Proyecto Cherokee: Diseño, implementación y aspectos de rendimiento de servidores web

Alvaro López Ortega <alvaro@gnu.org>

***Abstract*— Este artículo presenta los aspectos críticos que intervienen en el desarrollo de un servidor web. En primer lugar, se exponen los diferentes diseños clásicos, detallando las características de cada uno de ellos y estudiando sus mejoras; exponiendo en último lugar el diseño realizado en el servidor web Cherokee.**

A continuación, se estudian métodos para la implementación de un servidor de alto rendimiento, así como posibles puntos de mejora y aspectos del desarrollo para lograr un funcionamiento más eficiente.

Por último se presentan datos detallados sobre el rendimiento de cada uno de los principales servidores web libres y sus respectivas arquitecturas, así como su comportamiento en diferentes escenarios. Estas pruebas presentan la eficacia real de cada servidor y por lo tanto de su arquitectura.

***Index Terms*— servidor web, diseño, rendimiento, Cherokee**

I. Introducción

DESDE hace años, Internet ha estado creciendo de forma espectacular, tanto en número de usuarios como en la cantidad de información que transmite. En especial el World Wide Web se ha posicionado como la aplicación más importante de la pasada década. Este crecimiento ha introducido nuevos problemas en numerosos ámbitos: búsqueda y recuperación de información, categorización, filtrado, técnicas de transmisión y diseño de servidores, entre otras. Este documento se va a centrar en el problema de diseño de servidores Web. Es necesario que el software que se ejecuta en los nodos que forman Internet trabajen de una forma eficiente y sean capaces de satisfacer todas las necesidades de servicio de información que reciban. Actualmente, existen sitios Web muy activos que necesitan servir varios millones de peticiones por minuto, y en estas ocasiones es cuando toma especial relevancia el rendimiento del servidor. Existen numerosos factores que intervienen en el funcionamiento de un servidor, y que por lo tanto están directamente relaccionados con su eficiencia, siendo estos, los principales objetos de estudio del artículo.

Por otro lado, se presentará un proyecto de Software Libre llamado Cherokee. Se trata de un servidor web, con un diseño modular basado en plug-ins. El objetivo de este diseño es poder dotar al servidor de toda clase de funcionalidades y extensiones, posiblemente desarrolladas por terceras partes. En él se han includo, como se expone a continuación, los diseños e implementaciones que han demostrado ser más eficientes, de forma que aun siendo más grande, flexible y complejo que otros servidores web, también soporta un mayor volumen de tráfico.

Este artículo también compara el rendimiento de los servidores web libres más utilizados. El estudio se ha realizado en máquinas de diferentes características para obtener datos de cada una de ellas: tanto SMP como monoprocesador. En todos los casos el sistema operativo ha sido GNU/Linux.

II. Diseño

El objetivo de diseño se centra en conseguir un alto nivel de paralelismo a la hora de servir las conexiones. Es posible que el servidor tenga que enviar varios millones de respuestas por minuto, así que es crítica la forma en la que son manejadas. Resultaría completamente inútil un servidor que responda una por una las conexiones no concurrentemente. En este hipotético diseño, existiría una cola de conexiones en espera - la del propio sistema operativo - en la que se almacenarían hasta el momento en que, previo proceso de las anteriores, fuese la primera de la cola. Este diseño es inviable por varias razones, la principal de ellas es la alta latencia que podría llegar a tener una conexión, incluso rebasando el tiempo limite.

Resulta especialmente importante, el estudio de los siguientes puntos [1] en cada una de las arquitecturas:

- Velocidad de respuesta
- Escalado con alta carga
- Calidad de la respuesta en situación de sobrecarga

A. Servidores basado en procesos

El diseño de servidores basados en procesos fue el predecesor de todos los demás diseños que se presentarán a continuación. Se basa en la obtención de paralelismo mediante la duplicación del proceso de ejecutación. Este es el método sobre el que se implementaba el servidor de NCSA [2], y porteriormente Apache [3]. Existen varios diseños basados en procesos; el primero de ellos y más simple es en el que el proceso principal espera la llegada de una nueva conexión y en ese momento, se duplica creando una copia exacta que atenderá esta conexión. Sobre esta opción de diseño caben optimizaciones importantes, como las que incluyó Apache con la técnica de pre-fork.

El diseño de un servidor basado en procesos con pre-fork consiste en la creación previa de un grupo de procesos y su mantenimiento hasta que sea necesaria su utilización. El aumento en el rendimiento de este diseño frente a uno basado en procesos es grande; es muy posible que en la creación de los procesos se realice en tiempo que de otra forma sería perdido y que incrementaría la latencia de la conexión.

Las principales ventajas de este diseño residen en su simplicidad de implementación y su seguridad. En la implementación de uno de estos servidores únicamente hay que tener en

cuenta un hilo lógico de ejecución. El programador se abstrae del hecho de que es necesario conseguir concurrencia en la respuesta de las peticiones e implementa uno a uno, los pasos por los que debe pasar una conexión; posteriormente, la concurrencia en ejecución vendrá dada por la duplicación de procesos. El aspecto positivo en cuanto a la seguridad consiste en el hecho de que cada conexión se responde desde un proceso completamente autónomo. Si existiese algún problema de seguridad en la implementación del servidor, sólo afectaría al proceso que lo ejecutase, no al servidor completo.

La gran desventaja de este diseño es el bajo rendimiento. La creación de un proceso o eliminación de un proceso son tareas pesadas para el sistema operativo y consumen una gran cantidad de tiempo [4], especialmente cuando se trata de programas linkados dinámicamente - es decir, que deben buscar símbolos en librerías dinámicas del sistema. Por otro lado, el diseño adolece de otros problemas también relaccionados con el ámbito de los procesos. Las partes comunes que se podrían compartir entre procesos son duplicadas, además de no ser posible compartir ninguna información de una forma eficiente entre los diferentes procesos. Esta restricción elimina la posibilidad de la utilización de cachés comunes en el servidor así como de cualquier otra técnica de optimización que implique el acceso a un recurso compartido entre más de una conexión.

B. Servidores basados en hilos

El diseño de servidores basados en hilos (o threads), hoy en día es mucho más común que el basado en procesos. Los conceptos básicos respecto al funcionamiento de un servidor basado en procesos son aplicables también a este modelo. Las principales diferencias de los dos modelos reside en el propio concepto de hilo. Al contrario que en el caso anterior, la creación de un hilo no es tan costosa como la de un proceso. Varios hilos de un mismo proceso si pueden compartir datos entre ellos, ya que comparten el mismo espacio de direccionamiento y únicamente de publica el espacio de pila para cada uno de ellos.

El modelo de servidor basado en hilos hereda muchas de las características de los servidores basados en procesos, entre ellas la de la simplicidad en su diseño e implementación. Por otro lado, en el caso de los servidores basado en hilos, el compartir el espacio de direccionamiento en memoria implica un riesgo de seguridad del que no adolecen los servidores basado en procesos. Si uno de los hilos del proceso ejecuta una instrucción ilegal y produce cualquier tipo de fallo en el uso de la memoria, el servidor completo parará de ejecutarse.

C. Servidores basado en sockets no bloqueantes

Los servidores basados en sockets no bloqueantes, o también conocidos como servidores dirigidos por eventos, basan su funcionamiento en la utilización de lecturas y escrituras asíncronas sobre descriptores - normalmente sockets. A este método de lectura y de escritura asíncrona sobre descriptores se le conoce como entrada/salida no bloqueante.

Imaginamos que se trata de leer información de un socket en el que el buffer de entrada asociado a él está vacio. En el caso de tratarse de que el descriptor no se hubiese fijado como no bloqueante, el proceso se quedaría parado en la llamada al sistema hasta que se recibiera información. De lo contrario, si el descriptor se ha fijado en modo no bloqueante, la llamada a la función de lectura retornaría inmediatamente indicando en el código de error que no ha podido leer nueva información.

Normalmente, estos servidores utilizan una llamada al sistema que examina el estado de los descriptores con los que trabaja. Cada sistema operativo implementa una o más de estas funciones de examen de descriptores:

select	MacOS X, y sistemas antiguos
kqueue	FreeBSD, OpenBSD y NetBSD [5]
poll	La mayoría de los sistema Unix actuales
epoll	Linux 2.6

El objetivo de estas funciones es examinar el estado de un grupo de descriptores; normalmente los asociados con los sockets de cada una de las conexiones. La llamada a una de estas funciones realizará una espera - no activa - hasta el momento que uno de los descriptores sometidos a examen cambie su estado o se alcance un límite de tiempo especificado en uno de los parámetros con los que se llamó a la función.

Las ventajas de este modelos de diseño reside principalmente en su velocidad. Como se expondrá más adelante, los servidores basados en sockets no bloqueantes consiguen un alto rendimiento [6].

Por otro lado, este modelo también sufre una serie de problemas e inconvenientes. En primer lugar, la concurrencia es simulada: existe un único hilo lógico de ejecución - es decir, un sólo proceso y un sólo hilo - desde el cual se atienden todas las conexiones utilizando para ellos la compartición de tiempo dedicada a cada una de ellas. Por el hecho se tratarse de un proceso que atiende varias conexiones al mismo tiempo no debe ejecutarse ninguna instrucción que tarde un tiempo excesivo, de lo contrario, las demás conexiones permanecerán sin servicios durante este tiempo.

Otro de los problemas de este modelo se centra en la seguridad. Al igual que en el modelo de servidores basados en hilos, el espacio de direccionamiento de la aplicación completa es único, por lo cual, si una de las conexiones realiza una operación o acceso de memoria ilegal, el servidor completo dejará de ejecutarse.

Actualmente existen una serie de servidores web libre basados en este diseño de servidor, de los cuales destacan thttpd y Boa.

D. Servidores en el kernel

Los servidores empotrados en el kernel no tienen un diseño definido. Se trata de un intento de acelerar la velocidad de un servidor web clásico mediante el movimiento de su código de espacio de usuario a espacio de kernel.

El tratarse de código que se ejecuta dentro de kernel es un factor fundamental en esta clase de servidores: en primer lugar es posible optimizar algunas operaciones la interacción directa con la pila TCP/IP. Por otro lado, no es necesario intercambiar información entre espacio de kernel y espacio de usuario como

normalmente ocurre en el resto de servidores, de forma de que realizando la misma tarea, la implementación en el kernel sea mucho más eficiente.

Ahora bien, en contraposición con el beneficio que aporta en cuanto a la eficiencia, los problemas e inconvenientes de esta clase de servidores son muy grandes. Hay que tener en cuenta que cualquier problema que se produzca a nivel de kernel puede ocasionar la caida de todo el sistema completo; luego lo que en el modelo de servidores basados en procesos podría ocasionar que se perdiera una única conexión, en esta ocasión causaría la fallo del kernel al completo, y por lo tanto, cesaría la ejecución de programas en la máquina.

Por último, hay que tener en cuenta que el hecho de que estos servidores trabajasen con contenido dinámico implicaría incluir aun más código en el espacio de kernel. Esta solución es de antemano inviable, por lo cual, para solucionar este problema, las implementaciones existentes de servidores web empotrados delegan estas peticiones sobre un servidor en espacio de usuario que es el que ejecuta el código dinámico.

Actualmente existen dos servidores empotrados en el kernel: Tux [7], un servidor libre escrito por Red Hat Inc. y AFPA [8] un framework desarrollado por IBM.

III. Proyecto Cherokee

Cherokee[9] es un proyecto que nació por el interés de escribir un nuevo servidor web que mejorase algunas de las carencias de los servidores web que existían. El objetivo principal del proyecto es desarrollar un nuevo servidor con el diseño y arquitectura más eficiente, y a su vez, manteniendo la flexibilidad suficiente como para poder implementar sobre él cualquier funcionalidad que hoy en día pueda ser deseable.

El proyecto es por completo Software Libre; se ha publicado bajo licencia GPL[10] con el objetivo de que sea accesible por el mayor número de personas y exista una comunidad de usuarios y desarrolladores que lo soporten y mejoren.

	Apache	Cherokee	thttpd	Boa
Embedable	no	yes	no	no
Keepalive connections	yes	yes	yes	yes
Modules / Plug-in support	yes	yes	no	no
Virtual Servers support	yes	yes	pour	pour
Scale to SMP systems	yes	yes	no	no
/dev/epoll support	yes	yes	no	no
Authentication	yes	yes	yes	no
Encodeing support. Eg: gzip	yes	yes	no	pour
CGI	yes	yes	yes	yes
Custom error pages	yes	yes	pour	no
Cache friendly	yes	yes	yes	pour
sendfile()	yes	yes	yes	yes
https	yes	no	no	no
Other info				
Lines of code	185,000	20,000	10,000	10.000

A. Diseño

Cherokee se ha implementado con un diseño no expuesto anteriormente. Las pruebas demuestran, como se expone en este mismo documento en el punto de “Pruebas de rendimiento”, que la solución diseñada para Cherokee es más eficiente.

Se trata de un diseño híbrido que combina las carácterísticas del diseño de un servidor basado en sockets no bloqueantes con las de un servidor basado en hilos, en buscar de obtener los beneficios de ambas. En contraposición con los beneficios, los aspectos negativos de cada una de los dos diseños en los que se basa tambien son heradados. Estos aspectos negativos heredados son proporcionalmente mucho menores a los beneficios ya que tanto en el diseño basado the hilos como en el diseño basado en I/O no bloqueantes los problemas que presentan son básicamente los mismos.

El funcionamiento de este modelo es, a grandes rasgos, el de un servidor que procesa varias peticiones en cada uno de sus hilos. Estos hilos ni se crean ni se destruyen, se generan cuando arranca el servidor y permanecen vivos hasta que termina su ejecución.

La idea principal sobre la que se forjó esta forma de trabajo es la de intentar mantener la eficiencia de los servidores basados en entrada/salida no bloqueantes, al mismo tiempo que se intentaba mejorar el problema de los accesos bloqueantes por parte del servidor - por ejemplo, la apertura de un fichero del disco duro. Si uno de los hilos se bloquea momentáneamente, posiblemente el resto aun esten activos y puedan seguir procesando peticiones.

Existía un problema en el diseño de este nuevo modelo híbrido de servidor. Se plantearon varios problemas; el principal, en la gestión del único socket que puede acceptar nuevas conexiones en el servidor. Si existen varios hilos de ejecución, es necesario mantener una coherencia en la ejecución y establecer una serie de secciones críticas para asegurar el correcto funcionamiento del servidor. Durante el diseño de Cherokee se estudiaron varias posibilidades para solucionar este problema de diseño interno:

1) Hilo coordinador e hilos trabajadores
2) Hilos completamente independientes
3) Hilos independientes autogestionados

La primera de las opciones consiste en la creación de dos entidades lógicas de diseño completamente independientes: el concepto de hilo coordinador y el de hilo trabajador. En el servidor existiría un hilo coordinador y varios hilos trabajadores. El hilo que ejerciera la tarea de control sería el encargado de aceptar las nuevas conexiones al servidor y de asignarselo a un hilo trabajador. Por otro lado, la única tarea de los hilos trabajadores es la del proceso de conexiones, es decir, ejecutar las fases de la siguiente figura en cada una de las conexiones. El tratamiento de las conexiones por parte de los hilos trabajadores sigue el mismo diseño que la del proceso de un servidor que únicamente utilice I/O no bloqueante. Las pruebas demostraron que la implementación de este diseño era aceptablemente eficiente, aunque como veremos a contin-

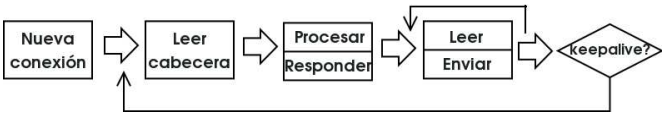

Fig. 1. Procesado de una petición HTTP

uación, susceptible de mejorar. El principal inconveniente del esquema de hilo coordinador e hilos trabajadores radica en la necesidad de mantener código diferente para cada uno de los comportamientos, con el aumento de posibilidades de fallo que esto supone.

El segundo diseño consiste básicamente en una simplificación del anterior. En este caso, todos los hilos ejecutan el mismo código y por tanto tienen el mismo comportamiento. La principal diferencia con el modelo anterior reside en que cualquiera de los hilos puede examinar el socket principal del servidor en busca de nuevas conexiones. Con este esquema la primera ventaja clara es la eliminación de la necesidad de mantener diferente código para cada tipo de hilo. Por otro lado, esta forma de trabajo tiene un problema importante: cuando se reciba una nueva conexión en el servidor, todos los threads detectarán un cambio en los descriptores a los que atienden - ya que el socket del servidor está includo en todos los hilos - y únicamente uno de ellos aceptará la conexión de forma satisfactoria, el resto recibirá un error tras llamar a la función *accept*. El disparo de este falso evento en la mayoría de los hilos genera una sobrecarga prescindible en el servidor que acasiona un rendimiento por debajo del óptimo en las pruebas realizadas.

Por último, el diseño basado en hilos independiente autogestionados ha desmostrado ser el más eficaz. Su funcionamiente se basa en la autogestión del socket del servidor entre todos los hilos, de forma de únicamente uno de ellos, pueda estar aceptando una petición desde el socket principal del servidor, para inmediatamente después dejarlo disponible para que otro thread lo pueda volver a examinar. Este diseño proporciona una bajísima acoplación, así como una implementación razonablemente sencilla y con él, se resuelve el problema de falsas señales expuesto en el caso anterior.

IV. Implementación

Son muchos los posibles actores que pueden aumentar drásticamente el rendimiento de un servidor web [11]. La mayoría de ellos están relaccionados con las llamadas al sistema que se utilizan para realizar determinadas tareas: lectura de un descriptor, escritura en un socket, propiedades de los nuevos sockets que se aceptan, etc. Es habitual la existencia de algunas funciones optimizadas con estos propósitos en la mayoría de los sistema operativos.

A continuación se describirán algunos de los que más importancia han tenido en la implementación de Cherokee:

mmap — La función mmap mapea un fichero en memoria. Esta función es especialmente útil para la implementación de un caché de ficheros en memoria. Las ventajas de la utilización de esta clase de caches radica en el decremento del número de descriptores de fichero abiertos - ya que no es necesario leer continuamente de él - y el aumento de velocidad que se experimenta en el servidor por el hecho de disponer del contenido en la memoria y no en un soporte sólido que posiblemente genere operaciones de entrada/salida bloqueantes. La implementación de caché de ficheros realizada en Cherokee, mantiene únicamente los más solicitados en memoria, para ello, gestiona el número de veces que se ha abierto cada uno con el fin de eliminar el mapeo de los ficheros poco solicitados.

writev — Esta función es básicamente la misma que *write* pero en algunas ocasiones mucho más eficaz que esta última. La principal diferencia entre las funciones *write* y *writev* consiste en que mientras que la primera de ellas únicamente admine un buffer a enviar, a *writev* es posible especificarle más de uno. En la implementación de Cherokee, writev es utilizado para enviar el buffer de la cabecera de respuesta HTTP [12] y el primer fragmento del contenido de la respuesta en una sola llamada al sistema. Como contrapunto, esta función necesita un procesado más laborioso si se produce una escritura parcial del contenido ya que hay que determinar en qué buffer y que desplazamiento se encuentra el punto desde el que hay que continuar la transmisión. [13]

sendfile — La función *sendfile* implementa la copia entre descriptores a nivel de núcleo. Una llamada a esta función es equivalente a una llamada a *read* y una a *write*, pero con la ventaja adicional que al tratarse de una copia a nivel de kernel no es necesario realizar costosas operaciones de copia de información entre espacio de kernel, el espacio de usuario y nuevamente el espacio de kernel. La experiencia de las pruebas realizadas ha demostrado que no siempre es más rápido utilizar *sendfile*, en los casos en los que se transmiten ficheros muy pequeños es más eficiente el ciclo clásico de *read* + *write*.

tcp_cork — Es una propiedad que se puede asignar a un socket, de forma que bloquee temporalmente el procesado del buffer asociado a él - a nivel de kernel. Esto permite que sea posible ejecutar varias operaciones consecutivas de *write*, asegurando que no se enviarán pequeñas e ineficientes tramas de red sin apenas información. [14]

epoll — La función *epoll* [15], como se ha comentado anteriormente en este mismo artículo, examina un conjunto de descriptores, terminando su ejecución cuando se produzca un cambio de estado solicitado o se rebase un tiempo máximo especi-

ficado en uno de los parámetros de la función. Inicialmente *epoll* fué implementada en la rama de desarrollo de Linux 2.5 como un device y recibió el nombre de /dev/epoll. Algún tiempo despues, se eliminó el device y se incluyó la nueva llamada. Su principal bondad respecto a otras funciones similares - como *poll* o *select* - estriba en que su complejidad: mientras que sus antecesoras tienen una complejidad de O(n), *epoll* realiza el mismo trabajo con una complejidad O(1). La disminución de complejidad tiene como consecuencia un aumento en la capacidad de escalado.

Por otro lado existen una serie métodos de implementación que contribuyen a eficiencia del servidor web. En el caso de Cherokee se ha realizado especial enfasis en los siguientes puntos:

- Es posible reusar la memoria que se ha reservado. Hay muchas ocasiones en las que no es necesario liberar un fragmento de memoria que se va a reservar de nuevo poco después; de esta forma se ahorra una llamada a *free*, una a *malloc* y posiblemente el trabajo de inicializar la memoria. En la implementación de Cherokee, existen estructuras que se reusan en lugar de seguir una política de creación y destrucción: el ejemplo más claro de esta forma de trabajo se puede encontrar en la clase Connection. Por cada una de las conexiones que recibe el servidor se ha de crear un nuevo objecto de la clase Connection, pero cuando finaliza en lugar de destruirse, se limpia y se guarda para ser reusada en la siguiente petición.

- Es aconsejable intentar acortar los caminos críticos [16] y los más transitados al responder a una petición. La implementación de Cherokee intenta detectar primero los casos más frecuentes, de forma que el overhead por el procesamiento de una conexión estándar no aumente por la comprobación de una serie de parámetros que no le pertenecen.

- Existen trabajos que demuestran un mayor rendimiento de un servidor web si se ordenan las conexiones que se atienden [17]. Para aumentar el rendimiento global del servidor hay que asignar una mayor prioridad en la ordenación a las conexiones que se servirán más rapidamente y dejar con una menor prioridad las más largas. Este método, mejora la media del servidor, pero perjudica ligeramente a las conexiones que más tardan en completarse. En Cherokee, se ha experimentado con esta optimización, pero por el momento no se ha incluido ya que las mejoras de rendimiento parecen no ser perceptibles.

V. Pruebas de rendiemiento

Se han realizado una serie de pruebas de rendimiento entre diferentes servidores web Libres con el objetivo de evaluar la velocidad y eficiencia de cada uno de ellos. Han sido evaluados los siguientes servidores: Apache 2.0.48, Boa 0.94.13, Cherokee 0.4.8 y thttpd 2.23beta1.

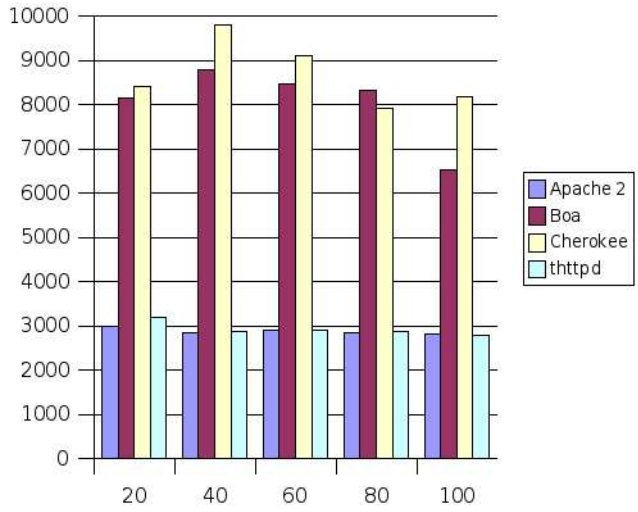

Fig. 2. Peticiones por seg. / conexiones concurrentes, sin keep-alive

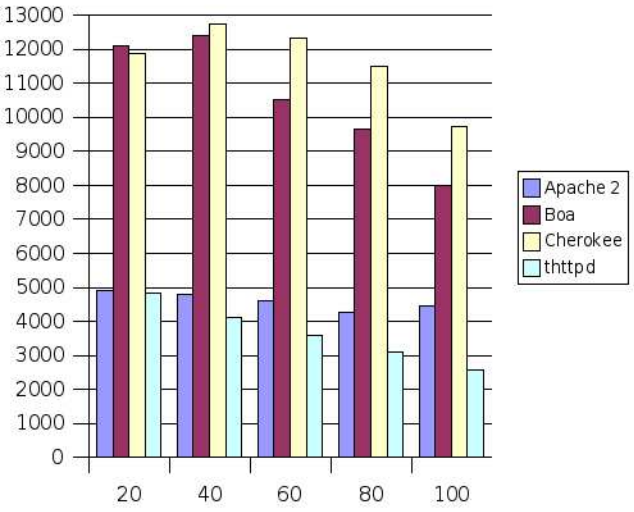

Fig. 3. peticiones por seg. / conexiones concurrente, con keep-alive [18]

VI. Trabajo actual y futuro

Actualmente se está trabajando en los siguientes aspectos del servidor:

- Reordenamiento de las conexiones para aumentar la eficiencia global.
- Algoritmo de detección de peticiones masivas. En caso de que Cherokee detecte un número de peticiones especialmente grande, es posible que un hilo debiera aceptar más de una conexión al adquirir el control sobre el socket del. Esta optimización también está directamente ligada con la eficiencia.
- Se está trabajando en el desarrollo de un módulo - handler - que implemente la ejecución eficiente de código Python. Con esta extensión se persigue la implementación de servicios de Web Services mediante la utilización de libcherokee y Python desde cualquier aplicación.

REFERENCES

[1] Balachander Krishnamurthy and Craig E. Wills, "Analyzing factors that influence end-to-end Web performance," *Computer Networks (Amsterdam, Netherlands: 1999)*, vol. 33, no. 1–6, pp. 17–32, 2000.

[2] Thomas T. Kwan, Robert McCrath, and Daniel A. Reed, "NCSA's world wide web server: Design and performance," *IEEE Computer*, vol. 28, no. 11, pp. 68–74, 1995.

[3] Richard Gregory and Ladan Tahvildari, "Architectural evolution: A case study of apache," .

[4] Adrian Cockcroft, "Performance: Static or dynamic linking," in *Sun Microsystems: IT World*, 2001.

[5] Jonathan Lemon, "Kqueue: A generic and scalable event notification facility," pp. 141–154.

[6] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel, "Flash: An efficient and portable Web server," in *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.

[7] Chuck Lever, Marius Eriksen, and Stephen Molloy, "An analysis of the tux web server," .

[8] Thiemo Voigt and Per Gunningberg, "Kernel-based control of persistent web server connections," .

[9] Cherokee, "Cherokee homepage, http://www.0x50.org," .

[10] GNU, "GNU homepage, http://www.gnu.org," since 1984.

[11] Felix von Leitner, "Scalable network programming," in *Linux Kongress*, 2003.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee, "RFC 2068: Hypertext Transfer Protocol — HTTP/1.1," Jan. 1997, Status: PROPOSED STANDARD.

[13] Rajeev Thakur and William Gropp, "Parallel i/o," .

[14] Kalpana S. Banerjee, "Tcp servers: A tcp/ip offloading architecture for internet servers, using memory-mapped communication," .

[15] Davide Libenzi, "Epoll homepage, http://www.xmailserver.org/linux-patches/nio-improve.html," 2001.

[16] Paul Barford and Mark Crovella, "Critical path analysis of TCP transactions," in *SIGCOMM*, 2000, pp. 127–138.

[17] Mark Crovella, Robert Frangioso, and Mor Harchol-Balter, "Connection scheduling in web servers," in *USENIX Symposium on Internet Technologies and Systems*, 1999.

[18] J. C. Mogul, "The Case for Persisten-Connection HTTP," in *Digital WRL Research Report 95/9*, 1995.