway.Gatera, java.io.BufferedReader,	6,93 ms
gta.cosme.common.ItemVariable.**getValor** ()	5,46 ms
gta.cosme.gateway.loader.ExecSequenceName.**<clinit>**	4,41 ms
gta.cosme.gateway.TelegramTokenizer.**<init>** (String)	3,75 ms
gta.cosme.gateway.FIFOListener.**run** ()	2,37 ms
gta.cosme.gateway.PasarelaAlta.**escribirTelegrama** (String)	0,000 ms
gta.common.xml.XMLCommonParser.**buildTree** (org.w3c.dom.Node)	0,000 ms

Tabla 12: *Profiling* sobre la CPU utlizada por la Pasarela

Tabla 15:

- *Self Time (CPU)*: tiempo esperando en todas las invocaciones del método, excluyendo las llamadas a métodos internos.

Optimizaciones

Se puede observar en la Tabla 15, que los métodos que más tiempo han estado ejecutándose son el constructor de la clase `Telegram` y el método `run` de la clase `RTPartner`. Por lo tanto, estos dos métodos son los candidatos a optimizar.

`Telegram.<init>(String)`

El constructor de la clase `Telegram` se encarga de obtener la información del *String* pasado como parámetro y es el objeto más utilizado en la Pasarela. Por lo tanto, su optimización beneficiaría al rendimiento.

Las posibles optimizaciones son las siguientes:

- Sustituir las sentencias *if/else* anidadas por un *switch*:

- Código sin optimizar:

```
if (comando.equals(cesta)){
}else if (comando.equals(publish)){
}else if (comando.equals(exec_seq)){
...
```

- Código optimizado:

```
switch (comando) {
      case cesta:
      case publish:
      case exec_seq:
```

- Enviar el objeto entero `Telegram` para no tener que parsear *Strings*. Pero esto no es recomendable ya que los *Strings* ofrecen más posibilidades de conexión entre diferentes paradigmas de programación.

`RTPartner.run()`

Este método solo se ejecuta una vez para conectar con otra Raspberry, por lo que no es necesario optimizarlo.

3.5.3 Resultados de las optimizaciones

Una vez aplicada la optimización en la concatenación de cadenas de texto con la clase `StringBuffer`, corregida la duplicación del telegrama `ping` y aplicada la optimización a los *if/else* anidados, los tiempos obtenidos son los siguientes:

Optimizaciones	Media	Desviación Típica	95%	100%
Sin optimizaciones	15,11	23,84	49,04	659,96
Con optimizaciones	15,13	23,86	49,03	650,82

Tabla 16: Tiempos de ejecución Pasarela (valores en ms)

Como se puede observar en la Tabla 16, la optimización en la concatenación de cadenas de texto con la clase `StringBuffer`, la corrección en la duplicación del telegrama `ping` y la optimización del constructor del método `Telegram`, no ha supuesto ninguna mejora en el funcionamiento de la Pasarela. Esto puede ser debido a que el tiempo de parseo de un telegrama es muy pequeño.

4. Corrección de errores en la Pasarela

Se ha encontrado un error en el código de la Pasarela, concretamente en la clase `AppLoader` en el método `getNombreFromVariable(String_variableName)`.

Este error se ha producido al eliminar los RtPartners anteriores a cada Raspberry, es decir, en un primer momento cada Raspberry conocía a todas las Raspberrys de la red distribuida, pero se decidió que cada Raspberry solo conociera la existencia de la Raspberry a la que enviaba algún dato. Por ejemplo, en la plataforma de pruebas destina a este proyecto, la Raspberry 1 solo tiene la dirección IP de la Raspberry 2, la Raspberry 2 de la Raspberry 3…

Este método se encarga de devolver el nombre del *runtime* a partir de una variable, pero devuelve el nombre del *runtime* local si el nombre del *runtime* de la variable pasada como parámetro no está en la *hash* de *RTPartners* (marcado en rojo en el código). Así mismo, como en dicha hash no están todos los *RTPartners* se produce el error.

Código con error:

```
// A partir del nombre de una variable, devuelve el nombre del runtime
private String getNombreRTFromVariable (String _variableName){
	String runtimeName = ;
	int primerPunto = _variableName.indexOf('.');

	if (primerPunto > 0){
		runtimeName=_variableName.substring(0, primerPunto);
	}

	// si ese nombre de runtime no está registrado en la hash, es
porque es una variable local. En ese caso el nombre que debe
devolverse es el del runtime local
	if (this.gatera.getRTPartners().get(runtimeName) == null){
		runtimeName = this.gatera.getAliasLocalRT();
	}

	return runtimeName;
}
```

Para corregir el error que se produce en la parte del código anterior marcado en rojo, basta con eliminar el condicional, por lo tanto el código quedaría de la siguiente manera:

Código sin error:

```
// A partir del nombre de una variable, devuelve el nombre del runtime
private String getNombreRTFromVariable (String _variableName){
	String runtimeName = ;
	int primerPunto = _variableName.indexOf('.');

	if (primerPunto > 0){
		runtimeName=_variableName.substring(0, primerPunto);
	}

	return runtimeName;
}
```

5. Conclusiones

Tras haber analizado los resultado obtenidos con la aplicación sintética Sumador Distribuido en la plataforma de pruebas. En la **Pasarela** se ha obtenido una mejora en el tiempo de ejecución del **36,91%** aplicando los parámetros **`UseConcMarkSweepGC`**, **`CompileThreshold=1000`** a la JVM en la ejecución de la Pasarela.

Se ha detectado que los diferentes tipos de recolectores de basura de Java no afectan a la ejecución de la "`función normal`", función que se ejecuta bajo restricción de tiempo real. Pero el tiempo real sí que afecta al tiempo de ejecución del ciclo distribuido, cuanto más tiempo dura la ejecución de la "`función normal`" más tiempo tarda en ejecutarse un ciclo distribuido.

En lo que se refiere a las latencias de red, activando la **QoS** en el *switch*, dando más prioridad a los puertos donde se encuentran conectadas las Raspberrys y utlizando el modo de encolar los paquetes **WRR**, se ha obtenido una mejora del **19,09 %** cuando hay carga en el *switch* y del **29,96 %** cuando no hay carga en el *switch*.

Con el ***profiling*** sobre la Pasarela se ha detectado una creación de objetos anómala con el telegrama `ping`. También se ha intentado optimizar el método más usado, el constructor de la clase *`Telegram`* y la concatenación de cadenas de caracteres cambiando los objetos de tipo `String` por `StringBuffer`. Pero esto no ha supuesto ninguna mejora en el tiempo de ejecución de la Pasarela debido a que el tiempo de parseo de un telegrama es muy pequeño.

Trabajo futuro

- Realizar las pruebas con una aplicación sintética que creara más carga de trabajo en COSME.
- Realizar las pruebas con diferente topología.
- Sustituir las Raspberrys por servidores con mayor capacidad de cómputo.
- Sustituir la comunicación mediante cadenas de texto por el estándar OPC-UA.

6. Agradecimientos

Quiero aprovechar estas líneas para agradecer a todas las personas que me han ayudado y me han apoyado a lo largo de estos meses en mi proyecto fin de grado.

En primer lugar a mis directores de proyecto Carlos Catalán Cantero y Félix Serna Fortea a quienes me gustaría expresar mi agradecimiento por su tiempo y dedicación para que este proyecto saliese adelante.

En segundo lugar a mi familia y a mi novia por la confianza que han depositado en mí y su gran apoyo día a día, que siempre han sabido aconsejarme en los perores momentos.

Y por último a mis compañeros de clase, de los cuales me llevo unos grandes amigos, que han hecho que estos años se hicieran mucho más amenos.

7. Bibliografía

1. Raspberry Pi:

   http://es.wikipedia.org/wiki/Raspberry_Pi

2. Configuración IP en Raspberry Pi:

   http://www.electroensaimada.com/ip-estaacutetica.html

3. Parámetros de la JVM HotSpot:

   http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html

4. JVM HotSpot:

   http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html

5. Memoria en la JVM HotSpot:

   http://es.slideshare.net/luisdebello/administracin-de-memoria-en-java

   http://www.slideshare.net/leonjchen/java-gc-javadeveloperdaytw

6. Memoria y recolectores de basura Java 7, tutorial de Oracle:

   http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

7. Java Ergonomics:

   http://www.oracle.com/technetwork/java/ergo5-140223.html

8. Recolector de basura G1, video explicativo de Oracle:

   https://www.youtube.com/watch?v=bhVzCIk3-Q4

9. Recolector de basura G1:

   http://www.infoq.com/articles/G1-One-Garbage-Collector-To-Rule-Them-All

10. Explicación de las trazas del recolector de basura:

    http://www.herongyang.com/JVM/Memory-PrintGCDetails-Garbage-Collection-Logging.html

11. Manual *switch* Cisco SLM2008:

    http://www.cisco.com/c/dam/en/us/td/docs/switches/lan/csbss/slm2005_slm2008/administration/guide/SLM2008AG.pdf

Anexos

Anexo 1: Clase Java para medir tiempos

La siguiente clase se ha implementado para poder medir tiempos en nanosegundos y volcar los datos en un fichero de texto.

```
package gta.util;

public class LatencyMeter {

    private String id;
    private long tInicio;
    private long tmax = 0;
    private long tmin = 2147000000;
    private long sumatorio = 0;
    private long numCiclos = 0;

    public LatencyMeter(String id) {
        this.id = id;
    }


    public void setStart(){
        tInicio = System.nanoTime();
    }


    public void setStop(){
        long t = System.nanoTime() - tInicio;
        sumatorio = sumatorio + t;
        numCiclos++;

        if (t > tmax) {
            tmax = t;
        } else if (t < tmin) {
            tmin = t;
        }
    }

    public String toString() {
       return id + \n + Tiempo min:  + tmin +  ns\n +
                 Tiempo medio:  + (sumatorio/numCiclos) +  ns\n +
                 Timepo max:  + tmax +  ns\n +
                 Num de muestras:  + numCiclos + \n +
                 Tiempo total:  + sumatorio +  ns\n;

    }
}
```

Anexo 2: *Shell script* para ejecutar COSME periódicamente

El siguiente *shell script* se ha implementado para ejecutar de manera automática diferentes configuraciones de la JVM, especificadas en un fichero de texto, durante un periodo de tiempo determinado.

```
#!/bin/bash
#Arranca COSME periódicamente con la configuración para la pasarela
determinada en el archivo configPasarelaJVM.txt

tiempoEnEjecucion=1800 #segundos

while read line
do

	#se introduce la configuración de la pasarela en el fichero
latencias.txt para saber cuál se ha usado
	echo $line >> ../runtime/latencias.txt

	#se elimina la última línea de launchPasarela.sh
	sed -i '$d' ../runtime/launchPasarela.sh

	#se añade la nueva línea con la configuración para ejecutar la
Psarela
	echo $line >> ../runtime/launchPasarela.sh

	cosme start -v
	sleep $tiempoEnEjecucion
	cosme stop -v
	sleep 10 #esperamos a que COSME termine
	cat ../runtime/gc.txt >> ../runtime/latencias.txt
done < configPasarelaJVMOptimize.txt
```

Anexo 3: Medidas de los tiempos de ejecución de COSME

Archivos con las medidas de los tiempos de ejecución de los distintos tipos de recolectores de basura y el umbral de compilación, de los distintos tipos de parámetros de rendimiento y de la variación del *heap*. Adjuntado en el CD.

- Directorio: *Anexo 3*.

Anexo 4: Medidas de los tiempos de la "función normal"

Archivo con los tiempos de ejecución de la "`función normal`", adjunto en el CD.

- Directorio: *Anexo 4*.

Anexo 5: Medidas del tiempo de ejecución de COSME Distribuido

Archivo con los tiempos de ejecución de COSME Distribuido con diferentes cargas de trabajo en la "`función normal`", adjunto en el CD.

- Directorio: *Anexo 5*.

Anexo 6: Medidas de latencia de red en COSME Distribuido.

Archivo con los tiempos de latencia de red, adjunto en el CD.

-Directorio: *Anexo 6*.

Anexo 7: Medidas del *profiling* sobre COSME.

Archivo con los resultados del *profiling* sobre COSME, adjunto en el CD.

-Directorio: *Anexo 7*.