Control de Recursos en un nodo de red programable con entorno de ejecución basado en Java

Andrés Sevilla[1] María Calderón[2] Manuel Urueña[2] David Larrabeiti[2]

[1]Escuela Universitaria de Informática
Universidad Politécnica de Madrid
E-mail: asevilla@eui.upm.es

[2]Dep. de Ingeniería Telemática
Universidad Carlos III de Madrid
E-mail: maria@it.uc3m.es, dlarra@it.uc3m.es, muruenya@it.uc3m.es

Abstract. *A difficult problem to solve in systems based in Java [1], such as execution environments of programmable network nodes, is how to control the resource consumption of applications. Because of this, buggy or malicious applications can potentially crash these network nodes. Java is a good choice to program the network, because it offers many features to ease development of network applications, such as portability, safety, multithreading, etc. This paper proposes and evaluates an interface to control the resource consumption of applications. The implementation uses only standard mechanisms of the Java Virtual Machine in order to get a portable and independent solution.*

1 Introducción

Cada día es mayor el número de entornos que ofrecen la movilidad de código o la extensibilidad como una de sus principales características. Estos entornos, de los que el paradigma de las redes activas [2] y, en menor medida, las redes programables [3] son claros exponentes, se ven expuestos a programas con comportamientos anómalos que pueden llegar a bloquear uno o varios nodos al consumir todos los recursos disponibles. Por este motivo la seguridad (*safety*) en entornos de ejecución abiertos ha sido un campo de intensa investigación en los últimos años [4].

La tecnología Java ha sido la elegida para implementar un gran número de estos entornos por ofrecer además de la portabilidad de código un modelo de ejecución donde los programas con comportamientos erróneos o mal intencionados se puedan ejecutar sin causar daños a terceros o al propio entorno. Un ejemplo de esto último son los conocidos *applets* de Java que ejecuta el propio navegador web impidiendo determinados accesos a los recursos del sistema como, por ejemplo, el disco duro.

A pesar de que Java ofrece un entorno de ejecución relativamente seguro, no dispone de ningún mecanismo que permita controlar el consumo de recursos de los programas que en él se ejecutan. A modo de ejemplo, baste un programa con un simple bucle infinito para consumir una elevada cantidad de tiempo de CPU sin que se le aplique ningún límite de gasto. Java, frente a este tipo de situaciones, todavía no dispone de ningún mecanismo que permita controlar el consumo de recursos como tiempo de CPU, memoria o ancho de banda. Esta carencia ha provocado en los últimos años el diseño de diversas soluciones para intentar paliarla. Dos han sido principalmente las estrategias seguidas: una, incluir el control en las propias aplicaciones [5][6] y otra, incorporar los mecanismos de control en la máquina virtual Java [7] [8].

La primera de las soluciones tiene por objetivo evitar la modificación de la máquina virtual Java, incorporando el control de los recursos en las propias aplicaciones mediante la inserción de diferentes controles en su código (*bytecode*). Reseñar que para que esta solución sea efectiva, los controles han de insertarse en todas las clases que forman una aplicación.

El segundo tipo de soluciones propuestas se basan en la modificación de la máquina virtual Java o en la utilización de los mecanismos de control de recursos del sistema operativo [6] en el que se ejecuta la máquina virtual. Este tipo de soluciones adolecen del problema de la portabilidad. En el primer caso, un sistema desarrollado en Java únicamente puede ejecutar en una determinada máquina virtual, y en el segundo caso, el código sólo funciona con una versión concreta del sistema operativo.

El trabajo aquí descrito se circunscribe en el campo de las redes programables, y más concretamente, en el sistema SARA [9], cuyo entorno de ejecución emplea una máquina virtual Java. Por lo tanto, se hace imprescindible disponer de mecanismos de control de recursos que protejan a los nodos de la red frente a aplicaciones activas o programas de red que por comportamientos erróneos o malintencionados

puedan dejar a un nodo sin recursos, y consecuentemente no operativo.

Los mecanismos propuestos en este artículo van a permitir controlar el gasto de memoria, tiempo de CPU y número de *threads* de las aplicaciones que se ejecutan en el nodo de red. La implementación de estos mecanismos mostrará que es posible controlar el gasto de recursos sin necesidad de modificar la máquina virtual Java o el sistema operativo anfitrión. Asimismo, la solución propuesta servirá de modelo para la incorporación progresiva de nuevos controles a medida que las necesidades del nodo lo vayan requiriendo.

2 El nodo programable SARA

Un nodo de red programable es un *router* con capacidad para actualizar dinámicamente tanto los protocolos soportados, como el código que realiza el tratamiento de los paquetes que lo atraviesan. Este procesamiento no está restringido exclusivamente a la capa 3, y entre las aplicaciones objetivo de este tipo de nodos, se hallan filtros inteligentes (sobre contenidos), detectores de ataques, traductores de protocolos, transformadores de codec, clasificadores/planificadores de paquetes, gestores de colas, etc.

Un nodo de red SARA es un nodo de red programable cuya peculiaridad es que se basa en la llamada arquitectura "*router*-asistente", formada por dos elementos con funcionalidades bien diferenciadas: el *router* IP, que encamina paquetes, identifica aquellos susceptibles de un procesamiento especial y los desvía al asistente para que los procese una aplicación específica, y el asistente, que no es más que un ordenador de propósito general (o conjunto de ordenadores, dependiendo de las necesidades de procesamiento) conectado al *router* mediante una red local de alta velocidad. La misión del asistente es proporcionar un entorno de ejecución especialmente diseñado para lanzar y ejecutar aplicaciones de red con rapidez y seguridad, desacoplando las funciones de reenvío de las de procesamiento costoso específico de aplicación. Un esquema relacionado es el propuesto en el grupo de trabajo IETF ForCES [10] para la construcción de arquitecturas de *routers* abiertas.

La arquitectura del asistente está formada por las capas siguientes (Figura 1): sistema operativo Linux, que realiza las funciones de *NodeOS* [11][12]; máquina virtual Java; código nativo que da acceso a servicios del sistema operativo (p.e. *raw sockets*); API de SARA; y las aplicaciones de red.

La incorporación del control de recursos en el nodo de red puede realizarse en tres puntos distintos, de acuerdo con la arquitectura del mismo: en el *NodeOS*, en el entorno de ejecución y en las aplicaciones de red. La incorporación en el *NodeOS*, que en el caso de SARA es Linux, pasa por hacer visibles al entorno de ejecución los controles ya disponibles en el sistema operativo. La segunda alternativa es hacerlo en el entorno de ejecución y más concretamente en la máquina virtual Java, que es precisamente el lugar donde más y mejor información está disponible sobre el consumo de recursos que realizan las aplicaciones.

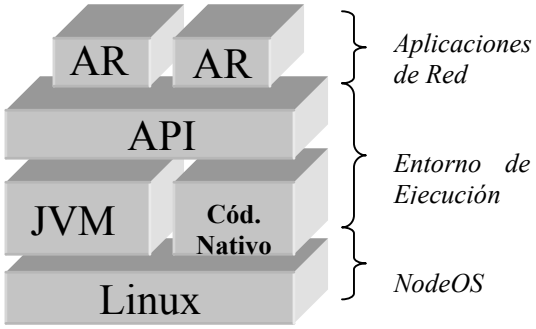

Figura 1

La tercera alternativa es incorporar el control de recursos en las aplicaciones activas, mediante la técnica de la reescritura de *bytecodes*. Esta no crea ningún tipo de dependencia con la máquina virtual o sistema operativo, y es compatible con cualquier otra forma de realizar el control, ya que esta solución puede inhibirse simplemente con no reescribir los ficheros *class* que forman una aplicación.

La solución al problema del control de recursos, una vez descritos los diferentes lugares para situarlo, deberá cumplir los objetivos siguientes:

- Minimizar el impacto de la incorporación de los mecanismos de control en la arquitectura software del nodo de red.
- Portabilidad. Evitar que la solución sea dependiente de una determinada implementación de la máquina virtual o de una versión concreta de un sistema operativo, por lo que se desecha la incorporación de los mecanismos en el *NodeOS* o en la máquina virtual. Para garantizar la portabilidad, la solución deberá únicamente utilizar interfaces estándar de Java.
- Extensibilidad. En una primera versión sólo se controlarán los consumos por aplicación de red de CPU, memoria y número de *threads*. El diseño de la solución deberá ser lo suficientemente flexible para permitir la incorporación del control de nuevos recursos como, por ejemplo, la limitación de uso de paquetes o clases del SDK de Java.
- Minimizar el impacto en coste de CPU, en el que siempre incurre cualquier tipo de control.
- Contabilización del gasto de recursos por aplicación de red.

Los mecanismos aquí propuestos servirán para desarrollar nuevas funcionalidades y potenciar otras

en el nodo SARA como, por ejemplo, la terminación de aplicaciones de red [13] que sobrepasen un determinado umbral de consumo de recursos o para facturar a los usuarios en función del consumo de recursos que sus aplicaciones realizan. Otro aspecto, no menos importante, es conocer el comportamiento del consumo de recursos de las aplicaciones que se ejecutan en SARA, con el objetivo de disponer de información real que facilite la configuración, tanto hardware como software, del nodo.

3 La solución propuesta

La solución propuesta está formada por dos módulos (ver Figura 2): una biblioteca de enlace dinámico (`ctrl_rec`) y un interfaz de acceso a la biblioteca desde programas Java (clase `Recursos`).

Desde el punto de vista de la arquitectura de SARA, la solución propuesta encaja en el entorno de ejecución, pero sin necesitar la modificación de la máquina virtual Java. Para ello se han utilizado únicamente las extensiones de la máquina virtual Java: JVMPI (*Java Virtual Machine Profile Interface*) [14][15] y JNI (*Java Native Interface*) [16]. JVMPI permite acceder, desde una biblioteca de enlace dinámico, a diversas informaciones internas de la máquina virtual Java. Esta interfaz suele utilizarse generalmente para labores de depuración o monitorización de las aplicaciones Java y, a través de ella, la máquina virtual hace visible sus informaciones internas mediante eventos, que son debidamente capturados y manejados por las funciones definidas en la biblioteca de enlace dinámico. Por su parte, JNI facilita el acceso, desde programas Java, a código nativo del sistema, también a través de bibliotecas de enlace dinámico.

Los consumos de recursos se almacenan en la biblioteca `ctrl_rec` y únicamente están accesibles, en modo consulta, al entorno de ejecución a través de la clase Java `Recursos`, con lo que se logra un nivel adicional de abstracción.

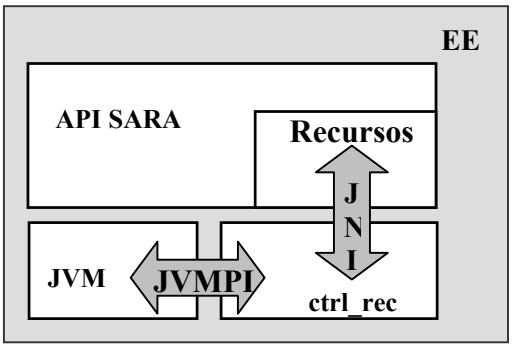

Figura 2

3.1 Interfaz Java

La interfaz descrita en la clase `Recursos` proporciona a la API de SARA todos los consumos de las aplicaciones que se ejecutan en el nodo – registrados y elaborados en la biblioteca `ctrl_rec`. Esta interfaz será utilizada posteriormente para implementar las políticas de gestión de recursos del entorno de ejecución del nodo.

Las aplicaciones de red se identifican dentro de la red programable por un número entero y están formadas por al menos un *thread* Java. Para poder utilizar los métodos definidos en la interfaz es necesario proporcionar el identificador de la aplicación, y cuando se accede al consumo realizado por la aplicación se está recuperando el consumo global realizado por todos los *threads* que pertenecen a la aplicación.

Todos los métodos definidos en la clase `Recursos` son métodos de clase (estáticos), con lo que se evita tener que crear un objeto para poder hacer uso de ellos. La definición de la clase es la siguiente:

```
static double getCpuTime (int AAid)
static long   getMem (int AAid)
static int    getNumThreads (int AAid)
```

El método `getCpuTime` devuelve el gasto de CPU en milisegundos, `getMem` el consumo de memoria en octetos, y `getNumThreads` el número de *threads* de los que consta la aplicación. Este es el interfaz básico, al que habrá que añadir los métodos que permitan acceder, en el futuro, al consumo de otros recursos.

3.2 Implementación

La interfaz JVMPI

El evento es la forma utilizada por la máquina virtual Java para exportar su información interna a la biblioteca de enlace dinámico proporcionada en el arranque de la máquina virtual (opción `-Xrun`). Los eventos son generados por la máquina virtual tras una petición de recepción realizada por la biblioteca `ctrl_rec` a través de la función `EnableEvent`. Los eventos y sus informaciones asociadas están predefinidas en la definición de JVMPI, y clasificados de acuerdo a las abstracciones siguientes:

- *Threads*. Eventos de arranque y terminación de *threads*.
- *Objetos*. Eventos de asignación y liberación de memoria.
- *Misceláneas*. Eventos de carga y descarga de clases, entradas y salidas de los *threads* de regiones críticas, comienzo de la ejecución del recolector de basura y, arranque y parada de la máquina virtual.

Además de la definición de estos eventos, JVMPI proporciona una serie de funciones para crear *threads* y regiones críticas en el interior de la máquina virtual

o gestionar la actividad de los *threads* o del recolector de basura, entre otras.

Inicio de la ejecución de las aplicaciones de red

De cara al control del gasto de recursos no existe una función específica en la API de SARA que informe a la biblioteca `ctrl_rec` de la creación de una nueva aplicación. Se considerará que una nueva aplicación ha sido creada cuando la máquina virtual arranque el primer *thread* de la aplicación, para lo cual será necesario que en la creación del *thread* se le asocie a éste una aplicación. La forma elegida para implementar esta asociación es agrupar todos los *threads* de una misma aplicación en un grupo (clase `java.lang.ThreadGroup`), codificando en el nombre del grupo el identificador de la aplicación. Cuando el *thread* es creado, la máquina virtual envía el evento `THREAD_START` junto con su identificador (`env_id`), nombre y grupo a la biblioteca `ctrl_rec`. En el tratamiento del evento se extrae el identificador de la aplicación del nombre del grupo y, en función de éste, se inicializan las estructuras de datos encargadas de llevar la cuenta del consumo de recursos de dicha aplicación.

Control del gasto de memoria

Cada vez que se crea un nuevo objeto, mediante la instrucción `new`, la máquina virtual Java envía el evento `NEW_OBJECT` a la biblioteca `ctrl_rec`. El evento comunica la cantidad de memoria reservada y el identificador del *thread* que creó el objeto. La biblioteca, mediante el identificador del *thread*, obtiene el identificador de la aplicación, pasando a continuación a registrar el consumo de memoria realizado. Para indicar que la memoria ocupada por el objeto ha sido liberada, la máquina virtual envía el evento `DELETE_OBJECT`, procediéndose en el tratamiento del mismo a descontar el espacio de memoria liberado del contador de consumo de la aplicación.

Existen otros eventos JVMPI relacionados con la gestión de la memoria como `MOVE_OBJECT`, `NEW_ARENA` y `DELETE_ARENA`, pero en la actual versión (1.4) de JVMPI no se utilizan.

Control del gasto de CPU

JVMPI no dispone de una función que permita conocer cuánto tiempo de CPU ha consumido un *thread* Java. La única opción disponible es realizar una estimación estadística del consumo de CPU utilizando las funciones `ThreadSuspend`, `ThreadResume` y `ThreadHasRun`. Esta última devuelve un valor lógico indicando si el *thread* ha consumido tiempo de CPU entre dos llamadas consecutivas a `ThreadSuspend`, obviamente, con una llamada a `ThreadResume` entre ambas. Para llevar a cabo la estimación del gasto de CPU, se crea en el arranque de la máquina virtual, un *thread* que hemos llamado monitor cuyo comportamiento es el siguiente:

```
while (TRUE) do
  sleep (T_MUESTREO);

  for t in Threads do
    ThreadSuspend (t);
  end for

  Thr_Activos = 0;

  for t in Threads do
    If ThreadHasRun (t) then
       Thr_Activos++;
    end if
  end for

  for t in Threads do
    Gasto (t).cpu +=
         T_MUESTREO/Thr_Activos;
    ThreadResume (t);
  end for

end while
```

En cada intervalo de muestreo (`T_MUESTREO`) se calcula el número de *threads* activos (`Thr_Activos`) que han consumido CPU y entre ellos se reparte, de forma proporcional, el gasto de CPU. Como puede observarse, este método para calcular el gasto de CPU es aproximado, ya que un *thread* puede que en un intervalo haya consumido una fracción muy pequeña del mismo y se le esté asignando un consumo que es proporcional al número de *threads* activos en el intervalo. Por ello, la elección del intervalo de muestreo es un parámetro crítico, no debiendo ser ni muy grande ni muy pequeño. Un intervalo grande provoca unas estimaciones malas, mientras que un intervalo muy pequeño proporciona una mejor estimación del gasto de CPU pero hace que la ejecución del *thread* monitor sea muy frecuente, con la consiguiente penalización en los tiempos de ejecución de las aplicaciones.

Este método, aun careciendo de exactitud en los cálculos, sirve para detectar situaciones de consumo excesivo de tiempo de CPU. Una posible mejora que estamos evaluando, consiste en utilizar alguna función más de JVMPI como, por ejemplo `GetCallTrace`, para realizar una estimación más exacta del gasto, basándose en la información de ejecución de los métodos Java. Aunque siempre implicará un mayor coste de cálculo.

4 Evaluación

La evaluación de la biblioteca de control de recursos se va a realizar desde dos puntos de vista: exactitud

en la contabilización del consumo de recursos y penalización introducida por los mecanismos de control de recursos añadidos.

4.1 Contabilidad del consumo de recursos

Antes de presentar los resultados obtenidos, vamos a describir brevemente qué técnicas y herramientas se han utilizado para obtener los resultados.

Para contrastar la validez del consumo de memoria devuelto por la biblioteca `ctrl_rec` se ha implementado el método `sizeof`. Este método, utilizando la reflexión de Java (clase `java.lang.reflect`), permite calcular el tamaño de un objeto, pasado como parámetro, a partir de los tipos básicos del lenguaje Java (`char`, `int`, `long`, etc) que tienen un tamaño conocido.

Para contrastar la validez del consumo de CPU devuelto por la biblioteca `ctrl_rec` se van a utilizar las herramientas de gestión de procesos del sistema operativo en el que se ejecuta la máquina virtual. El sistema operativo Linux dispone de varias herramientas que permiten conocer el consumo de CPU que realizan los procesos, siendo de especial utilidad el comando `time` que, entre otras informaciones, devuelve los tiempos de gasto de CPU de un proceso en los modos usuario y sistema. Estos dos tiempos van a permitir comprobar cómo de buena es la estimación realizada por la biblioteca `ctrl_rec` para el gasto de CPU.

Control del gasto de memoria

Para evaluar la exactitud del cálculo del gasto de memoria hemos elaborado una batería de pruebas consistente en un conjunto de programas Java con diferentes gastos de memoria. El entorno de pruebas permite obtener el consumo de memoria de una aplicación de dos formas, una utilizando el método `Recursos.getMem` de la biblioteca `ctrl_rec` aquí propuesta y otra, mediante el método `sizeof` antes comentado. En la Figura 3 se comparan los resultados obtenidos para dichas aplicaciones Java. Todas estas aplicaciones crean un vector de enteros pero en cada caso con un diferente número de entradas.

Se observa que el método `Recursos.getMem` devuelve un gasto de memoria ligeramente superior (en concreto 16 octetos más) que el tamaño calculado por `sizeof`, siempre y cuando el número de elementos del vector sea inferior a los 204.800. Esta diferencia puede deberse a la memoria adicional que necesita la máquina virtual Java para gestionar un objeto de tipo vector. Para vectores con más de 204.800 entradas la diferencia aumenta hasta los 8.168 octetos sin superar el 0,4%. A la vista de estos resultados, y teniendo en cuenta que el gasto de memoria de las aplicaciones se calcula a partir de los datos que suministra la máquina virtual a través de la interfaz JVMPI, se puede afirmar que el método `Recursos.getMem` calcula de una forma ajustada el consumo de memoria.

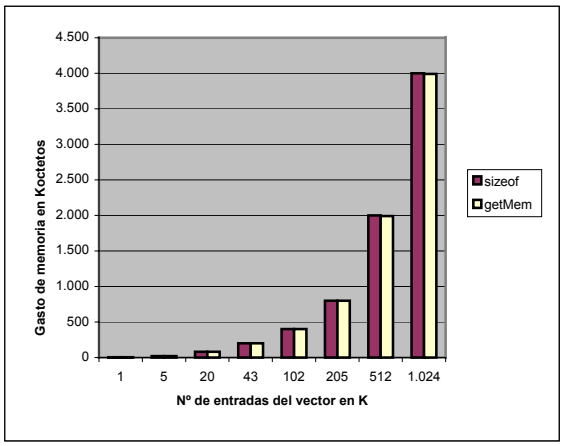

Figura 3

Control del gasto de CPU

Al igual que para el caso de la memoria, hemos elaborado una serie de pruebas para contrastar la exactitud de la estimación del gasto de CPU. Las pruebas se han clasificado de acuerdo a diferentes criterios que van desde el tipo de la prueba hasta las condiciones de realización (arquitecturas mono y multiprocesador, carga del sistema, parámetros físicos de configuración, etc). En la Figura 4 se muestra una comparativa de los tiempos devueltos por la biblioteca `ctrl_rec` y por el comando `time` (gasto en modo usuario) de Linux, para una prueba con una única aplicación de red compuesta de dos partes: una con gasto intensivo de CPU y otra con una espera temporizada. La prueba admite diferentes duraciones (iteraciones) que se muestran en el eje de ordenadas de la figura. Los resultados se han obtenido en un sistema con una carga de trabajo nula y un tiempo de muestreo de 500 ms.

Como puede observarse en la figura y teniendo en cuenta el amplio espectro de las pruebas, con duraciones que van de 6 (100 iteraciones) a 300 segundos (5000 iteraciones) de gasto de CPU, la estimación del gasto de CPU se ajusta bastante bien al gasto de CPU en modo usuario calculado por el comando `time`. La diferencia, en media, existente entre ambos tiempos y en todas las pruebas es aproximadamente el 7%, y puede explicarse debido, principalmente, a que no se tiene en cuenta el gasto de CPU de los *threads* de carácter administrativo de la máquina virtual como pueden ser el recolector de basura o el manejador de señales. Naturalmente si redujésemos el tiempo de muestreo la desviación sería menor pero a costa de que los propios mecanismos de control de recursos añadidos introducirían una mayor penalización en el sistema, consumirían más recursos.

Tras la realización de diferentes tipos de pruebas, hemos observado que existen dos factores que influyen en la estimación del gasto de CPU, la arquitectura multiprocesador y la carga del sistema. En el caso de las arquitecturas multiprocesador, y con una prueba compuesta por dos *threads*, el tiempo estimado es aproximadamente la mitad del tiempo devuelto por `time`. Esto se debe a que cada *thread* se ha ejecutado en un procesador, pero en cada intervalo de muestreo se reparte el tiempo entre los dos *threads* activos (la mitad para cada uno), cuando en realidad cada uno ha consumido un intervalo completo de CPU. El problema es debido a que la función `HasThreadRun` de JVMPI sólo indica si el *thread* se ejecutó en el intervalo, pero no en qué procesador. La solución no es fácil y no basta con multiplicar la duración del intervalo de muestreo por el número de procesadores, ya que podría ocurrir que dos *threads* de una aplicación se ejecutaran alternativamente en un único procesador, con lo que se estaría en el caso de la estimación de consumo en un monoprocesador. Este problema detectado, de momento, no está resuelto y en la actualidad estamos explorando diversas alternativas que permitan su resolución.

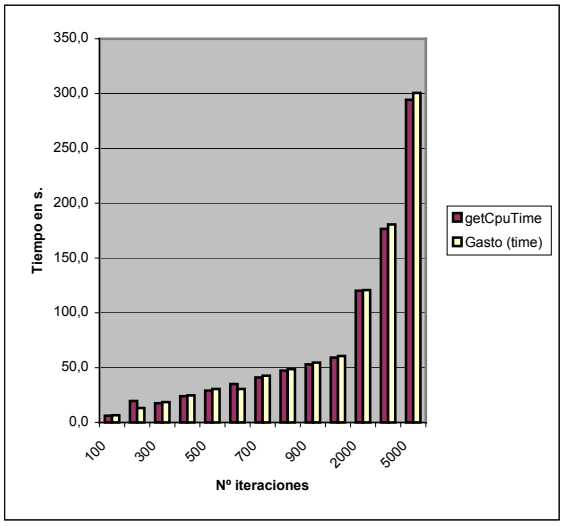

Figura 4

La carga del sistema es otro factor que influye negativamente en la estimación del gasto. En concreto cuando la carga del sistema es alta el porcentaje de tiempo de CPU que el sistema operativo dedica a cada *thread* decrece, con lo que los *threads* ejecutan menos tiempo en cada intervalo de muestreo, pero el algoritmo de estimación sigue asignando a cada *thread*, como gasto de CPU, la duración de cada intervalo de muestreo dividido por el número de *threads* activos en el mismo, con lo que el tiempo estimado es mayor que el consumo real. Con relación a este problema detectado, aunque no se prevé que las aplicaciones de red que se ejecutan en la máquina virtual Java compitan con otras aplicaciones del sistema por la CPU, se está estudiando mejorar el algoritmo para que tenga en cuenta la carga de trabajo del sistema operativo.

4.2 Impacto del control de recursos

Para comprobar el impacto de los controles en los tiempos de respuesta de las aplicaciones hemos utilizado una aplicación con gasto intensivo de CPU y memoria. Las pruebas, con diferentes duraciones (iteraciones), se han ejecutado con y sin los controles incorporados, obteniéndose los resultados de la Figura 5.

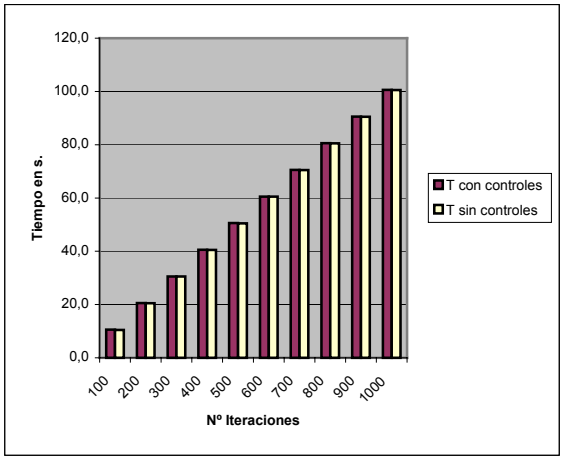

Figura 5

Los tiempos de consumo de CPU, medidos con el comando `time`, son prácticamente iguales, no existiendo una diferencia entre ambos por encima de los 100 milisegundos. Con lo cual se puede concluir que los mecanismos de control añadidos no introducen, en los tiempos de respuesta, ninguna sobrecarga añadida.

5 Trabajos relacionados

El control de recursos, en el campo de las redes activas y las redes programables, es desde hace varios años objeto de estudio, lo que ha dado lugar al diseño e implementación de diversas soluciones que van desde la incorporación del control de recursos en las capas más bajas de la arquitectura de los nodos activos (*NodeOS* o sistema operativo del nodo) [12], hasta su incorporación en el código de las aplicaciones de red [5][6], pasando por la máquina virtual Java en la que se apoyan los diferentes entornos de ejecución [7][8].

En [17] se propone incluir en la definición de Java el control del consumo de recursos por aplicación, utilizando para ello el concepto *isolate* recogido en la propuesta [18], que consiste, básicamente, en incorporar estado (p.e. montículos separados) a cada aplicación que se ejecuta en la máquina virtual Java. En caso de que finalmente se incorporase el control de recursos en la definición de Java, la solución propuesta en este artículo sería plenamente compatible y sólo sería necesario dejar de utilizar la biblioteca `ctrl_rec`.

Otras líneas de trabajo siguen desarrollando, utilizando técnicas de reescritura de *bytecodes* [19], la incorporación del control de recursos en las aplicaciones. Este enfoque tiene una clara ventaja en dispositivos dotados de procesadores Java, ya que el control se incluye directamente en las aplicaciones sin la necesidad de modificar otras partes del entorno.

El *NodeOS* es el punto elegido por [20] para incorporar el control de recursos, mediante la inserción de nuevas extensiones (aplicaciones) en el sistema operativo Janos utilizando una técnica mixta formada por controles estáticos (lenguaje de programación específico) y dinámicos (ideas tomadas del enrutador Click [21]).

Bees [22] es un entorno de ejecución sucesor de ANTS que, al igual que éste, se apoya en la máquina virtual Java, JanosVM, a la que se ha incorporado el concepto de *team* (proceso), para aislar la ejecución y el consumo de recursos de cada aplicación.

Por último citar el control de recursos de la plataforma XenoServer [23] que usa el concepto de máquina virtual, en el que ejecuta un sistema operativo convencional (Linux o FreeBSD) o no, para implementar los controles.

6 Conclusiones

La solución propuesta en este artículo permite controlar el consumo de CPU y memoria de las aplicaciones que se ejecutan en el nodo SARA de forma bastante precisa. Sus principales características son la portabilidad, el escaso coste de aplicar los controles y la facilidad para incorporar el control de nuevos recursos. Entre los aspectos que se necesita mejorar está que el algoritmo no es aplicable a entornos multiprocesador.

Otra importante característica de la solución aquí propuesta es que no es una solución particular para entornos de ejecución en nodos de una red programable, sino que puede utilizarse en cualquier entorno basado en Java, en el que se necesite controlar el gasto de los recursos del sistema.

Agradecimientos

El presente trabajo se ha realizado en el marco del proyecto del Ministerio de Ciencia y Tecnología AURAS TIC2001-1650-C02-01.

Referencias

[1] J. Gosling, B. Joy, G. Steeland G. Bracha. *The Java Language Specification*. 2nd Ed. Addison-Wesley, 2000.

[2] M. Calderón, M. Sedano y S. Eibe. *Principios y Aplicaciones de las Redes Activas*. Proc. of JITEL '99, Madrid, España.

[3] Andrew T. Campbell, Herman G. De Meer, Michael E. Kounavis, Kazuho Miki, John B. Vincente, and Daniel Villela. *A survey of programmable networks*. Computer Communication Review, 29(2):7-23, April 1999.

[4] Konstantinos Psounis. *Active networks: Applications, security, safety, and architectures. IEEE Communications Surveys, 2(1), Q1 1999*.

[5] W. Binder, J. Hulaas and A. Villazon. *Portable Resource Control in Java: The J-SEAL2 Approach*. ACM OOPSLA'01, Tampa, FL, October 2001.

[6] G. Czajkowski and T. von Eicken. *JRes: A Resource Control Interface for Java*. In Proceedings of ACM OOPSA'98, Vancouver, BC, Canada, October 1998.

[7] G. Back, W. Hsieh and J. Lepreau *Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java*. Proceedings of 4th OSDI, San Diego, CA, October 2000.

[8] P. Tullmann, M. Hibler and J. Lepreau. *Janos: A Java-oriented OS for Active Network Nodes*. In IEEE Journal on Selected Areas of Communication, March 2001.

[9] D. Larrabeiti, M. Calderón, A. Azcorra and M. Urueña. *A practical approach to Network-based processing*. IEEE 4th International Workshop on Active Middleware Services. IEEE Computer Society, pp. 3-10, ISBN: 0-7695-1721-8. Edinburgh, Scotland. July 2002.

[10] IETF Forwarding and Control Element Separation http://www.ietf.org/html.charters/forces-charter.html

[11] K. L. Calvert. *Architectural Framework for Active Networks version 1.0*. Active Network Working Group. July 27, 1999.

[12] L. Peterson, Y. Gottlieb, M. Hibler, P. Tullmann, J. Lepreau, S. Schwab, H. Dandekar, A. Purtell and J. Hartman. *An OS Interface for Active Routers*. In IEEE Journal on Selected Areas in Communications, 2001.

[13] A. Rudys and D. S. Wallach. *Termination in Language-based Systems*. Proceedings of the 2001 Network and Distributed System Security Symposium, San Diego, CA, February 2001.

[14] Sun Microsystems, Inc. *Java Virtual Machine Profiler Interface (JVMPI)*.

http://java.sun.com/j2se/1.4/docs/guide/jvmpi/index.html

[15] D. Viswanathan and S. Liang. *Java Virtual Machine Profile Interface*. IBM Systems Journal, vol. 39, no. 1, 2000.

[16] Sun Microsystems, Inc. *Java Native Interface (JNI)*. http://java.sun.com/j2se/1.4/docs/guide/jni/index.html

[17] G. Czajkowski, S. Hahn, G. Skinnerand and P. Soper. *Resource Consumption Interfaces for Java Application Programming – a Proposal*. ECOOP'02 Workshop on Resource Management for Safe Languages, Malaga, Spain, June 2002.

[18] JSR 121. *Application Isolation API Specification*. http://web1.jcp.org/en/jsr/detail?id=121

[19] W. Binder and B. Lichtl. *Resource Accounting in a J2ME Environment*. ECOOP'02 Workshop on Resource Management for Safe Languages, Malaga, Spain, June 2002.

[20] P. Patel, and J. Lepreau. *Hybrid Resource Control of Active Extensions*. OPENARCH'03, San Francisco, CA, April 2003.

[21] R. Morris, E. Kohler, J. Jannotti and M. F. Faashoek. *The Click Modular Router*, in Proceedings of the 17th ACM Symposium on Operating Systems Principles, 1999.

[22] T. Stack, S. Eide and J. Lepreau. *Bees: A Secure, Resource-Controlled, Java-Based Execution Environment*. OPENARCH'03, San Francisco, CA, April 2003.

[23] S. Hand, T. Harris, E. Kotsovinos and I. Pratt. *Controlling the XenoServer Open Platform*. OPENARCH'03, San Francisco, CA, April 2003.