Un Modelo de Threads Reactivos Usuario-Kernel para Java

Carlos Pineda G.[1] **Jordi García A.**[2] **Jorge H. Flores**[1]

[1] *Centro de Investigación en Computación*
Instituto Politécnico Nacional
Unidad Profesional Adolfo López Mateos
México DF 07738
carlospinedag@hotmail.com
jhflores@cic.ipn.mx

[2] *Departament d'Arquitectura de Computadors*
Universidat Politècnica de Catalunya
Jordi Girona 1-3 C6-E207 Campus Nord
Barcelona 08034, España
Fax: (+34) 93 4017055
jordig@ac.upc.es

Resumen

En este trabajo se presenta un modelo de threads usuario-kernel para Java, eficiente, altamente escalable, y orientado a la construcción de aplicaciones Internet de alto desempeño que requieren crear miles de threads y mantener igual número de conexiones simultáneas (p.e. servidores Web y servidores proxy). El modelo incluye un mecanismo de planificación reactiva que permite gestionar en forma implícita operaciones de I/O no bloqueantes sobre threads de usuario cooperativos.

Palabras clave: Java threads, User-Kernel threads, MxN threads, reactive threads, non-blocking I/O.

1 Introducción

Una *aplicación Internet* (AI) es un cliente, un servidor, o un cliente-servidor, que acepta conexiones de otros clientes y puede conectarse a otros servidores. Ejemplos de aplicaciones Internet son los servidores Web y los servidores proxy. Una AI se caracteriza por el número de transacciones por segundo que puede realizar (*throughput*), su escalabilidad ante el incremento de la carga (*load scalability*), y su escalabilidad ante el incremento de los recursos del sistema (*system scalability*). El throughput de una AI crece linealmente con respecto al número de conexiones hasta alcanzar un máximo. La escalabilidad de carga es una medida del número de clientes adicionales que puede soportar la aplicación manteniendo el throughput en una vecindad de su valor máximo.

El funcionamiento de una AI típica incluye las siguientes operaciones: aceptar una conexión, leer un requerimiento, ejecutar una tarea, y enviar el resultado al cliente que originó el requerimiento. Para cada conexión la aplicación debe crear un vehículo de ejecución.

Java es actualmente el lenguaje más popular para la programación de aplicaciones Internet, sin embargo, el throughput y la escalabilidad son limitados por el número de threads que la maquina virtual de Java (JVM) puede crear y gestionar, debido a restricciones impuestas por el sistema operativo [1].

En este artículo presentamos el diseño de un nuevo modelo de threads para Java. Este modelo permite la creación de miles de threads de usuario los cuales ejecutan sobre uno o más threads de kernel. También presentamos un algoritmo para la planificación reactiva de los threads, el cual integra de manera implícita la gestión de operaciones de I/O no bloqueantes. La combinación del modelo de threads de usuario-kernel con la planificación reactiva permite la programación de IA con la posibilidad de crear miles de threads desacoplados de las operaciones de I/O, incrementando así el throughput y la escalabilidad.

2 Trabajos relacionados

Alpern *et al.* del T. J. Watson Research Center de IBM, desarrollaron un sistema de threads MxN quasi-preventivos para Jalapeño [2], una máquina virtual para la ejecución de servidores Java. Jalapeño mapea los threads Java sobre procesadores virtuales que son implementados como pthreads de AIX. Para cada procesador físico se establece un procesador virtual. Es posible utilizar procesadores virtuales adicionales para gestionar la latencia de operaciones I/O. En Jalapeño los threads son cooperativos y sólo pueden ceder el control en puntos predefinidos por el compilador (*yield points*), lo cual es requerido para el funcionamiento del *garbage collector* de la máquina virtual. Los locks se implementan a cuatro niveles: *processor* locks, *thin* locks, *thick* locks, y monitores.

Welsh *et al.* de la Universidad de California [3], diseñaron una arquitectura que permite construir aplicaciones Internet de alto desempeño mediante etapas dirigidas por eventos interconectadas por colas, *SEDA (staged event-driven architecture*). SEDA usa un conjunto de controladores dinámicos para condicionar la carga en las etapas bajo un régimen de grandes fluctuaciones de carga. Utiliza una biblioteca de clases para Java denominada NBIO (*non-blocking* I/O) que permite implementar operaciones de I/O no bloqueantes.

SGI desarrolló una biblioteca conocida como State Threads [4] derivada de la biblioteca Netscape Portable Runtime library (NSPR), que permite escribir aplicaciones para Internet eficientes y altamente escalables. State Threads combina el modelo de threads cooperativos Mx1 con una arquitectura de máquina de estados dirigida por eventos (*event-driven state machine*), sin embargo no se trata de una biblioteca multithread de propósito general sino orientada específicamente al tipo de aplicaciones Cliente/Servidor de Internet.

3 Threads Reactivos

El modelo de los threads reactivos es MxN: M threads de Java que ejecutan sobre N threads de kernel. Los threads de Java son creados como threads de usuario cooperativos, y los threads de kernel son implementados como pthreads de Linux.

Procesadores virtuales

Para cada thread de kernel se construye una estructura que llamamos *procesador virtual de tipo CPU* (cpu_vp). Cada cpu_vp cuenta con cuatro colas de threads: la cola de ejecución, la cola de inicio, la cola de entrada, y la cola *garbage*. La cola de ejecución es una cola circular doblemente enlazada que contiene los threads listos para ser ejecutados. La cola de inicio es una cola lineal doblemente enlazada que contiene los threads creados cuyo contexto no ha sido todavía inicializado (stack, stack pointer, etc). La cola de entrada es una cola lineal doblemente enlazada que sirve como interfaz de entrada para threads ya inicializados, los cuales han sido enviados desde otro procesador virtual. La cola *garbage* es una cola lineal simple que contiene los threads terminados y listos para ser eliminados.

Creación de threads reactivos

Un thread reactivo puede ser creado en estado de ejecución (*running*) o en estado suspendido (*suspended*). Si un thread es creado en estado de ejecución entonces es colocado en la cola de inicio. Si un thread es creado en estado suspendido, es posible posteriormente ponerlo en estado de ejecución, así mismo, un thread en estado de ejecución puede ponerse en estado suspendido.

En la cola de ejecución existe un thread cooperativo permanente el cual realiza la función de despachador de threads. Con el fin de optimizar los cambios de contexto, el acceso a la cola de ejecución lo realiza exclusivamente el thread despachador, por lo que esta cola no requiere sincronización. El acceso a las colas de inicio y de entrada requiere ser sincronizado mediante locks, ya que diferentes procesadores virtuales pueden tratar de insertar simultáneamente threads en dichas colas.

Cuando el thread despachador recibe el control, inicializa el contexto de los threads que están en la cola de inicio. Posteriormente los threads inicializados son transferidos al final la cola de entrada. Entonces los threads de la cola de entrada son transferidos a la cola de ejecución, y el thread despachador cambia al contexto del siguiente thread en la cola de ejecución, o bien, entra en un estado de espera pasiva si no hay más threads (en la cola de ejecución, en la cola de inicio, o en la cola de entrada).

Es posible crear dos tipos de threads: *detach* y *join*. Un thread de tipo *detach* libera automáticamente la memoria que ocupa cuando termina, utilizando para tal efecto, la cola *garbage* del cpu_vp donde ejecuta. Por otro lado, cuando termina un thread de tipo *join*, no libera por sí mismo la memoria que ocupa, en su lugar, notifica que ha terminado y es algún otro thread que espera la terminación del primero, el encargado de liberar la memoria correspondiente.

Con el fin de optimizar la creación de threads, los stacks de los threads se adquieren de un *pool* de bloques de memoria pre-alojados al inicio de la JVM. Cada vez que un thread termina, el bloque de memoria correspondiente es nuevamente enlazado al pool, de tal manera que queda disponible para la creación de otro thread. Si no hay más bloques disponibles en el pool de threads, el thread actual espera hasta que otro thread de usuario libera un bloque de memoria. El tamaño de cada bloque en el pool es una constante múltiplo del tamaño de página del sistema.

Inicialización del contexto

La inicialización del contexto de un thread consiste en establecer la dirección para el inicio de la ejecución, el contenido inicial del stack, y los registros *stack pointer* y *base pointer*. Debido a que actualmente los threads reactivos no soportan la expansión dinámica del stack, con el propósito de contar con un mecanismo de protección en caso de desbordamiento del stack, la primera página del stack se protege de la lectura y escritura mediante la llamada a sistema *mprotect*, de tal manera que si el stack llegara a desbordarse entonces se produciría una señal SIGSEGV.

En el microprocesador Pentium el stack crece desde las posiciones altas a las posiciones bajas de memoria, de tal manera que el desbordamiento del stack se detecta cuando el procesador trata de escribir las posiciones más altas del stack. La figura 1 muestra el contenido del stack inicial de un thread reactivo.

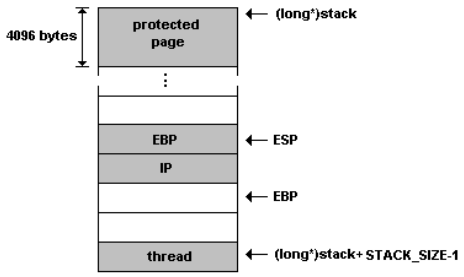

Fig 1: Stack inicial de un thread reactivo

Cambio de contexto

Al hacer un cambio de contexto no es necesario comprobar que la cola de ejecución esté vacía debido a que por lo menos existe el thread despachador el cual fue creado al momento de iniciar el cpu_vp y permanece en ejecución siempre. Tampoco es necesario utilizar un lock para sincronizar el acceso a la cola de ejecución, ya que los nuevos threads creados y aquellos que son enviados por otros procesadores virtuales son colocados en las colas de inicio y de entrada, respectivamente (las cuales sí se sincronizan mediante locks). Sólo el thread despachador puede transferir los threads de las colas de inicio y de entrada a la cola de ejecución, lo cual se hace en el momento que el thread despachador recibe el control. Este mecanismo permite ejecutar primero los nuevos threads y después los threads enviados por otros procesadores virtuales, esto es: después de que se inicializa el contexto de los threads de la cola de inicio, estos son insertados al principio de la cola de entrada, la que a su vez es insertada en la cola de ejecución, entre el thread despachador y el primer thread reactivo (si existe alguno). Tampoco se requiere sincronizar la operación de extracción de un thread de la cola de ejecución, ya que sólo el thread puede extraerse a sí mismo de la cola.

Sincronización

Los threads reactivos implementan mecanismos de sincronización al nivel de cpu_vp y al nivel de thread de usuario. Al nivel de cpu_vp se implementa *spin locks* y locks FIFO. Un spin lock es un entero de 32 bits igual a 1 si se encuentra bloqueado o igual a 0 si se encuentra desbloqueado. La adquisición de un spin lock es en base a un ciclo de espera activa (*busy-wait*).

Por otra parte, un lock FIFO cuenta con una cola de espera FIFO que permite asegurar que no se presente una situación de *starvation* en los cpu_vp. La implementación de los spin locks y los locks FIFO se hace mediante una instrucción de intercambio atómico, en particular para el Pentium se utiliza la instrucción de intercambio a 32 bits *xchg m32,r32* la cual intercambia el registro r32 y la memoria m32 en forma atómica.

La sincronización al nivel de threads de usuario se realiza mediante mutex recursivos, implementados mediante una instrucción del tipo “comparar e intercambiar” en forma atómica. En particular para el Pentium se utiliza la instrucción de comparación e intercambio a 32 bits *lock cmpxchg m32,r32* la cual compara el contenido de la memoria *m32* con el contenido del registro EAX, si son iguales entonces el contenido del registro *r32* se escribe a la memoria *m32*, de lo contrario el registro EAX recibe el contenido de la memoria *m32*.

Planificación reactiva

Cuando un thread ejecuta una operación bloqueante éste puede ser transferido a otro procesador virtual que denominamos *procesador virtual reactivo*, con el fin de liberar al cpu_vp de cambios de contexto superfluos en la cola de ejecución. Los procesadores virtuales reactivos se

implementan como pthreads de Linux y funcionan como monitores de los threads que ejecutan operaciones bloqueantes.

Actualmente tenemos definidos tres tipos de procesadores virtuales reactivos, los de tipo *net_vp* que gestionan aquellos threads que ejecutan las funciones *recv*, *send* y *accept* no completadas, los de tipo *io_vp* que gestionan aquellos threads que ejecutan las funciones *read* y *write* no completadas, y los de tipo *wait_vp* que gestionan los threads que ejecutan *sleep*. Los procesadores virtuales net_vp, io_vp y wait_vp entran en un estado de espera pasiva hasta que el sistema operativo les notifique que una o varias operaciones bloqueantes han sido completadas. Entonces los procesadores virtuales reactivos insertan los threads que han completado las operaciones bloqueantes en las colas de entrada de los cpu_vp correspondientes.

Extraer un thread de la cola de ejecución representa un costo, por lo tanto sólo es conveniente extraer los threads cuando el tiempo de bloqueo es considerable. La heurística que utiliza un cpu_vp para decidir cuándo extraer un thread de la cola de ejecución es asociar una constante SPIN a cada operación bloqueante de acuerdo al algoritmo mostrado a continuación, dónde es posible observar que la heurística esta orientada a reducir el tamaño de la cola de ejecución cuando se ejecutan operaciones bloqueantes.

```
for (;;)
{
  spin = SPIN
  while (spin > 0)
    if (blocking_operation_completed)
      return
    else
    {
      spin = spin – 1
      yield
    }
  extract_thread_from_running_queue
  insert_thread_into_reactive_virtual_processor
  yield
}
```

La figura 2 muestra el ciclo de vida de un thread reactivo. Primeramente el thread es creado tomando un bloque del pool de threads, este thread es insertado en la cola de inicio. Entonces el despachador del cpu_vp crea un contexto para el thread y lo envía a la cola de entrada. Después el despachador envía los threads existentes en la cola de entrada a la cola de ejecución. Si un thread ejecuta una operación de I/O bloqueante, el thread es enviado a un procesador virtual reactivo seleccionado mediante un mecanismo de *round robin*. Cuando la operación de I/O bloqueante es completada por el procesador reactivo, el thread correspondiente es enviado a la cola de entrada de un cpu_vp seleccionado también usando round robin . Si un thread termina entonces es enviado a la cola garbage. Cuando el thread despachador ejecuta, los threads en la cola garbage son transferidos al pool de threads.

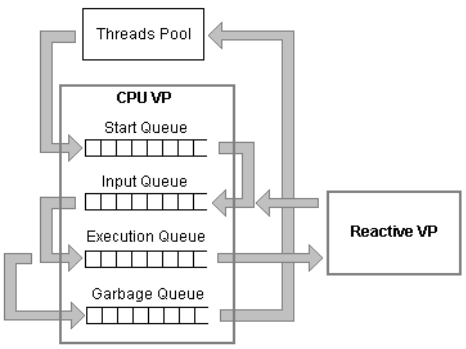

Fig. 2: Ciclo de vida de un thread reactivo

La figura 3 muestra un ejemplo de una aplicación en la que se ha definido dos cpu_vp, un cpu_vp por cada CPU físico. En este ejemplo se ha definido un io_vp para gestionar las operaciones sobre disco, y un net_vp para gestionar las operaciones sobre la red.

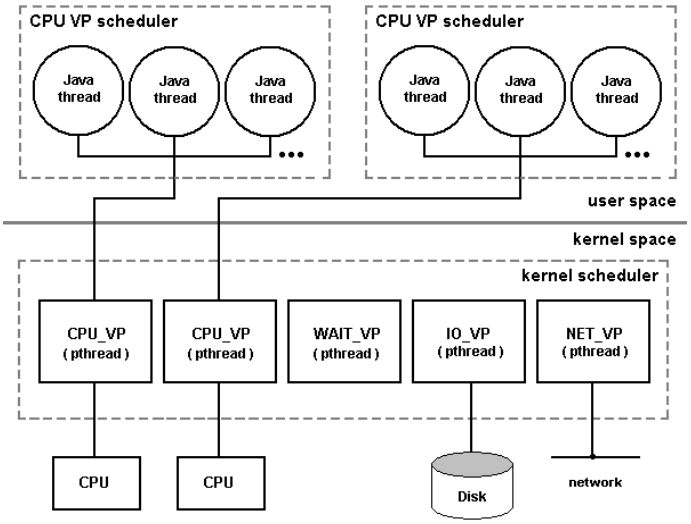

Fig 3: Una aplicación con dos procesadores virtuales cpu_vp.

4 Implementación de los threads Reactivos en la JVM

Con el propósito de implementar los threads reactivos en la máquina virtual de Java, construimos un compilador estático de bytecode Java, ASAP, el cual genera código en lenguaje C a partir de archivos de clases .class, y para cada archivo .class genera una biblioteca dinámica.

ASAP es un compilador AOT (*Ahead Of Time*) debido a que la compilación se realiza en forma estática, lo cual representa algunas ventajas sobre la compilación JIT, como puede verse en los trabajos [5][6][7]. ASAP incluye técnicas de optimización de tipo general y mecanismos de optimización específicos para Java.

Para cada invocación a los métodos de la clase java.lang.Thread, ASAP genera llamadas a funciones de la biblioteca de threads reactivos, así mismo para los métodos de notificación de la clase java.lang.Object. El thread principal de una aplicación es un thread nativo de Java (pthread); los threads subsecuentes serán creados como threads reactivos.

Construimos también una interface JIT (*Just In Time*) para la versión clásica de la JVM de Sun [8]. La interface JIT es una DLL que es cargada al momento que la JVM inicia. La interface JIT por defecto en la versión 1.2.2 de la JVM es “sunmjit”, pero es posible cambiar el JIT definiendo la opción –Djava.compiler=jit cuando el programa “java” es ejecutada.

Cuando la JVM trata de cargar una clase, nuestra interface JIT busca en el disco la DLL correspondiente, si existe entonces la biblioteca es cargada por la JVM, de otra manera los métodos de la clase serán interpretados.[1]

[1] En una versión futura incluiremos un compilador JIT para ejecutar los métodos de las clases no compiladas estaticamente.

5 Un servidor Web con conexiones persistentes

En esta sección se presenta la aplicación de los threads reactivos en la construcción de un servidor Web con conexiones persistentes compatible con el protocolo HTTP/1.1 [9]. Se usó httperf [10] para medir el rendimiento del servidor Web cuando es ejecutado por maquinas virtuales de Java con diferentes implementaciones de threads.

Los experimentos se realizaron sobre una red de computadoras con Pentium III a 650 Mhz y Linux 2.2.14-5.0smp conectadas mediante un switch Emulex cLAN con un ancho de banda de 1.25 Gb/s full duplex. Cada experimento consistió en 1,000 conexiones y 5 requerimientos por conexión. Cada requerimiento solicitó un archivo de 10,000 bytes. Las figuras 4 y 5 muestran el throughput del servidor Web en términos de tráfico en la red (Net I/O) y respuestas por segundo, correspondientes a una tasa de 10 a 120 conexiones por segundo (50 a 600 requerimientos por segundo).

La gráfica **sunwjit 1.2.2** corresponde al throughput del servidor Web ejecutado por una JVM con JIT versión 1.2.2. Las gráficas **Hotspot 1.2.2** y **Hotspot 1.4.2** corresponden al throughput del servidor Web ejecutado por Hotspot[2] en sus versiones 1.2.2 y 1.4.2. La gráfica **R.Threads** corresponde al throughput del servidor ejecutado por la JVM con nuestra interface JIT que implementa los threads reactivos. Es posible observar que el rendimiento cuando se utiliza threads reactivos es mucho mejor, esto es debido a que nuestro modelo de threads permite que el servidor Web cree cientos de threads por segundo desacoplados de las operaciones de I/O.

6 Conclusiones

El modelo de threads reactivos para Java cuenta con un mecanismo de planificación reactiva eficiente y altamente escalable, orientado a la construcción de aplicaciones Internet de alto desempeño que requieren crear miles de threads y mantener un número igual de conexiones simultáneas. Los resultados que hemos obtenido nos hacen ver las enormes posibilidades que tiene esta implementación de threads MxN, sobre todo en aplicaciones que demandan un alto desempeño y la necesidad de crear un gran número de threads que realizan operaciones de I/O.

Referencias

[1] U. Drepper, I. Molnar, **The new Native POSIX Thread Library for Linux**, Red Hat Inc., January 2003.

[2] B. Alpern, *et al.*, **The Jalapeño vitual machine**, IBM Systems Journal, Vol. 39, No. 1 2000.

[3] Matt Welsh, *et al.*, **SEDA: An Architecture for Well-Conditioned Scalable Internet Services**; Computer Science Division, University of California, Berkeley, 2001.

[4] Silicon Graphics Inc, **State Threads for Internet Applications**, December 2002.

[5] Robert Fitzgerald, *et al.*, **Marmot: An Optimizing Compiler for Java**, Microsoft Research, June 1999.

[2] Hotspot es un compilador optimizador de alto rendimiento desarrollado por Sun Microsystems.

[6] Todd A. Proebsting, *et al.*, **Toba: Java For Applications, A Way Ahead of Time (WAT) Compiler**, The University of Arizona.

[7] Todd Smith, *et al.*, **Practical Experiences with Java Compilation**, Silicon Graphics, Inc.

[8] Frank Yellin, **The JIT Compiler Interface Specification**, Sun Microsystems, http://java.sun.com/docs/jit_interface.html

[9] R. Fielding, *et al.*, **Hypertext Transfer Protocol HTTP/1.1**, Network Working Group, Internet RFC 2616, June 1999.

[10] David Mosberger, Tai Jin, **httperf: A Tool for Measuring Web Server Performance**. Hewlett-Packard Research Labs, 1998.

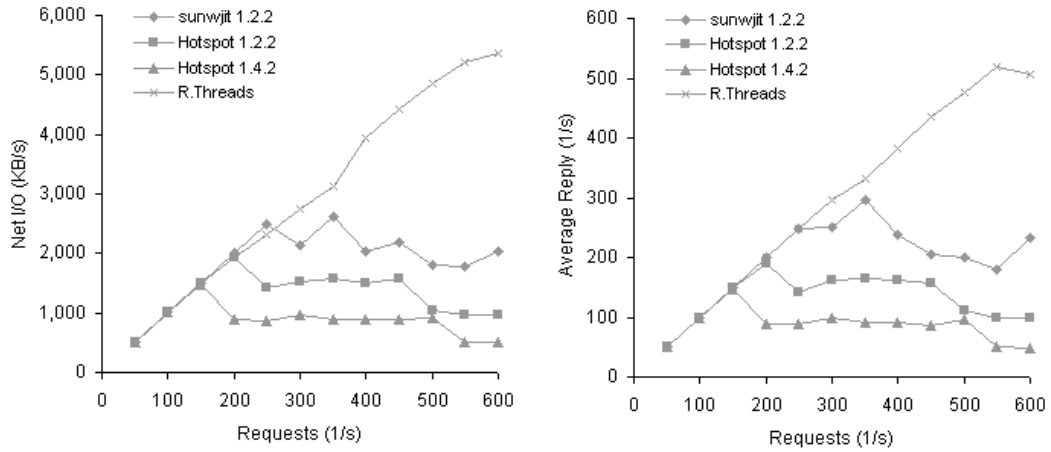

Fig 4: Desempeño del servidor Web cuando se ejecuta en una computadora con un procesador

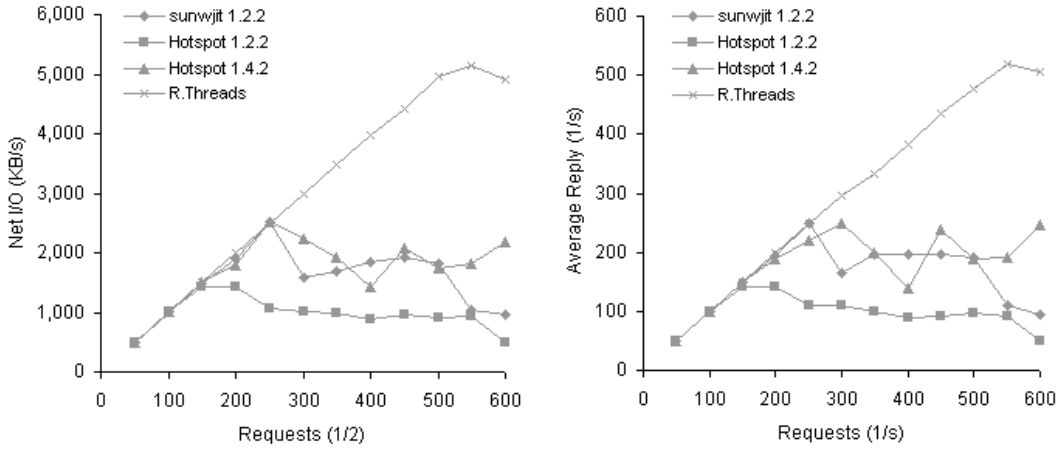

Fig 5: Desempeño del servidor Web cuando se ejecuta en una computadora con dos procesadores