Trabajo Fin de Grado

Creación de núcleos de servidor web siguiendo diferentes modelos de concurrencia y comparación de rendimiento entre ellos

Autor: David Márquez Hernández

Profesor Colaborador: Jesús Arribi Vilela

Grado de Ingeniería Informática. Enero 2017

Índice General

Índice de tablas

Índice de figuras

Resumen / Abstract

Pese a la infinidad de protocolos existentes, uno de los más populares en Internet es el protocolo HTTP. La mayoría de servicios más notorios (por ejemplo, las principales redes sociales) se basan en él, de forma que exponen APIs HTTP para desarrolladores.

El punto central en toda interacción mediante protocolo HTTP es el servidor web, se trata del software que recibe las peticiones, las procesa y envía las respuestas pertinentes. Además, debido a los grandes volúmenes de usuarios que debe soportar un servidor WEB, es necesario que sea eficiente (para obtener el máximo aprovechamiento del hardware posible), con capacidad de atender un elevado número de peticiones simultáneas y escalable (debe ser capaz de crecer y decrecer de acuerdo con la necesidad de cada momento).

Este trabajo aborda la creación de tres núcleos de servidor web, cada uno implementado con un modelo de concurrencia diferente: procesos, *threads* o hilos y eventos, con el objetivo de evaluar los pros y contras de cada modelo y poder realizar una comparación de rendimiento entre ellos. Estos núcleos estarán implementados en C y desarrollados en Linux, no obstante, estarán escritos de forma que sería sencillo portarlos para que funcionen en otros sistemas Unix e incluso en sistemas no-Unix.

Even though there are a lot of protocols in use today, one of the most popular protocol on the Internet is HTTP. Most prevalent services (e.g. the main social networks) are based on HTTP and they expose HTTP APIs to developers.

The central node of every HTTP interaction is the web server, it is the software that receives the requests, processes them and sends the adequate responses. Additionally, due to the sheer number of users that a web service must support, there is a necessity for the web server to be efficient (so it gets the best possible use of hardware), with the capacity to attend a huge number of simultaneous requests and scalable (it must be able to grow and decrease according to each moment's requirements).

This paper centers around the creation of three web server cores, each one implemented based on a different concurrency model: processes, threads and events, with the objective of evaluating the pros and cons of each model and being able to do a performance comparison between them. These cores will be implemented in C and developed in Linux, nevertheless, they will be written in a way that would make it simple to port them to other Unix systems and even non-Unix systems.

Palabras clave

Servidor web, concurrencia, procesos, threads, bucle de eventos.

Web server, concurrency, processes, threads, event loop.

Punto de partida

Los servidores actuales son equipos que cuentan con un gran número de *cores* o unidades de procesamiento, cada core es capaz de ejecutar una tarea simultáneamente a los otros cores, sin tener en cuenta otras limitaciones del sistema como controladores de memoria, buses I/O, etc. No obstante, explotar esta característica desde el punto de vista del software no es una tarea sencilla. Los sistemas operativos y algunas librerías de desarrollo proporcionan diferentes modelos de abstracción para poder emplear estas unidades de procesamiento de la manera más eficiente posible. Cada uno de estos modelos presenta ciertas ventajas e inconvenientes que los pueden hacer más idóneos para algunos escenarios de uso.

Objetivo

A partir de los tres modelos de abstracción planteados: procesos, hilos o threads y eventos, se desarrollará un servidor web con cada uno de ellos para, a continuación, realizar una serie de pruebas de rendimiento que permita determinar la eficiencia y los pros y contras de cada implementación.

Productos obtenidos

La elaboración de este trabajo se acompaña de los siguientes adjuntos:

- Software: Código fuente de 3 núcleos de servidor web.
  - Procesos.
  - Hilos o *threads*.
  - Eventos.
- Resultados de las pruebas de rendimiento.

Enfoque y metodología

A lo largo de esta memoria, se introducirán una serie de conceptos y nociones teóricas necesarias para comprender el ámbito de la problemática planteada, así como las posibles soluciones. Cada apartado, comenzará con un prólogo sobre la materia en cuestión, para pasar a continuación a la aplicación concreta necesaria para la resolución del problema planteado. A pesar de comenzar en un plano teórico, todos los planteamientos concluyen en la implementación práctica del código de aplicación de los servidores web creados en el transcurso de este TFG.

Planificación

Tarea 1: Documentación

Objetivo: Recopilación de información y estudio de documentación.
Plazo: Del 1 de octubre al 15 de octubre.
Subtareas:

- Modelos de concurrencia:
  - Procesos.
  - *Threads*.
  - Bucle de eventos.
- Protocolo HTTP
  - RFC 1945 (HTTP/1.0) , RFC 2616 (HTTP/1.1) , RFC 7540 (HTTP/2).
- Diseño de software
  - Técnicas de diseño de software no OOP.

Tarea 2: Diseño y requisitos

Objetivo: Diseño de los diferentes modelos de núcleo de servidor web.
Plazo: Del 15 de octubre al 1 de noviembre.
Subtareas:

- Especificación de requisitos a cumplir por el servidor web (versión de HTTP, códigos de respuesta soportados, etc.).
- Diseño de varios núcleos de servidor web basados en diferentes modelos de concurrencia.

Tarea 3: Implementación

Objetivo: Implementación de los diferentes modelos de núcleo de servidor web.
Plazo: Del 1 de noviembre al 15 de diciembre.
Subtareas:

- Implementación en C de servidor web basado en procesos.
- Implementación en C de servidor web basado en *threads*.
- Implementación en C de servidor web basado en eventos.
-

Tarea 4: Pruebas

Objetivo: Pruebas de funcionamiento y comparativa de rendimiento (*Benchmarking*):
Plazo: Del 15 de diciembre al 25 de diciembre.
Subtareas:

- Comprobación de funcionamiento y cumplimiento de requisitos.
- Elección de software de medición de rendimiento: elaboración propia o un paquete existente.
- Ejecución de medidas de rendimiento de los diferentes servidores web.

Tarea 5: Documentación de instalación y memoria

Objetivo: Elaboración de la documentación de instalación y la memoria.
Plazo: Del 25 de diciembre al 8 de enero.
Subtareas:

- Documentar el proceso de compilación / instalación del servidor web.
- Elaborar el documento de memoria del TFG.

Tarea 6: Presentación

Objetivo: Elaboración de la presentación del TFG
Plazo: Del 8 al 15 de enero.
Subtareas:

- Creación de la presentación del TFG

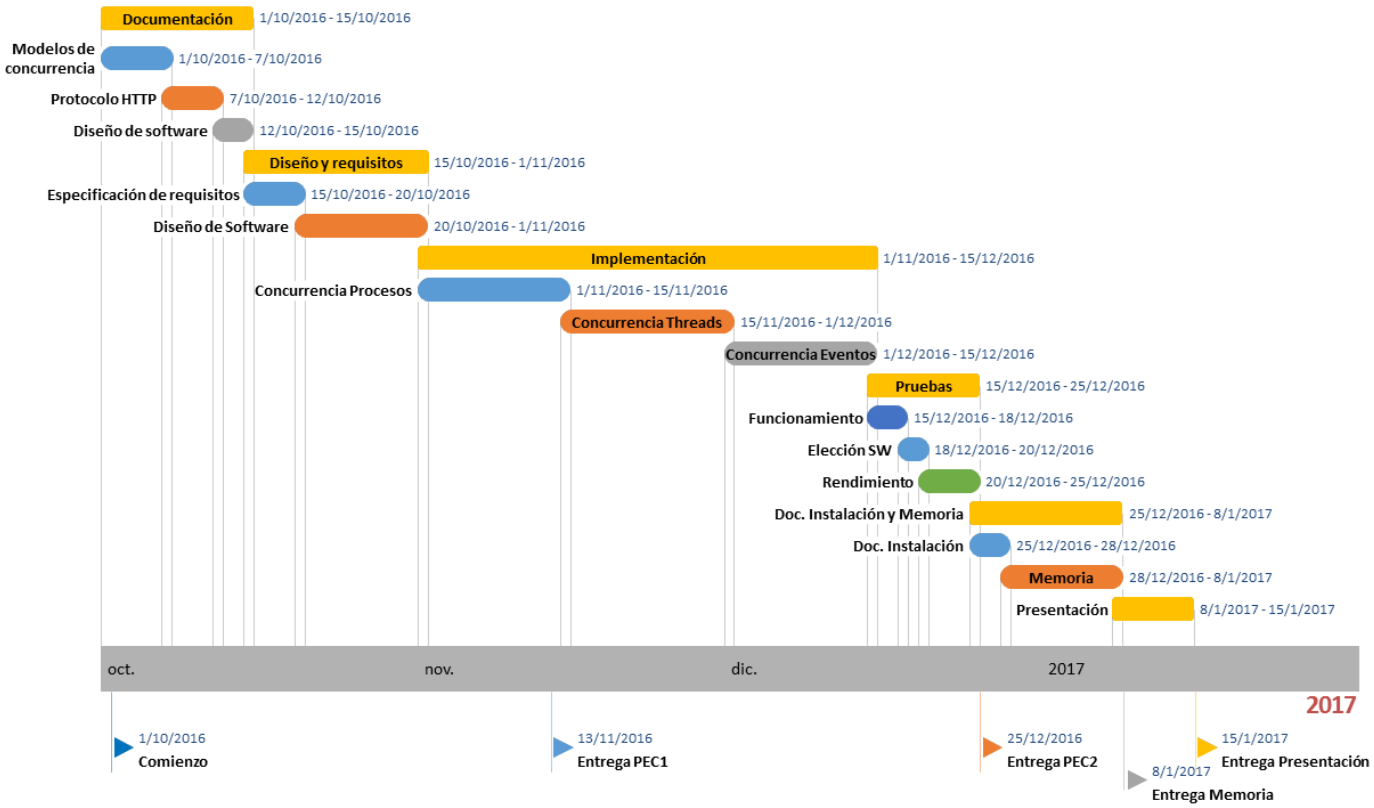

Fig. 1. Plan de trabajo del TFG.

Modelos de concurrencia

Concurrencia y paralelismo

Habitualmente se habla de concurrencia y paralelismo de forma intercambiable, cuando son términos que, si bien expresan ideas relacionadas, tienen significados diferentes [1]. El paralelismo es la ejecución simultánea de varias operaciones (en paralelo), esto es posible siempre que dispongamos de un sistema capaz de realizar esas operaciones a la vez, por tanto, debe tratarse de un sistema con varias unidades de procesamiento (según el sistema pueden ser *cores* en un mismo procesador, varios procesadores, un clúster de máquinas interconectadas, etc.)

En contraste, un sistema concurrente es aquel en el que las diferentes tareas que ejecuta, no necesitan esperar a que termine otra para continuar su ejecución, es decir, que pueden parar en un momento determinado, pasar a ejecutar otra tarea y volver a la inicial. Esto es lo que ocurre en un sistema con un único procesador *single-core* que emplea un sistema operativo multitarea.

La coordinación de ejecución de múltiples tareas que no tienen dependencia entre ellas sólo tiene que ocuparse de dividir el tiempo de ejecución de los diferentes recursos del sistema entre las tareas. Sin embargo, cuando esas mismas tareas sí tienen relación, en particular, si emplean recursos comunes o dependen de los resultados de otra tarea, aparecen problemas como los *deadlocks* o las *race condition* [2][3].

Una situación de *deadlock* ocurre cuando una tarea A está esperando a que una tarea B termine una acción y, al mismo tiempo, la tarea B está esperando a que la tarea A termine otra acción, de esta manera, se crea un problema en el que ninguna de las dos puede avanzar. Por ejemplo, si la tarea A tiene bloqueado un fichero F1 para escribir y está esperando a que antes se libere otro fichero F2 y, además, una tarea B tiene bloqueado el fichero F2 y está esperando poder acceder al fichero F1. Como vemos, ninguna de las dos tareas proseguirá su ejecución.

Una *race condition* aparece cuando los eventos de varios procesos o *threads*, que se afectan entre sí, no ocurren como deberían. Por ejemplo, si tenemos dos procesos que acceden a una misma variable, ambos procesos leen el mismo valor de la variable, lo incrementan en un 1 y luego la intentan escribir. La intención original es que cada proceso incremente la variable en 1 respecto al valor anterior. Sin embargo, lo que ocurre es que ambos han leído el mismo valor, han sumado uno y han intentado escribir ese valor+1, pero uno de los aumentos se perderá, ya que solo se registrará la última escritura. Para evitar *race conditions* se definen secciones críticas cuyo acceso debe ser de exclusión mutua (*mutex*): sólo pueden ser accedidas por un proceso a la vez, así pues, los otros procesos se quedan esperando a que el primero termine antes de acceder. En nuestro ejemplo, la variable quedaría bloqueada por un *mutex* hasta completar totalmente la primera suma y luego se ejecutaría la segunda, de forma que el resultado final sería valor+2. También se pueden emplear otras primitivas de control de acceso como semáforos, barreras o spinlocks.

Existe un lenguaje formal denominado *communicating sequential processes* (CSP) [4], desarrollado por A. Hoare, que permite realizar comprobaciones sobre código paralelo para determinar si existen posibles *deadlocks.*

Se podría pensar que dado un problema que se ejecuta en un tiempo finito "t" en un elemento de proceso, se ejecutará en t/n cuando tenemos "n" elementos de proceso. No obstante, esto no es posible, ya que un problema rara vez es paralelizable al 100%. Normalmente un programa tiene una parte que requiere ser ejecutada en serie y otra que es paralelizable, lo que limita el potencial incremento de rendimiento de la paralelización del programa. Por ejemplo, si tenemos un programa que tarda 2 horas en ejecutarse, de las cuales 1 hora se ejecuta en serie (no paralelizable) y la otra hora sí es paralelizable, por mucho que añadamos procesadores al sistema, la hora de ejecución en serie no disminuirá. Por tanto, existe un límite marcado por la parte no paralelizable que no es posible mejorar. Esta limitación viene formulada por la ley de Amdahl[5] de la siguiente forma:

$$Slatency(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

Donde:

- S es el factor de mejora total de rendimiento que obtenemos.
- p es el porcentaje del proceso que mejora su velocidad de ejecución cuando aumentan los recursos del sistema, es decir, la parte paralelizable.
- s es el factor de incremento de recursos (p.ej. unidades de procesamiento).

Veamos en una tabla cómo evoluciona S en función de p y s:

Mejora 30% paralelizable (S para p=30%)	Mejora 50% paralelizable (S para p=50%)	Mejora 80% paralelizable (S para p=80%)	Recursos Disponibles (s)	Porcentaje paralelizable (p)	Porcentaje paralelizable (p)	Porcentaje paralelizable (p)
1	**1**	**1**	1	0,3	0,5	0,8
1,18	**1,33**	**1,67**	2	0,3	0,5	0,8
1,25	**1,5**	**2,14**	3	0,3	0,5	0,8
1,29	**1,6**	**2,5**	4	0,3	0,5	0,8
1,32	**1,67**	**2,78**	5	0,3	0,5	0,8
1,33	**1,71**	**3**	6	0,3	0,5	0,8
1,35	**1,75**	**3,18**	7	0,3	0,5	0,8
1,36	**1,78**	**3,33**	8	0,3	0,5	0,8
1,36	**1,8**	**3,46**	9	0,3	0,5	0,8
1,37	**1,82**	**3,57**	10	0,3	0,5	0,8
1,38	**1,83**	**3,67**	11	0,3	0,5	0,8
1,38	**1,85**	**3,75**	12	0,3	0,5	0,8
1,38	**1,86**	**3,82**	13	0,3	0,5	0,8
1,39	**1,87**	**3,89**	14	0,3	0,5	0,8
1,39	**1,88**	**3,95**	15	0,3	0,5	0,8
1,39	**1,88**	**4**	16	0,3	0,5	0,8
1,39	**1,89**	**4,05**	17	0,3	0,5	0,8
1,4	**1,89**	**4,09**	18	0,3	0,5	0,8
1,4	**1,9**	**4,13**	19	0,3	0,5	0,8
1,4	**1,9**	**4,17**	20	0,3	0,5	0,8
1,4	**1,91**	**4,2**	21	0,3	0,5	0,8
1,4	**1,91**	**4,23**	22	0,3	0,5	0,8
1,4	**1,92**	**4,26**	23	0,3	0,5	0,8
1,4	**1,92**	**4,29**	24	0,3	0,5	0,8
1,4	**1,92**	**4,31**	25	0,3	0,5	0,8
1,41	**1,93**	**4,33**	26	0,3	0,5	0,8
1,41	**1,93**	**4,35**	27	0,3	0,5	0,8
1,41	**1,93**	**4,38**	28	0,3	0,5	0,8
1,41	**1,93**	**4,39**	29	0,3	0,5	0,8
1,41	**1,94**	**4,41**	30	0,3	0,5	0,8
1,41	**1,94**	**4,43**	31	0,3	0,5	0,8
1,41	**1,94**	**4,44**	32	0,3	0,5	0,8
1,41	**1,94**	**4,46**	33	0,3	0,5	0,8

Tabla 1. Mejora de rendimiento con el incremento de recursos de procesos según la ley de Amdhal.

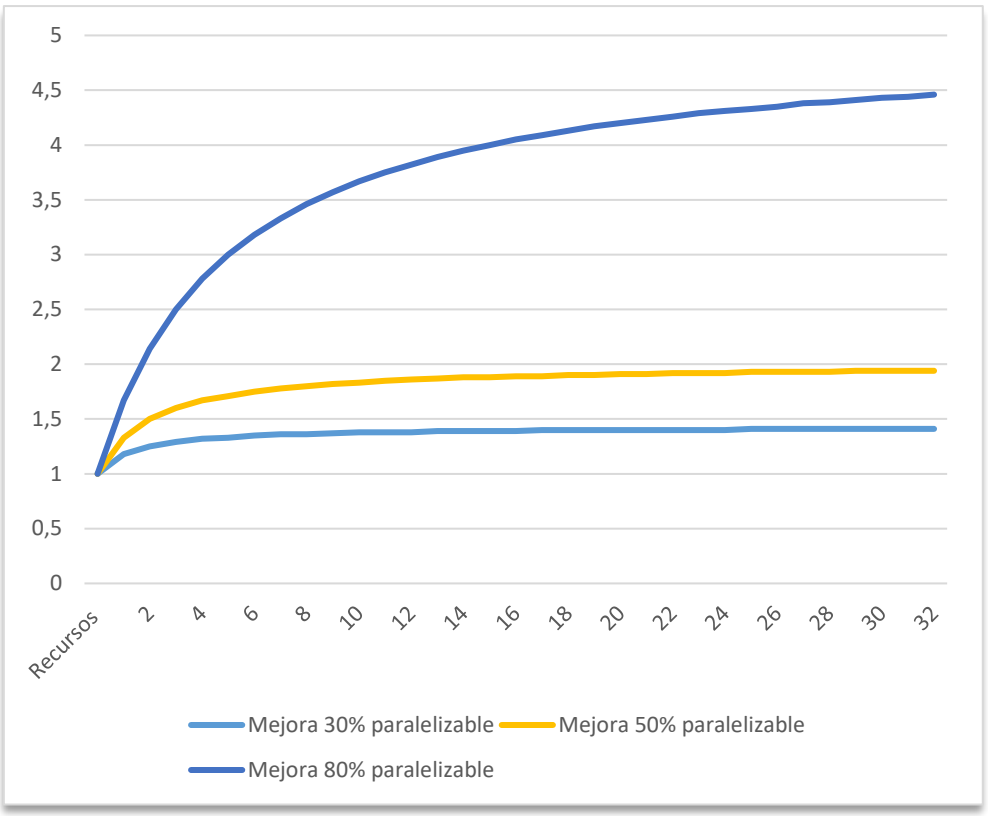

Fig. 2. Representación de la mejora de rendimiento en función de los recursos según la ley de Amdhal.

Como vemos, a partir de determinado número de recursos, la mejora se empieza a acercar asintóticamente a un valor máximo, es decir, existe un límite determinado por el porcentaje del proceso que sea paralelizable a partir del cual la velocidad de ejecución no mejora con más recursos. La razón de este comportamiento es que la parte en serie no se ejecuta más rápido, aunque tenga más recursos disponibles. Cuanto más paralelizable es el proceso (por tanto, menor la parte en serie), más se beneficiará del incremento en el número de recursos.

Sin embargo, la ley de Amdhal sólo considera problemas cuya carga de trabajo se mantiene constante, aunque aumenten los recursos del sistema para realizar ese trabajo. Cabe pensar que a medida que el sistema aumente sus recursos disponibles, querremos también aumentar la carga de trabajo que tiene que afrontar. Para poder modelar el comportamiento de sistemas cuya carga de trabajo aumenta a medida que crecen los recursos disponibles, Gustafson formuló [5], a partir de la ley de Amdhal, la siguiente variante:

$$Slatency(s) = 1 - p + sp$$

Donde:

- S es el factor de mejora total que obtenemos.
- p es el porcentaje del proceso que mejora su velocidad de ejecución cuando aumentan los recursos del sistema, es decir, la parte paralelizable.
- s es el factor de incremento de recursos (p.ej. unidades de procesamiento)

En este caso, podemos ver, al repetir nuestra tabla de valores, como si la carga de trabajo aumenta cuando aumentan los recursos, el aprovechamiento de los recursos crece indefinidamente.

Mejora 30% paralelizable (S para p=30%)	Mejora 50% paralelizable (S para p=50%)	Mejora 80% paralelizable (S para p=80%)	Recursos Disponibles (s)	Porcentaje paralelizable (p)	Porcentaje paralelizable (p)	Porcentaje paralelizable (p)
1	**1**	**1**	1	0,3	0,5	0,8
1,3	**1,5**	**1,8**	2	0,3	0,5	0,8
1,6	**2**	**2,6**	3	0,3	0,5	0,8
1,9	**2,5**	**3,4**	4	0,3	0,5	0,8
2,2	**3**	**4,2**	5	0,3	0,5	0,8
2,5	**3,5**	**5**	6	0,3	0,5	0,8
2,8	**4**	**5,8**	7	0,3	0,5	0,8
3,1	**4,5**	**6,6**	8	0,3	0,5	0,8
3,4	**5**	**7,4**	9	0,3	0,5	0,8
3,7	**5,5**	**8,2**	10	0,3	0,5	0,8
4	**6**	**9**	11	0,3	0,5	0,8
4,3	**6,5**	**9,8**	12	0,3	0,5	0,8
4,6	**7**	**10,6**	13	0,3	0,5	0,8
4,9	**7,5**	**11,4**	14	0,3	0,5	0,8
5,2	**8**	**12,2**	15	0,3	0,5	0,8
5,5	**8,5**	**13**	16	0,3	0,5	0,8
5,8	**9**	**13,8**	17	0,3	0,5	0,8
6,1	**9,5**	**14,6**	18	0,3	0,5	0,8
6,4	**10**	**15,4**	19	0,3	0,5	0,8
6,7	**10,5**	**16,2**	20	0,3	0,5	0,8
7	**11**	**17**	21	0,3	0,5	0,8
7,3	**11,5**	**17,8**	22	0,3	0,5	0,8
7,6	**12**	**18,6**	23	0,3	0,5	0,8
7,9	**12,5**	**19,4**	24	0,3	0,5	0,8
8,2	**13**	**20,2**	25	0,3	0,5	0,8
8,5	**13,5**	**21**	26	0,3	0,5	0,8
8,8	**14**	**21,8**	27	0,3	0,5	0,8
9,1	**14,5**	**22,6**	28	0,3	0,5	0,8
9,4	**15**	**23,4**	29	0,3	0,5	0,8
9,7	**15,5**	**24,2**	30	0,3	0,5	0,8
10	**16**	**25**	31	0,3	0,5	0,8
10,3	**16,5**	**25,8**	32	0,3	0,5	0,8
10,6	**17**	**26,6**	33	0,3	0,5	0,8

Tabla 2. Mejora de rendimiento con el incremento de recursos de procesos según la ley de Gustafson.

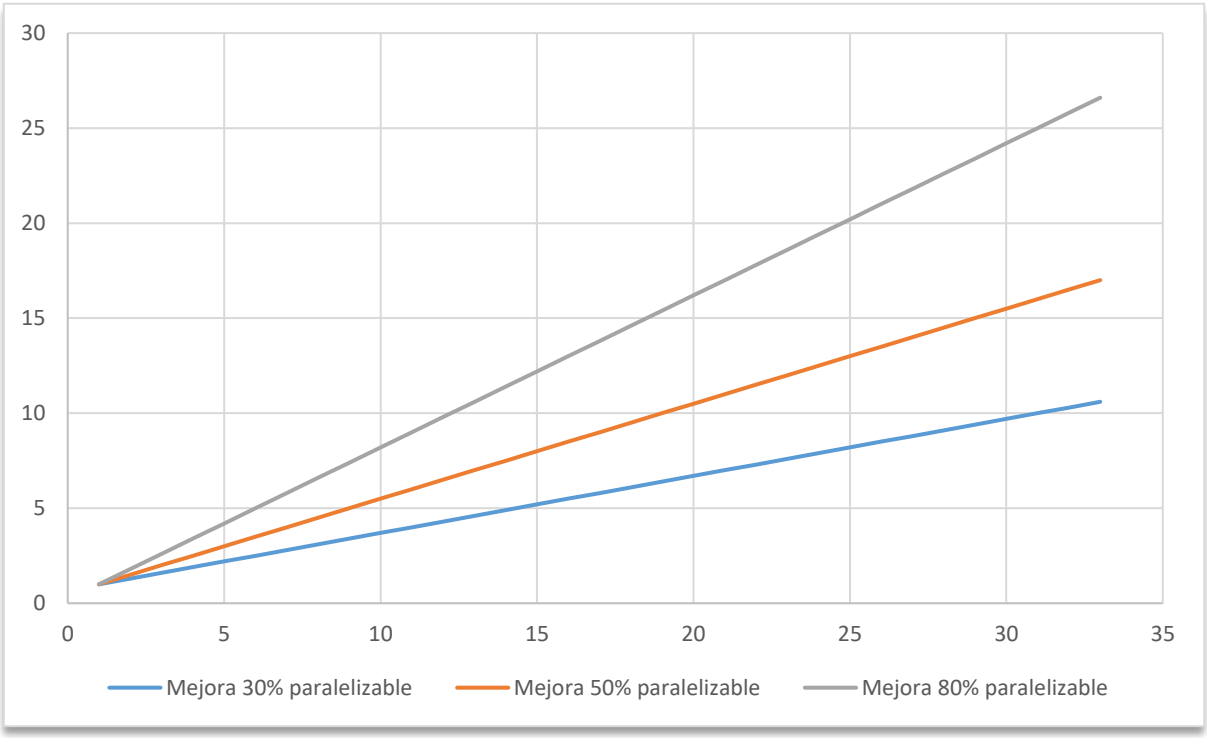

Fig. 3. Representación de la mejora de rendimiento en función de los recursos según la ley de Gustafson.

Es decir, si bien la parte que se ejecuta en serie sigue sin poderse acelerar, cada vez se hace más pequeña respecto al total del trabajo, ya que la carga de trabajo se incrementa a medida que aumentan los recursos del sistema. Así pues, nunca se alcanza un beneficio máximo equivalente al aumento de recursos (si se multiplican los recursos por 100, no aumenta la velocidad de ejecución en un factor cien), pero sí que se incrementa con el aumento de recursos.

Concurrencia en servicios de red en una arquitectura cliente-servidor

Es común que a un servidor web se conecten multitud de clientes, por tanto, debe ser capaz de atender a un gran número de peticiones de forma simultánea y, además, debe hacerlo en un tiempo de respuesta aceptable y con el mejor aprovechamiento posible de los recursos de la máquina.

Si aplicamos los modelos de Amdhal y Gustafson a un servidor web, vemos que la parte que se ejecuta en serie es pequeña; principalmente consta de la secuencia de inicio y de las partes del programa que se encargan de controlar las nuevas conexiones, la concurrencia y la compartición de recursos para evitar *deadlocks*/*race conditions*.

Debemos ser especialmente cuidadosos con el acceso a recursos compartidos, de forma que en primer lugar planteemos el código de forma que se minimicen esos accesos y, en segundo lugar, definamos las diferentes secciones críticas que deben ser accedidas siguiendo una política de exclusión mutua para evitar *race conditions* y la aparición de *deadlocks.*

Así pues, a lo largo de este trabajo exploraremos la relación entre recursos, modelos de concurrencia y capacidad de servicio para un servidor web, de forma que establezcamos la mejor combinación de esos factores de cara a obtener el máximo rendimiento posible.

Procesos y *threads*

Los dos sistemas más utilizados para conseguir concurrencia son los procesos y los *threads* o hilos. La mayor diferencia entre ellos es que cada proceso se ejecuta en un espacio de memoria independiente, mientras que puede haber varios *threads* en un proceso, que comparten espacio de memoria [6]. Además, los procesos tienen una relación jerárquica, donde un proceso padre puede generar varios procesos hijos que a su vez pueden tener hijos. En cambio, los *threads* dentro de un proceso se ejecutan todos al mismo nivel.

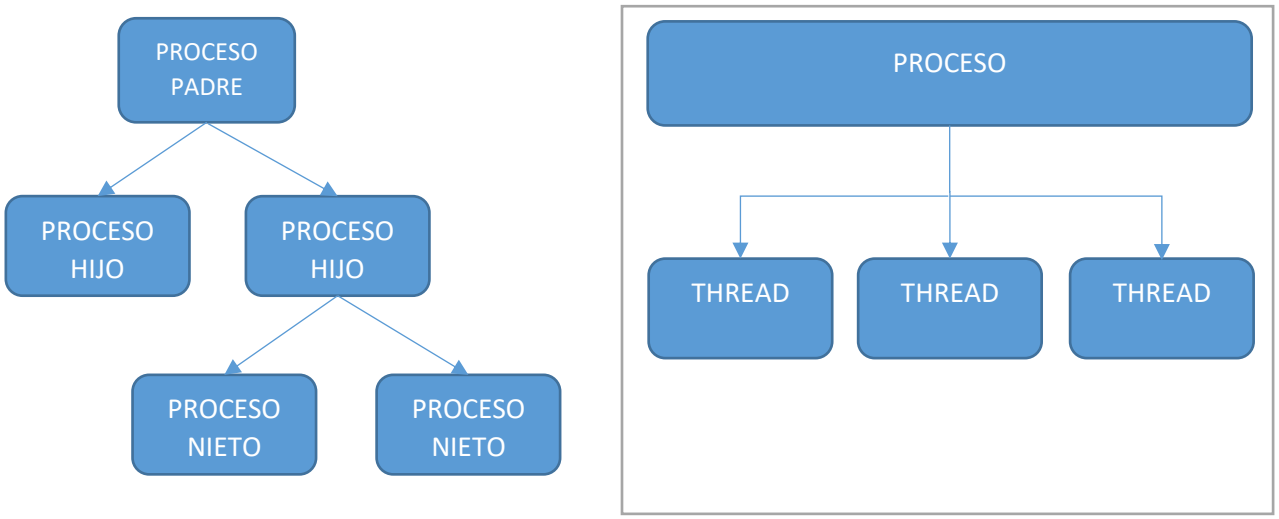

Fig. 4. Jerarquía de procesos y *threads*.

Cuando un proceso genera un hijo, se realiza una copia del segmento de datos del padre. Entonces, si el padre realiza un cambio en su espacio de memoria, el hijo no puede verlo sin ayuda de algún mecanismo adicional de comunicación padre/hijo. En contraste, todos los *threads* de un proceso tienen visibilidad del espacio de memoria común del proceso.

Podemos entender un proceso como una tarea que se ejecuta con un único *thread*. Así pues, tenemos [7]:

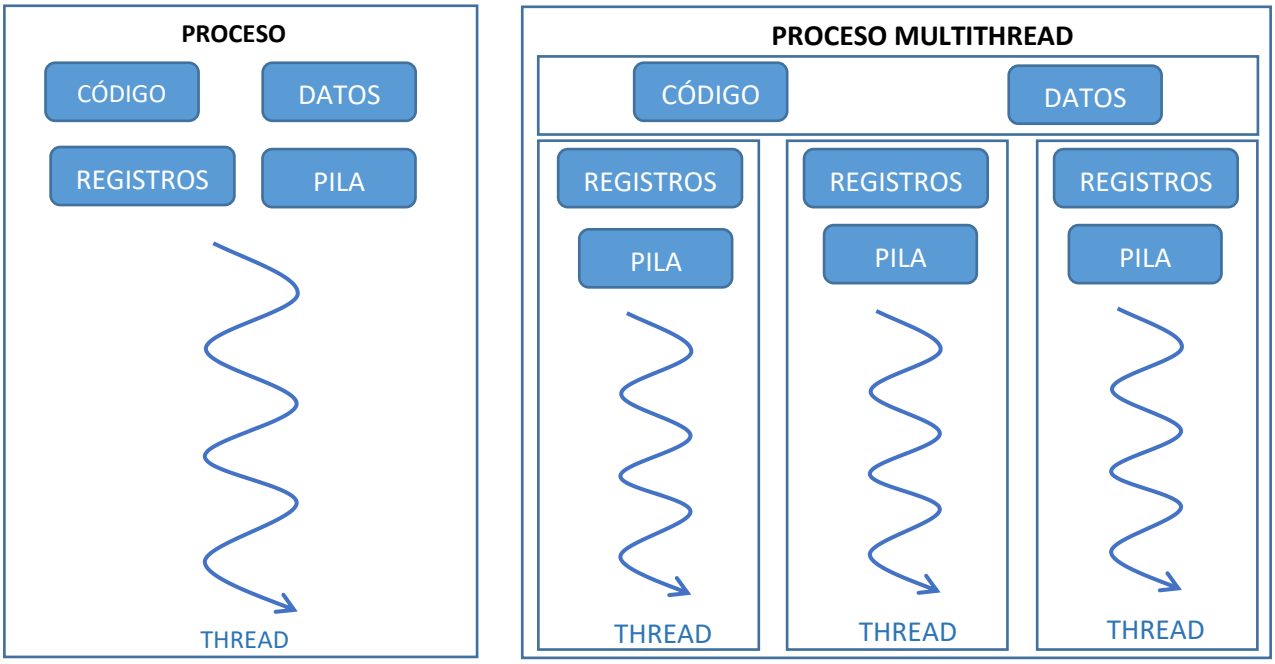

Fig. 5. Representación de procesos y threads, basado en Computer Science Lectures. Curso CIT 595. University of Pennsylvania [7]

Cuando estamos ejecutando un gran número de *threads* o de procesos, mayor que el número de unidades de proceso disponibles, el flujo de ejecución va cambiando entre los *threads*/procesos. Este cambio de un *thread* a otro o de un proceso a otro se denomina cambio de contexto (*context switch*) y tiene un coste temporal importante, ya que es necesario guardar, entre otros, el estado de todos los registros y del puntero a la pila.

Este cambio de contexto es diferente en el caso de los procesos y de los *threads*. Como hemos visto, los procesos tienen cada uno una copia diferente del segmento de datos, que debe también ser considerado al hacer el cambio de contexto. Por el contrario, los *threads* sólo deben ocuparse de cambiar las partes relacionadas con los registros y la pila al hacer el cambio de contexto, ya que los datos son compartidos entre todos los *threads* de un proceso. Por tanto, un cambio de contexto entre *threads* será más rápido que un cambio de contexto entre procesos.

Eventos

La tercera forma de concurrencia que consideraremos en este trabajo son los sistemas basados en eventos [8]. En estos sistemas, se definen una serie de funciones (*callbacks*) que, a su vez, se asocian a eventos. De tal manera que cuando se produce uno de esos eventos, se llamará a la función asociada.

En un sistema basado en *threads*, cuando recibimos una petición, podemos generar un nuevo *thread* que la atienda. En cambio, en un sistema basado en eventos, cuando se recibe un nuevo evento, se coloca en una "cola". Existe, además, un bucle principal de eventos (*event loop)* que se encarga de gestionar todos los eventos que haya encolados en cada momento, de forma que todo se ejecuta en único *thread* y se evitan cambios de contexto. Cada tipo de evento que haya que procesar es gestionado por una rutina que gestionará esos eventos denominada *handler.* Es importante resaltar que una rutina tipo *handler* no debe bloquear nunca, ya que entonces el bucle principal de procesado se pararía. Para ello, una manera de lograrlo es que cuando hay que realizar una acción susceptible de bloquear (por ejemplo, un acceso a disco), el sistema lanza la operación y define una rutina *callback* que será llamada cuando el resultado está disponible,

de forma que mientras tanto el resto de eventos puede continuar su ejecución. En el caso que nos ocupa para un servidor web, veremos que en la implementación basada en eventos configuraremos los *sockets* en modo no bloqueante (*non-blocking*) para que no se detenga el bucle de eventos mientras se espera una conexión en el *socket*.

Notemos que, el modelo de eventos, al ejecutarse en un sistema con múltiples unidades de proceso, sólo empleará una de ellas, mientras que un sistema basado en *threads* o en procesos automáticamente va usando todos los procesadores disponibles según se van creando *threads*/procesos. Para solucionarlo, los sistemas basados en eventos, pueden cargar varios procesos de bucle de eventos (por ejemplo, tantos como procesadores haya disponibles). En tal caso, es necesario repartir las peticiones con un mecanismo externo (como puede ser con un balanceador TCP). Por ejemplo, si tenemos un sistema con 8 cores y estamos ejecutando Node.js (una plataforma para construir aplicaciones de red basada en eventos), que se ejecuta en un *thread* sencillo con un bucle de eventos, podemos arrancar 8 procesos de Node.js y repartir el tráfico entrante entre ellos con un balanceador TCP externo. Otra manera de hacerlo es mediante el módulo Clúster [9] que crea varios procesos Node.js internamente.

Procesos, *threads* y eventos: Ventajas y desventajas

Podemos ahora enumerar las ventajas e inconvenientes de cada modelo de concurrencia:

Procesos	
Ventajas	▪ Sencillez de implantación.
Desventajas	▪ Coste de creación de cada nuevo hijo. ▪ Dificultad de comunicación entre procesos al no compartir espacio de memoria.

Threads	
Ventajas	▪ Se crean más rápido que un proceso ya que no necesitan espacio de memoria independiente. ▪ Fácil comunicación entre *threads*, ya que comparten espacio de memoria.
Desventajas	▪ El cambio de ejecutar un *thread* a otro (*context switching*) aunque es más liviano que en el caso de procesos, sigue siendo costoso en tiempo. ▪ Necesario control especial para acceso a recursos compartidos (*mutex*) que evite *deadlocks* y *race conditions*.

Eventos	
Ventajas	▪ No hay que realizar un cambio de contexto (*context switch*) para pasar de una tarea a otra.
Desventajas	▪ Dificultad de programación: ○ No puede haber gestores de eventos que bloqueen. ○ Hay que programar mediante *callbacks*, menos intuitivo que la programación convencional. ▪ Para poder aprovechar sistemas multi-procesador, los bucles de eventos deben además recurrir a *threads*/procesos convencionales.

Tabla 3. Ventajas e inconvenientes de los distintos modelos de concurrencia.

Protocolo HTTP

HTTP (Hyper Text Transfer Protocol) es un protocolo que, según el modelo de capas OSI, pertenece a la capa de aplicación [10]. Habitualmente se emplea sobre TCP, sin embargo, puede funcionar sobre cualquier capa de transporte que garantice la entrega (es decir, no podría funcionar correctamente sobre UDP).

	Capa	Ejemplo en la pila TCP
1	Física	Definición hardware del enlace
2	Enlace de datos	PPP, SLIP
3	Red	IP, ICMP, OSPF
4	Transporte	TCP, UDP
5	Sesión	Sockets
6	Presentación	SSL/TLS
7	Aplicación	HTTP, SMTP, FTP, NFS

Tabla 4. Capas modelo OSI.

- HTTP es un protocolo sin estado (*stateless)*, cada petición es independiente y no conoce lo ocurrido en las peticiones precedentes. Sin embargo, mediante el uso de *cookies* es posible mantener una sesión a lo largo de varias peticiones HTTP. (Por ejemplo, cuando nos autenticamos en una web y durante toda la navegación seguimos autenticados).
- HTTP funciona según un modelo cliente servidor *request-response*: Un cliente realiza una petición (HTTP *request*) y un servidor contesta la respuesta (HTTP *response*).

Flujo de una conexión HTTP

El flujo habitual de una conexión HTTP es [11]:

1. El cliente abre una conexión TCP al servidor: Normalmente se conecta con el puerto 80 en el servidor para HTTP y con el 443 para HTTPS.
2. El cliente envía la petición HTTP.
3. El servidor analiza la petición y envía la respuesta HTTP.
4. El cliente cierra la conexión si ha terminado o la reutiliza si va a hacer más peticiones.

Hipertexto

HTTP fue concebido como un mecanismo para transportar hipertexto. Entendemos por hipertexto [12] un texto no lineal que contiene enlaces a otros textos. A medida que la técnica lo permitió, se extendió no sólo a texto sino también a imágenes, vídeos y otros elementos, en lo que se denomina hipermedia.

URIs

Para localizar un recurso de red, se emplea una URI (*Uniform Resource Identifier*) que indica la ubicación de ese recurso. El formato de una URI viene definido en el RFC 3986 [13]. Ese documento indica que las URLs son el subconjunto de las URIs tales que, además de identificar el recurso, también proporcionan una manera de localizar el recurso describiendo su método principal de acceso.

La sintaxis genérica de una URI tiene el formato:

esquema : jerarquía [“?” consulta] [“#” fragmento]

En el caso que nos ocupa para el protocolo HTTP son de la forma:

"http:" "//" servidor [":" puerto] [camino] [“?” consulta] [“#” fragmento]

De esta manera, con estas URI indicaremos al cliente http a qué servidor, qué recurso y qué parte del recurso queremos acceder, así como si también queremos indicar algún parámetro adicional.

HTTP/1.0

Si bien existe una versión anterior del protocolo HTTP (la 0.9), la primera versión estandarizada en un documento RFC (*Request For Comments*) es la 1.0, que viene especificada en el documento RFC 1945 [14] de la IETF (Internet Engineering Task Force): Hypertext Transfer Protocol - - HTTP/1.0.

En ese documento, se establece el funcionamiento básico del protocolo:

1. El cliente debe enviar una petición con un método, una URI y una versión del protocolo, seguido por un mensaje MIME con modificadores de la petición, información del cliente y contenido.
2. El servidor responde con una línea de estado, y un código de retorno que indica el éxito o fracaso de la petición, metadatos adicionales y contenido.

También se define la posible presencia de otros elementos intermedios entre el cliente y el servidor; como pueden ser servidores proxy, puertas de enlace o túneles. Además, toda conexión debe ser iniciada por el cliente antes de hacer una petición y cerrada por el servidor al enviar la respuesta. No obstante, la conexión puede acabar antes, bien por acción del usuario, por un *timeout* o por algún error, lo que debe ser gestionado adecuadamente por las dos partes.

La compatibilidad hacia atrás debe mantenerse, por tanto, un cliente HTTP/1.0 debe soportar respuestas de un servidor HTTP/0.9 y, de manera similar, un servidor HTTP/1.0 debe soportar peticiones de un cliente HTTP/0.9.

Asimismo, también se indican los diferentes formatos de fecha y juegos de caracteres que debe soportar y la posible compresión del contenido de un recurso mediante gzip. También se indica que los recursos van tipificados mediante MIME y que pueden codificarse varios recursos en una sola respuesta.

Se detalla el formato de las peticiones, incluyendo los 3 posibles métodos estándar: GET, HEAD y POST y el resto de cabeceras que puede incluir una petición. De la misma forma, se describe el formato que deben tener las respuestas.

El servidor, cuando contesta al cliente, debe incluir un código de respuesta, que puede ser:

Significado	Formato del código
Informativo	1xx
Éxito	2xx
Redirección	3xx
Error del cliente	4xx
Error del servidor	5xx

Tabla 5. Códigos de respuesta HTTP.

También se describe un mecanismo de autenticación para que el cliente pueda identificarse cuando solicita acceder a un recurso protegido, fechas de caducidad para validez del contenido, fechas de modificación para que el cliente conozca cuando se ha alterado el contenido, y otras cabeceras. Entre éstas últimas podemos destacar "User-Agent" que contiene información sobre el cliente que hace una petición, típicamente nombre y versión del navegador y a veces también el sistema operativo.

HTTP/1.1

Las principales novedades que introdujo la versión 1.1 del protocolo HTTP [15] [16] son:

- Soporte de *virtual hosts*: Alojamiento de múltiples servidores web (bajo diferentes dominios) en una misma IP.
- Indicación de versión de HTTP soportada por todos los saltos intermedios: HTTP/1.1 incorpora la cabecera "Via" en la que se indica la versión de HTTP soportada para todos los saltos intermedios entre el cliente y el servidor (por ejemplo, servidores proxy).
- Inclusión del método OPTIONS, con el que un cliente puede preguntar las capacidades de un servidor sin necesidad de hacer una petición real de un recurso.
- Mejoras en el cacheo: HTTP/1.1 incorpora un *entity tag* que permite comparar dos recursos entre sí para ver si han cambiado. Además, también se incorporan otras cabeceras de control de caché.
- Se permite la descarga parcial de recursos, especificando un rango de bytes a descargar, lo que hace posible reanudar descargas que fallaron o se pausaron sin necesidad de volver a comenzar desde el principio.
- Comprobación de que el servidor está listo para realizar una subida: En algunos casos, al transmitir un cuerpo de mensaje muy largo, el servidor podía haber denegado la autenticación y, aun así, el cliente enviaba el cuerpo del mensaje de todas formas ya que el mensaje de error sólo lo recibía al terminar su petición. En HTTP/1.1 se incluye un nuevo código de respuesta: 100-Continue que permite comprobar si el servidor acepta el envío del cuerpo del mensaje tras haber enviado las cabeceras.
- Mejoras en la compresión: Además de la compresión gzip que incorporaba HTTP/1.0, en la versión 1.1 se diferencia entre codificación end-to-end y codificación entre cada salto, de forma que se pueden emplear diferentes codificaciones (entre ellas, compresiones) entre los distintos saltos que haya entre el cliente y el servidor.
- Conexiones Persistentes: Las conexiones permanecen abiertas después de hacer una petición y recibir una respuesta, de manera que se puede utilizar la misma conexión para realizar otra petición posterior.
- *Pipelining*: Para mejorar el rendimiento, no es necesario que un cliente espere una respuesta antes de seguir mandando peticiones, empleando pipelining puede lanzar todas las peticiones de recursos que necesite por una conexión sin esperar a que se conteste la primera.
- *Chunked transfer*: La cabecera debe indicar cuanto ocupa el cuerpo del mensaje, no obstante, esto es un problema si el cuerpo del mensaje es generado dinámicamente. En HTTP/1.0 el servidor cerraba la conexión cuando terminaba, sin embargo, con la aparición de las conexiones persistentes en HTTP/1.1, hacía falta un nuevo mecanismo para esos cuerpos de mensaje cuya longitud es desconocida al empezar a transmitirlos. C*hunked transfer* permite enviar los cuerpos de mensaje por partes (*chunks)* de forma que cada trozo se envía indicando su longitud y el cuerpo de mensaje termina con un trozo de longitud cero.
- Mejoras en los mecanismos de autenticación, incluyendo autenticación específica para proxies.
- Recomendaciones de privacidad en URIs: HTTP/1.1 recomienda que los formularios sean enviados mediante POST y no mediante GET, ya que al hacerlo mediante GET, los contenidos del formulario aparecen en la propia URL y pueden quedar registrados en los logs de un proxy o del servidor.
- Cabecera Content-MD5: Permite detectar cambios accidentales en el mensaje, aunque no modificaciones intencionadas, puesto que bastaría con recalcular el hash MD5 de nuevo por parte de la entidad que modifica el mensaje.
- Mejoras en la negociación de contenido para conseguir la mejor representación posible de un recurso (por ejemplo, el idioma entre la lista de idiomas soportados por un cliente).

HTTP/2.0

Con el objetivo de obtener mejoras de velocidad de aproximadamente un 50% respecto a HTTP 1.1 (cifra que finalmente se consiguió casi alcanzar), Google desarrolló el protocolo SPDY en 2012 [17]. Los puntos principales de SPDY son: Mejora del uso de una única conexión TCP, compresión de cabeceras, *server push*, etc. Finalmente, estas aportaciones de SPDY formaron parte del estándar HTTP/2.0 [18], que se describe en el RFC 7540 [19]. En detalle, las principales diferencias entre HTTP/1.1 y la versión 2.0 son [20][21]:

- De cara a eliminar posibles errores (mayúsculas, espacios) HTTP/2.0 es binario en lugar de texto, lo que además hace que sea más compacto: la misma información ocupará menos en una transmisión HTTP/2.0 respecto a las versiones anteriores.
- Si bien HTTP/1.1, mediante *pipelining*, soporta poder enviar muchas peticiones sin esperar respuestas, puede ocurrir que una respuesta muy voluminosa retrase el resto de respuestas. En HTTP/2.0, se ha implementado multiplexación, de forma que es posible intercalar una respuesta corta en medio de una respuesta más grande. Además, HTTP/2.0 permite que el cliente indique al servidor qué recursos son prioritarios (por si ocupan más espacio visual en la página o es más útil que el usuario los vea antes), de forma que, en el flujo de respuestas multiplexadas, el servidor enviará antes esos recursos más significativos.
- Esta nueva optimización de la conexión, hace posible que un cliente emplee un menor número de conexiones para cargar una página. En versiones anteriores, un cliente abría varias conexiones, aunque haga *pipelining* dentro de cada una, para ir obteniendo los recursos. Con HTTP/2.0 esto no es necesario, ya que la multiplexación consigue el mismo resultado (obtener los recursos de manera más rápida) consumiendo una única conexión, con lo que se reduce la carga sobre el servidor y sobre el equipamiento intermedio de red. Además, para mejorar aún más la velocidad, HTTP/2.0 comprime las cabeceras mediante un algoritmo denominado HPACK (RFC 7541[22]), mientras que las versiones anteriores comprimían solo el cuerpo de las páginas.
- HTTP/2.0 incorpora como novedad el *server push,* es decir, el servidor puede enviar recursos al cliente sin que éste los haya solicitado. El ejemplo habitual para esta característica es el de una página que el navegador descarga, procesa y una vez procesada, solicita una serie de recursos del servidor. En el servidor, el creador de la página ya sabe qué recursos va a solicitar el cliente tras el procesamiento inicial y puede utilizar esa información para enviarlos todos desde el principio sin esperar a que el cliente los solicite. De esta manera, se consigue que la página cargue más rápido y se minimiza el número de peticiones al servidor.

Estructura de una petición y de una respuesta HTTP

En general, la estructura básica de una petición o respuesta en HTTP es la siguiente:

Acción	Contenido	Ejemplo Petición Cliente	Ejemplo Respuesta Servidor
1	Petición o Respuesta	GET /index.html HTTP/1.0	HTTP/1.0 200 OK
2..N	Cabeceras	User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)	Server: Apache/2.2.14 (Win32)
N+1	Línea en blanco		
N+2..M	Cuerpo del mensaje		<html> <body> Hello, World! </body> </html>

Tabla 6. Estructura de una petición y de una respuesta HTTP.

Diseño de software

Todo proyecto de software debe definir sus objetivos, los criterios que determinen cuando se ha completado, así como los distintos hitos del plan de trabajo para llevar el proyecto a buen término. Estos puntos se reflejan en un *Software Design Document* (SDD)[23][24] o Documento de Diseño de Software.

Elaboraremos una especificación funcional que indique el comportamiento esperado, así como qué partes del estándar HTTP soportaremos. Por otra parte, también detallaremos una especificación técnica en la que definiremos el flujo de ejecución de cada implementación. Todo ello formará parte de esta memoria a modo de documento de diseño de software.

Especificación funcional

Los requisitos funcionales que soportarán los servidores web desarrollados, tanto a nivel de estándar HTTP como de funcionamiento del servidor en general, serán los siguientes:

- Cabeceras del cliente soportadas (para registrarlas en un log):
  - User-Agent
  - From
- Cabeceras del servidor soportadas (se rellenarán por el servidor en cada respuesta):
  - Server
  - Date
  - Content Type
  - Content-Length
- Comandos de HTTP soportados por el servidor:
  - GET: Si el recurso existe, lo devolverá en el cuerpo de la respuesta.
- Códigos de respuesta HTTP soportados por el servidor:
  - 200 OK: El recurso existe y se enviará en el cuerpo de la respuesta.
  - 404: El recurso no existe.
  - 501: El cliente ha solicitado un método no soportado por el servidor.
- Todas las peticiones se registrarán en un fichero de log, basado en el Common Log Format definido por el W3C (el que usa el servidor web Apache) [29]:
  - IP_cliente usuario fechayhora “petición” codigodeestado bytes
  - Ejemplo: 127.0.0.1 david [20/Oct/2016:23:38:50 +0100] “GET /index.html HTTP/1.0” 200 1536
- El registro en el fichero log debe ser desconectable, para que no influya en las mediciones de rendimiento.
- El servidor debe ser configurable, al menos en los siguientes parámetros:
  - Puerto de escucha.
  - Directorio raíz desde el que se sirven ficheros.
  - Nivel de *logging*
  - Concurrencia máxima
- Debe soportar varias peticiones concurrentes.
- La estructura del código debe ser tal que se debe maximizar la parte paralelizable. Como establece la Ley de Amdahl, la escalabilidad de un algoritmo será mayor cuanto mayor sea su parte paralelizable.

Especificación técnica

Nos basaremos en el estándar UML (Unified Modeling Language) para definir el comportamiento de cada uno de los 3 servidores web. En concreto, los diagramas que emplearemos serán:

- Diagrama de secuencia [25][26]: Explica las diferentes interacciones entre el cliente y el servidor, de la siguiente manera:

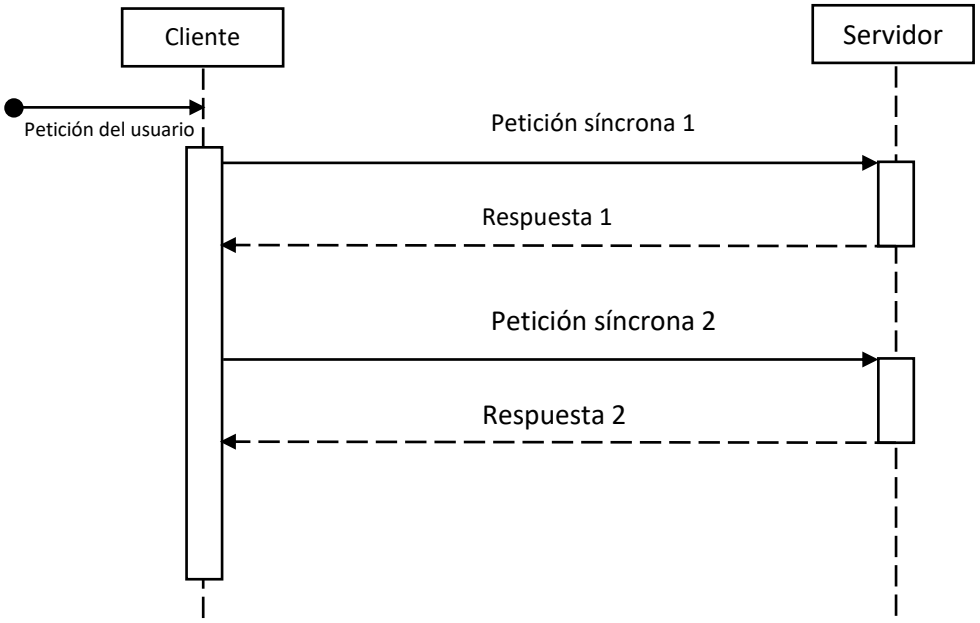

Fig. 6. Diagrama de secuencia UML.

Diagrama de actividad con *swimlanes* [27][28]: Indicará, para cada modelo de concurrencia, el comportamiento de los diferentes componentes del sistema, con un carril (*swimlane*) para cada componente principal como, por ejemplo: Proceso principal, proceso 1, proceso 2 o *thread* 1, *thread* 2, … etc., de la siguiente forma:

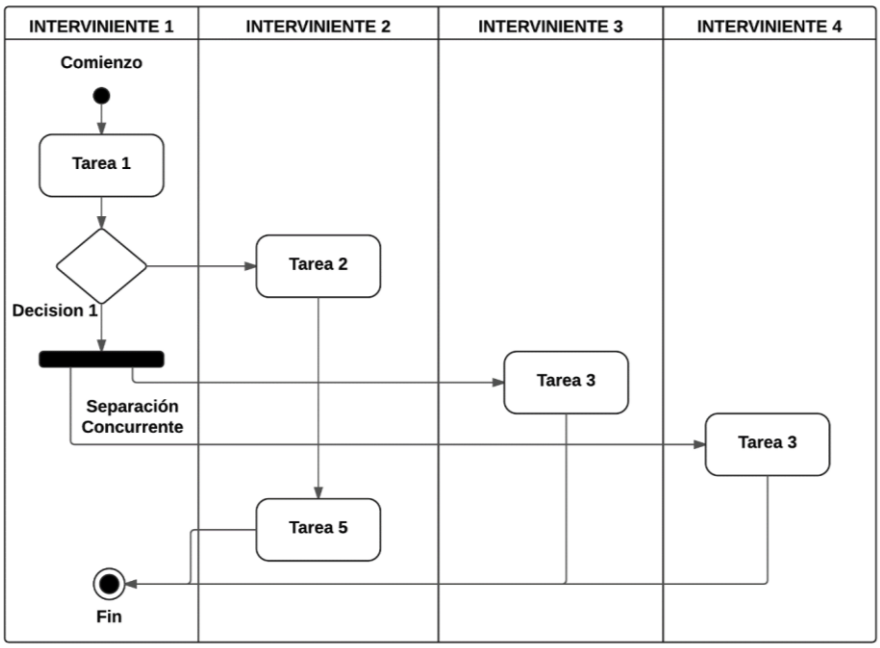

Fig. 7. Diagrama de actividad/*swimlanes* UML.

- Diagrama de secuencia cliente / servidor

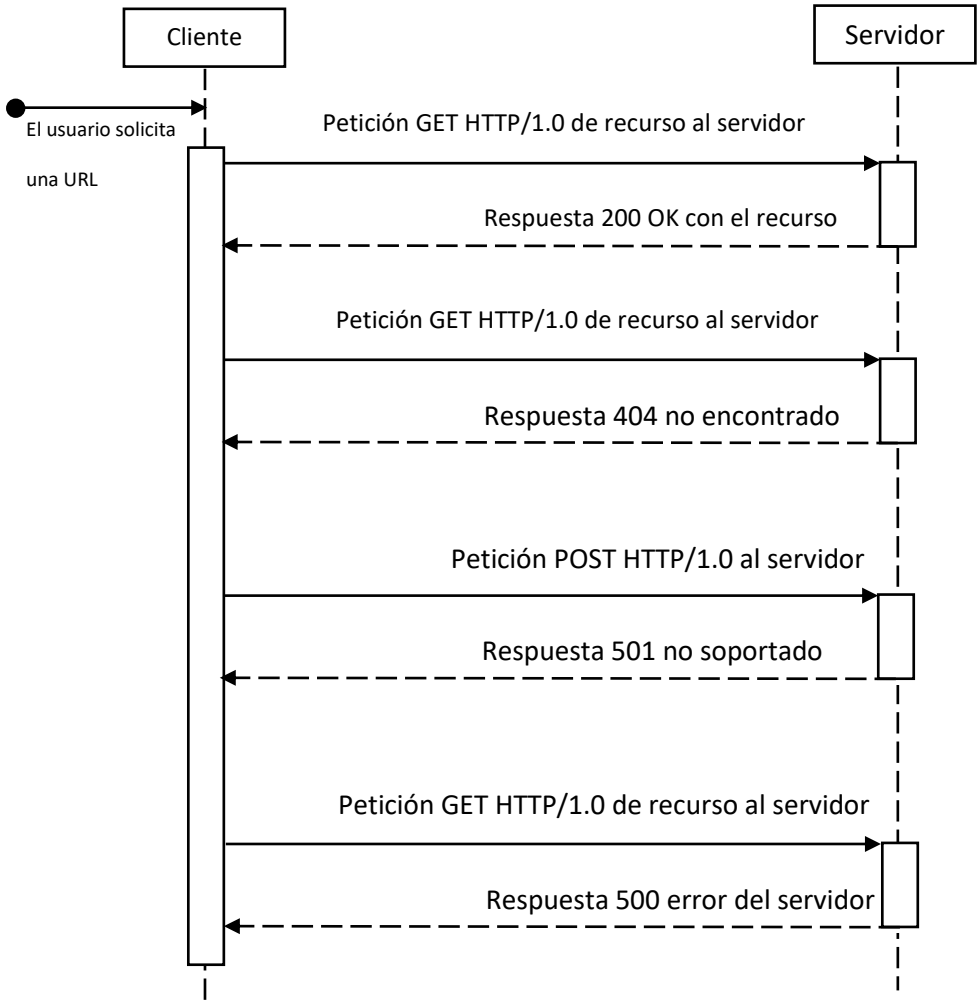

Fig. 8. Diagrama de secuencia cliente/servidor HTTP.

Comentarios a los modelos

De acuerdo con la ley de Amdahl, debemos tratar de maximizar la parte paralelizable del código, ya que así, para una carga de trabajo constante, cuanto más aumenten los recursos, más aumentará el rendimiento.

A nivel de diseño, esto quiere decir que debemos traspasar el control de una petición cuanto antes al subproceso, *thread* o sistema de eventos y procesar lo mínimo posible en el proceso/*thread*/función principal. Si bien sería posible recibir una conexión, esperar el header de la petición y en función de si el header es correcto o no, pasar a otro proceso/*thread*/función la información para que devuelva el recurso o dar error directamente en el proceso/*thread*/función principal, esto no maximizaría la parte paralelizable del código. Para conseguirlo, en cuanto recibamos una nueva conexión debemos pasarla al proceso/*thread*/función cuanto antes, de forma que el proceso/*thread*/función principal quede libre para recibir una nueva conexión lo antes posible y maximicemos así la concurrencia.

Notemos que, aunque el modelo de procesos y el de *threads* son muy similares, su gestión por parte del sistema operativo es diferente. Así, en el modelo basado en procesos, si el proceso principal muere por algún motivo (ya sea una salida producida por el propio programa o bien un error que provoque que el proceso se cierre), los procesos hijos seguirán ejecutándose, pero en estado zombie. Es decir, en el caso de Linux, el proceso INIT del sistema los hereda. En contraste, en el modelo basado en *threads*, si el *thread* principal muere o se cierra, todos los demás mueren con él, puesto que es un único proceso para el sistema operativo.

El modelo de eventos tiene su principal diferencia con los otros procesos en que todo ocurre en un mismo proceso, las separaciones verticales del diagrama de actividad con swimlanes no indican aquí separación entre procesos o *threads*, sino lo que se ejecuta en la función principal y lo que se ejecuta en los diferentes *callbacks* asociados a eventos. En la función principal se definen las funciones que serán llamadas cuando se reciba el evento asociado.

Criterio de finalización

El proyecto se considerará completo cuando implemente todas las funcionalidades definidas en la especificación funcional en los 3 modelos de concurrencia y su flujo de ejecución se corresponda con los diseños indicados. Por tanto, para considerarse completo debe cumplir:

- Funcionalidades:
    - Cabeceras del cliente soportadas.
    - Cabeceras del servidor soportadas.
    - Comandos de HTTP soportados.
    - Códigos de respuesta HTTP soportados.
    - Registro a fichero log.
    - Configuración básica.
    - Soporte de peticiones concurrentes.
    - Código que maximice la parte paralelizable.
- Flujo de ejecución:
    - Según diseños de swimlanes.

- Diagrama de actividad con *swimlanes* para cada modelo de concurrencia
    - Modelo basado en procesos

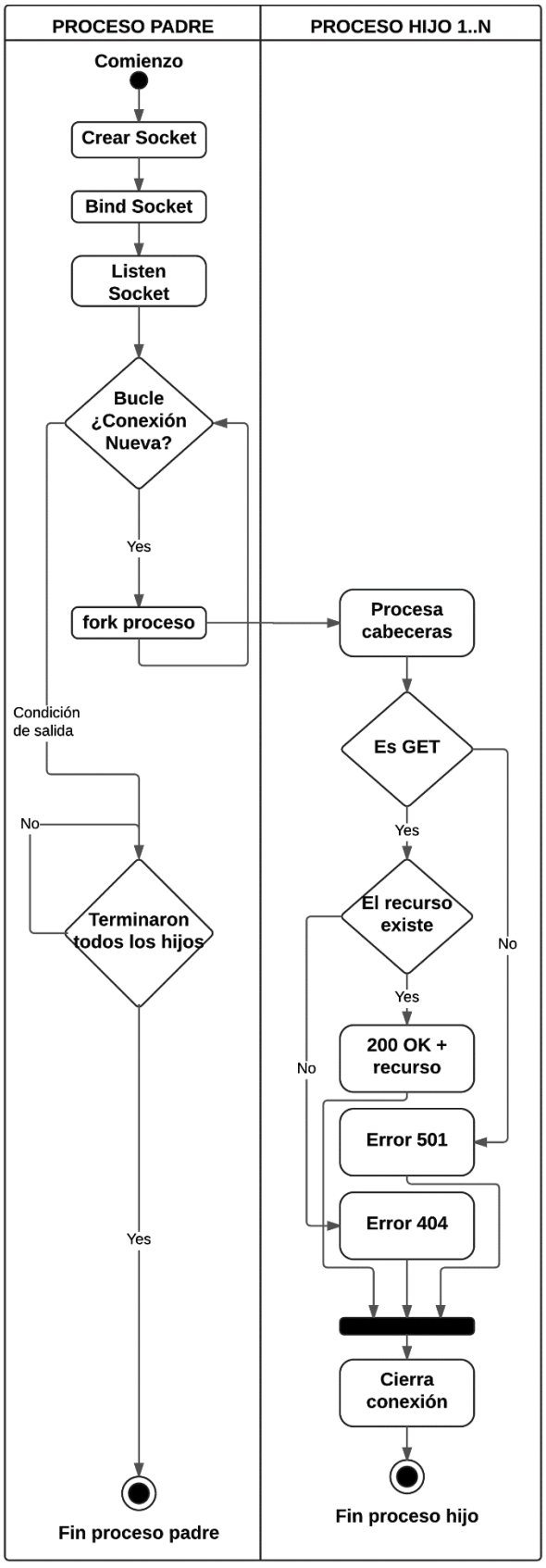

Fig. 9. Diagrama de secuencia cliente/servidor HTTP basado en procesos.

- Modelo basado en *threads*

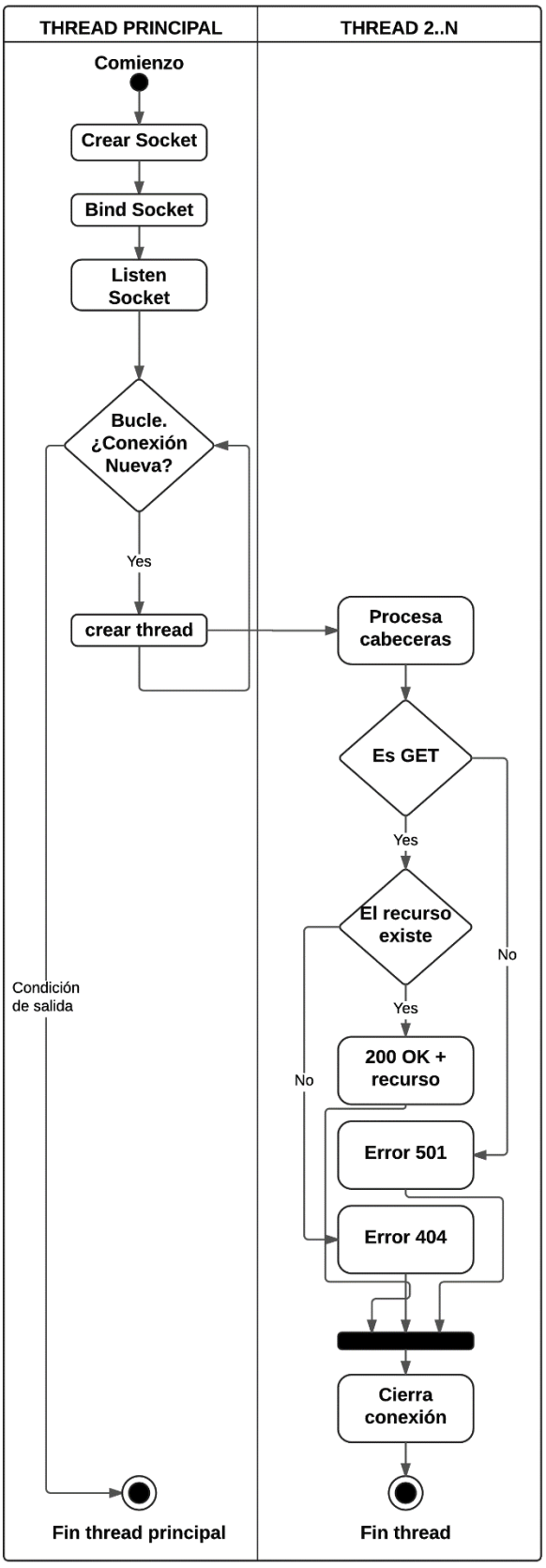

Fig. 10. Diagrama de secuencia cliente/servidor HTTP basado en *threads*.

- Modelo basado en eventos

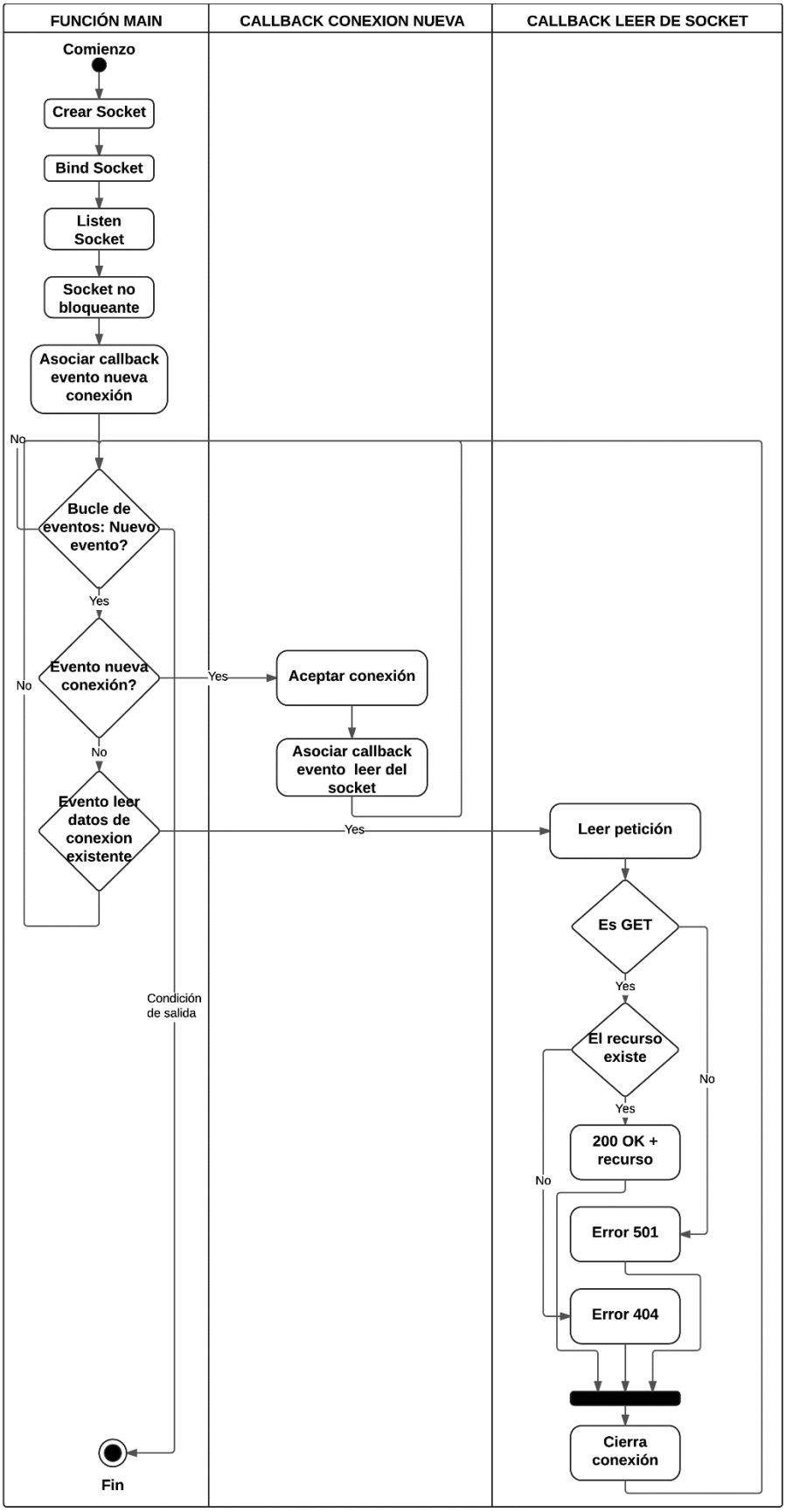

Fig. 11. Diagrama de secuencia cliente/servidor HTTP basado en eventos.

Implementación

La implementación se ha denominado, de forma genérica para los tres modelos, cserver: concurrent server.

Modelos implementados

Para implementar los 3 modelos de concurrencia (procesos, *thread*s o hilos y eventos) emplearemos los siguientes mecanismos:

- **Procesos**: Crearemos procesos nuevos mediante la llamada de sistema fork() [30]. Esta llamada hace que el *kernel* cree una copia nueva del proceso actual, que se sigue ejecutando en el mismo punto tanto en el padre como en el hijo, pero que devuelve un valor diferente a esa llamada en el caso del padre y del hijo, lo que permite, a partir de ese punto, variar la ejecución en uno y otro caso.
- **Threads:** Emplearemos POSIX *Thread*s que es un modelo de concurrencia mediante *thread*s independiente del lenguaje. En los sistemas UNIX, el interfaz está definido en el estándar IEEE POSIX 1003.1c [31]. En concreto, usaremos la implementación proporcionada por la librería libphtread en la implementación de GNU [32], que habitualmente se distribuye de forma conjunta con la libc. Libpthread nos proporciona funciones para crear y destruir *thread*s, así como también para gestionar *mutex*, *spinlocks*, semáforos, barreras, etc.
- **Eventos:** Para implementar el modelo de eventos usaremos libevent [33] que incorpora un modelo de gestión de concurrencia basado en eventos en el cual se puede asociar una función *callback* a un evento de tal forma que cuando el evento ocurra, la función será llamada.

Detalle de implementación

La implementación se ha realizado en C sobre un sistema Linux. Por un lado, se han creado cuatro módulos que serán comunes a los tres modelos de concurrencia, como son: la gestión de las peticiones y las respuestas HTTP, la gestión del *logging*, la gestión de errores y la gestión de la configuración y por otro, tres programas principales, uno por cada modelo de concurrencia.

Código común a todos los modelos

El código común a todos los modelos es el siguiente:

- Módulo cerror: Proporciona rutinas relacionadas con la gestión de errores, tanto con los errores ERRNO estándar de C, como mensajes de error personalizados de la aplicación.
- Módulo clogger: Registra la actividad del servidor en el syslog. Compatible tanto con syslog convencional como con systemd (journald+rsyslogd)
- Módulo cconfig: Implementa las funciones relacionadas con la configuración del servidor: Parámetros de línea de comandos, valores por defecto, etc.
- Módulo chttp: Contiene las funciones que se encargan de leer las peticiones HTTP y de enviar las respuestas HTTP pertinentes (códigos de respuesta, errores, ficheros, etc.)

Código diferente según el modelo

- cserver-procesos: Contiene las funciones específicas del modelo de procesos, basado en fork.
- cserver-threads: Contiene las funciones específicas del modelo de threads o hilos, basado en POSIX Threads según la implementación GNU de libpthread.
- cserver-events: Contiene las funciones específicas del modelo de eventos, basado en libevent.

Descripción de los ficheros

El código está organizado con la siguiente estructura:

Tipo	Nombre	Descripción
Directorio	cconfig	Módulo cconfig
Fichero	cconfig/cconfig.c	Módulo cconfig – código
Fichero	cconfig/cconfig.h	Módulo cconfig – cabeceras
Directorio	cerror	Módulo cerror
Fichero	cerror/cerror.c	Módulo cerror – código
Fichero	cerror/cerror.h	Módulo cerror – cabeceras
Directorio	chttp	Módulo chttp
Fichero	chttp/chttp.c	Módulo chttp – código
Fichero	chttp/chttp.h	Módulo chttp – cabeceras
Directorio	clogger	Módulo clogger
Fichero	clogger/clogger.c	Módulo clogger – código
Fichero	clogger/clogger.h	Módulo clogger – cabeceras
Directorio	www	Directorio raíz del servidor web
Fichero	www/index.html	Fichero html de prueba, sólo texto
Fichero	www/imagetest.html	Fichero html de prueba, texto e imagen
Fichero*	www/paisaje.jpg	Fichero html de prueba, sólo imagen (JPG)
Fichero*	www/test.png	Fichero html de prueba, sólo imagen (PNG)
Directorio	.	Directorio raíz del proyecto
Fichero	compila.sh	Shell script para compilar, contingencia del Makefile
Fichero	Makefile	Fichero make para compilar los tres binarios
Fichero**	cs-procesos	Binario del servidor concurrente (cserver), versión procesos
Fichero**	cs-threads	Binario del servidor concurrente (cserver), versión *thread*s
Fichero**	cs-events	Binario del servidor concurrente (cserver), versión eventos

Tabla 7. Listado de ficheros del código fuente.

*Las imágenes incluidas son imágenes libres de derechos de autor y pueden ser reutilizadas libremente.
**Los binarios solo aparecerán una vez se ejecute make o el script compila.sh.

Los resultados de las pruebas de rendimiento están organizados con la siguiente estructura:

Tipo	Nombre	Descripción
Directorio	results/1core	Resultados de las pruebas con el servidor limitado a 1 core
Fichero	salida_eventos_1core_21_12_2016_20_04.txt salida_eventos_1core_21_12_2016_20_04.RATES.txt salida_eventos_1core_21_12_2016_20_04.LONGEST.txt	Resultados modelo de eventos, servidor con 1 core. RATES: Filtro por transaction rate. LONGEST: Filtro por longest transaction
Fichero	salida_procesos_1core_21_12_2016_19_02.txt salida_procesos_1core_21_12_2016_19_02.RATES.txt salida_procesos_1core_21_12_2016_19_02.LONGEST.txt	Resultados modelo de procesos, servidor con 1 core. RATES: Filtro por transaction rate. LONGEST: Filtro por longest transaction
Fichero	salida_threads_1core_21_12_2016_18_35.txt salida_threads_1core_21_12_2016_18_35.RATES.txt salida_threads_1core_21_12_2016_18_35.LONGEST.txt	Resultados modelo de *thread*s, servidor con 1 core. RATES: Filtro por transaction rate. LONGEST: Filtro por longest transaction
Directorio	results/multicore	Resultados de las pruebas con el servidor con todos los cores activados.
Fichero	salida_eventos_21_12_2016_16_11.txt salida_eventos_21_12_2016_16_11.RATES.txt salida_eventos_21_12_2016_16_11.LONGEST.txt	Resultados modelo de eventos con 1 servidor web, servidor multicore. RATES: Filtro por transaction rate. LONGEST: Filtro por longest transaction
Fichero	salida_eventos_url4_21_12_2016_16_43.txt salida_eventos_url4_21_12_2016_16_43.RATES.txt salida_eventos_url4_21_12_2016_16_43.LONGEST.txt	Resultados modelo de eventos con 4 servidores web, servidor multicore. RATES: Filtro por transaction rate. LONGEST: Filtro por longest transaction
Fichero	salida_eventos_url8_21_12_18_21.txt salida_eventos_url8_21_12_18_21.RATES.txt salida_eventos_url8_21_12_18_21.LONGEST.txt	Resultados modelo de eventos con 8 servidores web, servidor multicore. RATES: Filtro por transaction rate. LONGEST: Filtro por longest transaction
Fichero	salida_procesos_21_12_2016_16_25.txt salida_procesos_21_12_2016_16_25.RATES.txt salida_procesos_21_12_2016_16_25.LONGEST.txt	Resultados modelo de procesos, servidor multicore. RATES: Filtro por transaction rate. LONGEST: Filtro por longest transaction
Fichero	salida_threads_21_12_2016_18_36.txt salida_threads_21_12_2016_18_36.RATES.txt salida_threads_21_12_2016_18_36.LONGEST.txt	Resultados modelo de *thread*s, servidor multicore. RATES: Filtro por transaction rate. LONGEST: Filtro por longest transaction
Directorio	results/resources	Resultados de las pruebas de consumo de CPU
Fichero	cpu-eventos.txt	Resultados de consumo de CPU para el modelo de eventos.
	cpu-procesos.txt	Resultados de consumo de CPU para el modelo de procesos.
	cpu-threads.txt	Resultados de consumo de CPU para el modelo de *thread*s.

Tabla 8. Listado de ficheros del resultado de las pruebas.

Descripción del código

A continuación, comentaremos brevemente todas las funciones creadas:

cconfig

cconfig_init
Fija los parámetros por defecto para todas las opciones. El orden de prioridad será siempre en primer lugar los indicados por la línea de comandos y en segundo lugar los valores por defecto.

cconfig_cmdline
Lee las opciones desde la línea de comandos, comprueba que tengan valores válidos y en caso de que se indique la opción para mostrar la ayuda, llama a la función que lo hace.

cconfig_show_usage
Imprime en pantalla la ayuda sobre las opciones.

cconfig_show_config
Imprime en pantalla los valores actuales de las opciones.

cerror

cerror_show
Imprime en pantalla la cadena de texto asociada a un error estándar ERRNO de C y también la registra en el log del sistema.

cerror_show_string
Imprime en pantalla y en el log del sistema la cadena de texto pasada como parámetro.

clogger

clogger_init
Abre el log del sistema para escribir.

clogger_log
Registra el mensaje indicado en el log del sistema.

clogger_stop
Cierra el log del sistema.

chttp

get_file_content_type
Identifica el tipo de fichero y devuelve el content-type asociado de la cabecera HTML.

send_to_socket
Envía la cadena indicada como parámetro a un *socket* previamente abierto.

reply_http_request
Envía una respuesta HTTP a una petición previa, de acuerdo con las cabeceras de la petición y la configuración de cserver. Si la respuesta es correcta, indicará un código 200 OK y el recurso, si no la encuentra indicará un error 404 y una página de error y si es un método está soportado un error 501 y una página de error.

read_http_request
Lee una petición http de un *socket* abierto y almacena las cabeceras.

main – Procesos

process_new_connection

Una vez establecida una nueva conexión con un cliente, lee las cabeceras de la petición y envía la respuesta (mediante chttp).

main

Lee la configuración de la línea de comandos / por defecto, crea y configura el *socket* de servidor para escuchar en el puerto indicado en la configuración. Cuando llega una nueva conexión, crea un proceso hijo nuevo para tratarla y controla que no se sobrepase el máximo de concurrencia indicado en la configuración, gestionando también los hijos que van finalizando.

main – Threads

process_new_connection

Una vez establecida una nueva conexión con un cliente, lee las cabeceras de la petición y envía la respuesta (mediante chttp). Además, incluye un *mutex* para que al gestionar los límites de concurrencia no se produzca una *race condition* con la función main (que se ejecuta en otro *thread* paralelo).

main

Lee la configuración de la línea de comandos / por defecto, crea y configura el *socket* de servidor para escuchar en el puerto indicado en la configuración. Cuando llega una nueva conexión, crea un nuevo *thread* configurado como detached (lo que permite que no haya que gestionar el *thread* cuando termina). Al igual que en process_new_connection, existe un *mutex* para que al gestionar los límites de concurrencia no se produzca una *race condition*.

main – Eventos

process_new_connection

Una vez establecida una nueva conexión con un cliente, lee las cabeceras de la petición y envía la respuesta (mediante chttp). Esta función será configurada como *callback* para el evento de lectura en un *socket*.

accept_callback

Función *callback* que es llamada cuando se recibe una nueva conexión en un *socket*, a su vez, asocia a los eventos de lectura sobre el *socket* un nuevo evento que llamará a la función process_new_connection.

main

Lee la configuración de la línea de comandos / por defecto, crea y configura el *socket* de servidor para escuchar en el puerto indicado en la configuración. A diferencia de los main anteriores, aquí además configuramos el *socket* como no bloqueante y asociamos al evento “nueva conexión sobre el *socket*” la función *callback* accept_callback.

Problemas encontrados

Modelo de procesos

En el modelo de procesos, es importante tener en cuenta que al crear un proceso hijo, debemos cerrar el *socket* de servidor en el proceso hijo, puesto que es una copia del *socket* del padre que es el que debe seguir abierto para aceptar nuevas conexiones.

Modelo de threads

La implementación del modelo de *threads* es la que más problemas ha planteado. En primer lugar, se alcanzaba una concurrencia máxima baja (inferior a la indicada en las opciones) porque se producía una *race condition* en la variable que controlaba el número de *threads* en ejecución: Se modificaba tanto en la función main en el *thread* principal como en la función process_new_connection en todos los otros *thread*s, con lo cual, algunas de las modificaciones se perdían y el servidor no era capaz de alcanzar su máximo rendimiento. Una vez colocados *mutex* en las dos funciones alrededor de las líneas que modifican la variable, el *throughput* en transacciones por segundo aumento en aproximadamente un 50%.

Además, fue necesario cambiar ciertas funciones de la libc estándar por versiones *thread safe*: strerror-r() en lugar de strerror() y localtime_r() en lugar de localtime().

Es importante notar que, al compartir los *threads* el espacio de memoria, cambian algunos comportamientos respecto a la versión de procesos. En concreto, en procesos al crear un hijo cerrábamos el *socket* del servidor en el hijo, ya que era una copia del padre y quedaba abierto en el padre. Este cierre no es necesario en la versión de *thread*s, puesto que sólo existe una variable compartida para todos los *threads*. Si lo cerráramos al crear un *thread* nuevo, tiraríamos el servidor ya que dejaría de escuchar por el *socket*.

Sin embargo, el problema más difícil de solucionar lo encontramos cuando pasamos de usar el navegador para hacer pruebas (que siempre funcionaban correctamente) a usar el programa de *benchmarking* que lanzaba cientos de conexiones simultáneas. En ese caso, de forma aleatoria daba un segmentation fault. Sin embargo, si se ejecutaba el programa dentro del *debugger* gdb o de valgrind, no ocurría.

Tras varias pruebas, identificamos que el problema únicamente aparecía cuando se generaban un gran número de *threads* en el servidor (centenares) y además de manera muy rápida. Por eso al usar gdb no ocurría, porque la ejecución en el depurador es más lenta. Finalmente, resultó ser un problema de tamaño de pila. Cuando existen decenas de *threads* funciona bien, en cambio, al pasar a centenares de *threads* generados muy rápidamente, el espacio de pila se agotaba. La solución fue doble:

1. Generar los *threads* con un tamaño de pila de 128KB, para evitar que se produzca un *segmentation fault* cuando hay muchos *threads*.
2. Disminuir lo máximo posible el tamaño de pila empleado dentro de las funciones que se ejecutan en el *thread*. Para ello, sustituí la mayor parte de las variables de cierto tamaño declaradas en las funciones de los *threads* por punteros que se reservan dinámicamente, de esta forma estas variables no se almacenarían en la pila.

Con estas dos medidas el fallo quedó resuelto. Veamos ahora porqué al ejecutar dentro de un depurador no ocurría: Al ralentizar la ejecución el depurador, daba tiempo a que los *threads* fueran sirviendo la página y se fueran destruyendo antes de generar nuevos *threads*, lo que hacía que el programa no se saliera de los límites de la pila. En cambio, sin depurador, se generaban *threads* tan rápido como era posible hasta que se llenaba la pila y el programa fallaba.

Además, comenzamos el desarrollo empleando máquinas virtuales y observamos que el rendimiento en transacciones por segundo era bajo y, además, muy similar tanto en procesos como en *threads*. Al pasar a máquina física en el mismo hardware, el comportamiento cambió drásticamente, multiplicándose el rendimiento de forma que, además, el modelo de *threads* resulta ser bastante más rápido que el de procesos, que coincide con lo esperado según la teoría, ya que la creación de un thread y un cambio de contexto entre *threads* son más rápidas que en el caso de procesos.

Modelo de eventos

Para el modelo de eventos, hubo que tener en cuenta que se pierde el control directo sobre el flujo de ejecución, ya que el modelo de eventos funciona mediante *callbacks* asociados a eventos. Se define una función que será llamada cuando se produzca el evento al que se asocia y, además, las funciones *callback* tienen que tener un formato determinado en cuanto a parámetros y valores de retorno, lo que hace que tuviéramos que ajustar nuestras funciones a esa convención de llamada.

Entorno de desarrollo y ejecución

El desarrollo se ha realizado en Linux Mint 18 con Kernel 4.4 y en Linux Debian 8.3 con kernel 3.16.

Los requisitos necesarios para compilar el software son:

- Compilador GCC.
- Makefile (si no estuviera disponible se pueden generar los binarios con el script "compila.sh" incluido).
- Librerías de desarrollo libc.
- Librería de desarrollo libpthread (habitualmente incluida en libc).
- Librería de desarrollo libevent versión 2 o superior.

Cómo crear los binarios

Existen dos maneras de compilar el código para crear los tres binarios:

1. Si el sistema tiene Makefile, simplemente hay que ejecutar make en el directorio para que se creen los tres binarios del servidor: cs-procesos, cs-threads y cs-events
2. Si no tiene Makefile, se puede ejecutar el script compila.sh que llamará a gcc para crear los 3 binarios.

Cómo ejecutar cserver

Al invocar cualquiera de las versiones de cserver, las opciones por defecto son:

- Puerto de escucha TCP: 1080
- Directorio desde el que se sirven las páginas: ./www/
- Logging: 0
- Concurrencia máxima: 1000

Para cambiar cualquiera de estas opciones podemos usar los parámetros:

```
-p port  : Puerto de escucha TCP
-r path  : Path el directorio desde el que se servirán las páginas
-l 0/1/2 : Nivel de logging: 0 – Desactivado, 1 – Normal, 2- Debugging
-c num   : Concurrencia máxima
-h       : Ayuda
```

Por defecto se ha dejado el valor de *logging* a 0 para que no impacte en el rendimiento al hacer las pruebas de rendimiento.
Si en la URL que solicitemos existe un fichero index.html, el servidor la autocompletará internamente y servirá el fichero index.html. Es decir, si solicitamos http://127.0.0.1:1080/ en el navegador y en el directorio www hay un index.html, lo servirá.

Pruebas de validación

Para probar el servidor se pueden emplear los ficheros incluidos en el directorio www/ que son:

Nombre	Descripción
index.html	Fichero html sólo texto, pequeño tamaño, usado para las pruebas de rendimiento.
imagetest.html	Fichero html con texto e imagen, usado como prueba de fichero combinado.
paisaje.jpg	Fichero imagen (JPG), usado para pruebas de ficheros grandes (~0.5 MB).
test.png	Fichero imagen (PNG), usado para probar los content-types.

Tabla 8. Ficheros de prueba incluidos.

De manera similar, podemos comprobar que el servidor devuelve código 404 al solicitar una página que no existe y 501 al solicitar un método no implementado.

Si hemos arrancado el servidor con *logging* a 1 o a 2, podremos observar en el syslog del sistema las peticiones, siguiendo el formato de log de Apache. Si el nivel de *logging* es 2 (*debugging*), veremos además mucha más información de depuración en pantalla para cada petición. Con el *logging* activado, se puede comprobar como en cada petición, el servidor recibe las cabeceras, las procesa, y responde de acuerdo con lo solicitado, rellenando las cabeceras de respuesta pertinentes.

Pruebas de rendimiento

Herramientas

Siege

La principal herramienta utilizada para medir el rendimiento de los tres núcleos de servidor web desarrollados es Siege[35] en su versión 3.0.8 instalada mediante paquete de Debian. La salida habitual de una prueba de rendimiento con Siege es la siguiente:

```
** SIEGE 3.0.8
** Preparing 40 concurrent users for battle.
The server is now under siege...
Lifting the server siege...      done.

Transactions:                 493468 hits
Availability:                 100.00 %
Elapsed time:                   9.16 secs
Data transferred:              76.24 MB
Response time:                  0.00 secs
Transaction rate:           53872.05 trans/sec
Throughput:                     8.32 MB/sec
Concurrency:                   38.77
Successful transactions:      493468
Failed transactions:               0
Longest transaction:            0.01
Shortest transaction:           0.00
```

Es decir, da el número total de transacciones enviadas, el % de disponibilidad del servidor, el tiempo total empleado en procesar todas las transacciones, el número de datos transferidos, el tiempo de respuesta, el número de transacciones por segundo, el ancho de banda en tráfico de red, la concurrencia, el número de transacciones exitosas y fallidas y, por último, el tiempo en responder la transacción más larga y la más corta. De estos datos nos interesan sobre todo el número de transacciones por segundo y el tiempo en procesar la transacción más larga.

En detalle:

- El número de transacciones *(Transactions)* indica el total de peticiones HTTP que se han realizado, en general, en las pruebas llegan a alcanzarse más de 700.000 transacciones en una única prueba y para cada modelo de servidor realizamos un total de 33 pruebas con diferentes concurrencias. Es decir, para medir el rendimiento de cada modelo empleamos varios millones de peticiones.

- La disponibilidad (*Availability)* indica el porcentaje de transacciones exitosas vs fallidas. Durante las pruebas es casi siempre del 100% con 0 transacciones fallidas. No obstante, en algunas ocasiones es posible que aparezca algún mensaje de aviso en el servidor en las pruebas de máxima concurrencia, que coincide con el momento en que Siege finaliza su ejecución, lo que ocurre es que Siege cierra alguna conexión a medias al terminar. En todo caso hablamos de que, si ocurre, es del orden de 1 conexión por cada varios millones, lo que no es estadísticamente significativo para las mediciones.

- El tiempo transcurrido (*Elapsed time*) especifica el tiempo total de duración de la prueba.

- Los datos transferidos (*Data transferred)* miden el total de bytes transmitidos durante la prueba.

- El tiempo de respuesta (*Response Time)* es el tiempo medio que ha tardado el servidor en responder cada petición. Observaremos que en muchas de las pruebas es 0.00 porque no tiene suficiente resolución temporal para medir la respuesta, ya que el servidor es muy rápido respondiendo.

- El número de transacciones por segundo (*Transaction Rate)* es el principal indicador de rendimiento de cada modelo de servidor web, puesto que indica el número de peticiones que el servidor ha sido capaz de completar por segundo.

- El *throughput* indica la media de bytes transferidos cada segundo. Este valor es importante puesto que debe encontrarse holgadamente por debajo de la máxima velocidad del enlace de red entre el cliente y el servidor. Como comentaremos más adelante, si se emplean ficheros muy grandes para las pruebas (p.ej. imágenes jpeg de varios MB) se saturará el enlace de red con muy pocas peticiones – veremos que el throughput alcanza aproximadamente 125 MB/s que corresponden al enlace gigabit – y la prueba no nos servirá para medir el rendimiento del servidor ya que estará limitado por el enlace.

- La concurrencia (*Concurrency)* [36] es la media de peticiones simultáneas en cada momento. Este valor será siempre muy próximo al indicado en el parámetro -c de la línea de comandos.

- Transacciones exitosas (*Successful transactions)* mide el número de peticiones que han tenido éxito.

- Transacciones fallidas (*Failed transactions)* mide el número de peticiones que han fallado.

- Transacción más larga (*Longest transaction)* y más corta (*Shortest transaction)* mide el tiempo que ha tardado la petición que más ha tardado en ser respondida por el servidor y la que menos tiempo ha tardado, respectivamente.

Un ejemplo de línea de comandos utilizada es el siguiente:

```
siege -b -t 10s -c 400 http://192.168.1.34:1080/
```

Los parámetros empleados son:

-b: Modo *benchmark*, no transcurre tiempo entre cada petición, las lanza tan rápido como es posible. No es un escenario realista en cuanto al uso típico de un servidor web pero sí es ideal para medir el rendimiento del servidor. Se trata de una "inundación" de peticiones tan rápido como el cliente es capaz de realizarlas (hasta el umbral de concurrencia). Más adelante observaremos que se alcanza el límite de capacidad del cliente.
- t 10s: La prueba tiene una duración de 10 segundos.
-c 400: Umbral de concurrencia máxima, las transacciones que se ejecutan simultáneamente en cada momento.

SAR

Además, también se ha medido el consumo de CPU en el servidor de cada uno de los 3 modelos de concurrencia cuando el número de transacciones por segundo era el mismo. Para hacerlo, se ha empleado la herramienta del sistema SAR, que forma parte del conjunto de utilidades Sysstat [37]. La línea de comandos empleada para esta medición es:

```
sar 1
```

Este comando nos proporciona mediciones del uso de CPU cada segundo, la salida es de la forma:

```
Linux 4.4.0-21-generic (kraken)      12/21/2016     _x86_64_       (8 CPU)

10:51:48 PM     CPU     %user     %nice   %system   %iowait    %steal     %idle
10:51:49 PM     all      0,25      0,00      0,75      0,13      0,00     98,87
10:51:50 PM     all      2,28      0,00      9,64      0,00      0,00     88,07
10:51:51 PM     all      2,29      0,00     10,04      0,00      0,00     87,67
10:51:52 PM     all      2,30      0,00      9,58      0,26      0,00     87,87
```

Aunque SAR es capaz de informar del consumo de cada core, elegimos una representación global de consumo de CPU y no una core a core puesto que los interesa es conocer el consumo total de cada modelo. Por tanto, el valor que nos interesa será el resultado de la operación: 100 - %idle y eso nos dará el % de ocupación total de todos los cores.

Equipos de prueba

Las especificaciones de los equipos empleados para las pruebas de rendimiento son las siguientes:

- Cliente:
  - CPU Intel Core i3 6100T (2 cores - 4 con HyperThreading)
  - Memoria DDR3 2133 8GB (2x4GB)
  - Sistema Operativo: Linux Debian 8.3 con Kernel 3.16

- Servidor:
  - CPU Intel Core i7 4770K (4 cores - 8 con HyperThreading)
  - Memoria DDR3 2400 16GB (2x8GB)
  - Sistema Operativo: Linux Mint 18 con Kernel 4.4.0

Ambos sistemas están conectados a la misma red local mediante gigabit ethernet.

Detalles de las pruebas

Los 3 servidores web (procesos, *threads* y eventos) se han arrancado para las pruebas con las siguientes opciones:

- Logging=0 para que el *logging* a fichero/pantalla no impacte en el rendimiento
- Concurrencia=1000 número máximo de procesos/*threads*/cola de conexiones, por encima de los límites de concurrencia que se fijarán para cada prueba en el lado cliente.
- Puerto=1080
- Directorio=./www/

Por sencillez, se han fijado esos mismos parámetros como los parámetros por defecto de los 3 servidores web.

Para evaluar el rendimiento, se harán pruebas con las siguientes concurrencias limitadas en el cliente (en Siege): 4, 16, 32, 48, 64, 80, 100, 200, 400, 600 y 800. No se harán pruebas con concurrencias mayores puesto que el cliente no es capaz de generar las peticiones más rápido. Se realizará cada medición 3 veces y se tomará como valor la media de esas tres mediciones. Así pues, haremos un total de 33 pruebas para cada modelo, lo que nos da un total de varios millones de peticiones. Es importante notar que, debido a que estamos empleando el modo benchmark de Siege, una concurrencia de 800 es muy superior a un uso habitual de un servidor web con usuarios reales. Con estos valores de concurrencia, obtendremos rendimientos de varias decenas de miles de peticiones por segundo contestadas por el servidor.

Además, para las pruebas emplearemos el fichero index.html suministrado dentro del directorio www. Se trata de un fichero html sencillo que sólo contiene texto. Se han realizado también pruebas con grandes ficheros .jpg en lugar de con ficheros pequeños y, lo que se ha observado en ese caso, es que, debido al tamaño del fichero, se alcanza rápidamente el máximo de capacidad de transmisión de la conexión gigabit con pocas conexiones. Por tanto, no se estresa al servidor, que es lo que se persigue en estas pruebas.

Como indicábamos en los capítulos iniciales en los que definimos los diferentes modelos de concurrencia, el modelo de eventos ejecuta su bucle de eventos en un único *thread* por lo que emplea un solo core del procesador. Así pues, para compararlo con los modelos de procesos y *threads*, que escalan naturalmente a tantos cores como estén disponibles, ejecutaremos varios procesos servidores con el modelo de eventos. Estos servidores escucharán en diferentes puertos TCP por lo que lanzaremos Siege en el cliente con la opción -f urls.txt que contendrá varias urls (ver Anexo B) cada una apuntando a uno de los servidores web. Realizaremos la medición de rendimiento con 1, 4 y 8 servidores basados en eventos, ya que se ha observado que, para el hardware utilizado, no existen mejoras de rendimiento más allá de 8 servidores. Los resultados de estas pruebas están en el directorio results/multicore.

Si bien de esta última manera evaluamos la capacidad máxima de servir peticiones del servidor con cada algoritmo, para medir el rendimiento de cada uno cuando sólo existe un core en el sistema, realizaremos una segunda tanda de pruebas limitando en BIOS el número de cores activos del servidor a 1. Los resultados de estas pruebas están en el directorio results/1core.

Además, también mostraremos los tiempos máximos de respuesta, es decir, el tiempo que ha tardado la transacción que más tiempo ha llevado responder, para cada modelo y cada carga de concurrencia.

Finalmente, para evaluar el uso de CPU de cada algoritmo hemos planteado una prueba con concurrencia fijada a 800 en Siege, puesto que como se verá más adelante, para esa concurrencia el número de transacciones que atiende cada algoritmo se iguala, de forma que podemos comparar el consumo de CPU en ese momento. Para ello se ha generado carga durante 30 segundos mediante Siege y empleando la utilidad del sistema SAR [37][38] se ha medido el consumo de CPU.

Por tanto, las pruebas que realizaremos serán:

- Rendimiento en transacciones/segundo de los 3 modelos de servidor, con los 4 cores activados en el servidor (multicore).
  Nos permite observar para qué concurrencia máxima cada modelo es más eficaz, así como comparar también entre los 3 modelos cuál es más eficiente.

- Rendimiento en transacciones/segundo del modelo de eventos en multicore, con 1 proceso, 4 procesos y 8 procesos (el rendimiento no aumenta más allá de los 8 procesos).
  Esta prueba nos proporciona información sobre cómo aumenta el rendimiento del modelo de eventos a medida que se incrementa el número de servidores (cada uno escuchando en un puerto TCP diferente).

- Diferencia de rendimiento en transacciones/segundo del modelo de eventos con un proceso en multicore vs 1 core.
  Esta prueba nos permite observar la diferencia de rendimiento entre ejecutar el modelo de eventos en un único *thread* en un sistema con 1 core o hacerlo en un sistema con varios cores, ya que el sistema siempre tiene otros procesos ejecutándose.

- Rendimiento en transacciones/segundo de los 3 modelos de servidor con un 1 core activado en el servidor.
  En esta prueba podemos ver el rendimiento de cada uno de los 3 modelos cuando el servidor únicamente tiene un core disponible. Es una prueba que permite comprobar qué modelo sería más eficaz cuando el número de cores es reducido (por ejemplo, para un servidor virtual en cloud donde se paga por core, veremos qué modelo es más eficiente para un único core).

- Tiempo de la transacción más lenta de los 3 modelos, con los 4 cores activados en el servidor.
  Con esta comparativa vemos qué modelo puede presentar una peor experiencia para el usuario final (en forma de esperas), en concreto, veremos que el modelo más rápido no es el que mejores tiempos de respuesta tiene.

- Uso de CPU de los 3 modelos de servidor para concurrencia 800, con los 4 cores activados en el servidor.
  Con esta prueba buscamos averiguar qué modelo de concurrencia es más eficiente desde el punto de vista de consumo de CPU en el servidor. Para ello, para un mismo número de transacciones comparamos el consumo de procesador para cada uno de los 3 modelos de servidor.

Ficheros de resultados

Los ficheros de resultados de las pruebas de rendimiento con Siege constan de 3 ficheros para cada medición:

1. Fichero de salida de siege salida_modelo_fecha_hora.txt
2. Fichero filtrado a partir del fichero de salida inicial que sólo contiene los *transaction rates*: salida_modelo_fecha_hora.RATES.txt
3. Fichero filtrado a partir del fichero de salida inicial que sólo contiene los *Longest Transaction:* salida_modelo_fecha_hora_LONGEST.txt

- Directorio results/multicore:
  - Modelo de eventos en multicore con 1 proceso:
    - salida_eventos_21_12_2016_16_11.LONGEST.txt
    - salida_eventos_21_12_2016_16_11.RATES.txt
    - salida_eventos_21_12_2016_16_11.txt
  - Modelo de eventos en multicore con 4 procesos:
    - salida_eventos_url4_21_12_2016_16_43.LONGEST.txt
    - salida_eventos_url4_21_12_2016_16_43.RATES.txt
    - salida_eventos_url4_21_12_2016_16_43.txt
  - Modelo de eventos en multicore con 8 procesos:
    - salida_eventos_url8_21_12_18_21.LONGEST.txt
    - salida_eventos_url8_21_12_18_21.RATES.txt
    - salida_eventos_url8_21_12_18_21.txt
  - Modelo de procesos en multicore:
    - salida_procesos_21_12_2016_16_25.LONGEST.txt
    - salida_procesos_21_12_2016_16_25.RATES.txt
    - salida_procesos_21_12_2016_16_25.txt
  - Modelo de threads en multicore:
    - salida_threads_21_12_2016_18_36.LONGEST.txt
    - salida_threads_21_12_2016_18_36.RATES.txt
    - salida_threads_21_12_2016_18_36.txt
- Directorio results/1core:
  - Modelo de eventos en 1 core:
    - salida_eventos_1core_21_12_2016_20_04.LONGEST.txt
    - salida_eventos_1core_21_12_2016_20_04.RATES.txt
    - salida_eventos_1core_21_12_2016_20_04.txt
  - Modelo de procesos en 1 core:
    - salida_procesos_1core_21_12_2016_19_02.LONGEST.txt
    - salida_procesos_1core_21_12_2016_19_02.RATES.txt
    - salida_procesos_1core_21_12_2016_19_02.txt
  - Modelo de threads en 1 core:
    - salida_threads_1core_21_12_2016_18_35.LONGEST.txt
    - salida_threads_1core_21_12_2016_18_35.RATES.txt
    - salida_threads_1core_21_12_2016_18_35.txt

Los ficheros de salida de SAR con los resultados de consumo de CPU son:

- Directorio results/resource:
  - Modelo de eventos en multicore:
    - cpu-eventos.txt
  - Modelo de procesos en multicore:
    - cpu-procesos.txt
  - Modelo de threads en multicore:
    - cpu-threads.txt

Resultados y comentarios

En general, observaremos que los rendimientos se igualan para concurrencia 800, esto es debido a que el cliente en el que se ejecuta Siege no es capaz ya de incrementar el número de conexiones más allá de este punto. Sin embargo, en concurrencia 600 aún podemos ver diferencias entre el rendimiento de cada modelo, por lo que no estamos limitados por el cliente todavía.

Al evaluar el rendimiento de los modelos de procesos y *thread*s, tenemos que tener en cuenta que concurren dos costes que impactan en el rendimiento: Uno es el coste de creación de un proceso/*thread* y otro el coste del cambio de contexto. Si bien en un *thread* ambos son más ligeros puesto que comparten el espacio de memoria, en el caso de los procesos cada proceso tiene su propio espacio de memoria. La gran ventaja de los modelos de procesos/*threads* frente al modelo de eventos es que los primeros escalan de manera automática a tantos cores como tengamos disponibles, mientras que los segundos requieren lanzar varios procesos, aunque su rendimiento global es superior, como veremos en los resultados.

En el Anexo A podemos encontrar un resumen de los ficheros de resultados. Únicamente contienen los valores relativos a transacciones por segundo y a transacción más larga, filtrados mediante grep a partir de los ficheros completos de salida de Siege.

Rendimiento de los 3 modelos (procesos, *threads* y eventos con 8 procesos) en configuración multicore

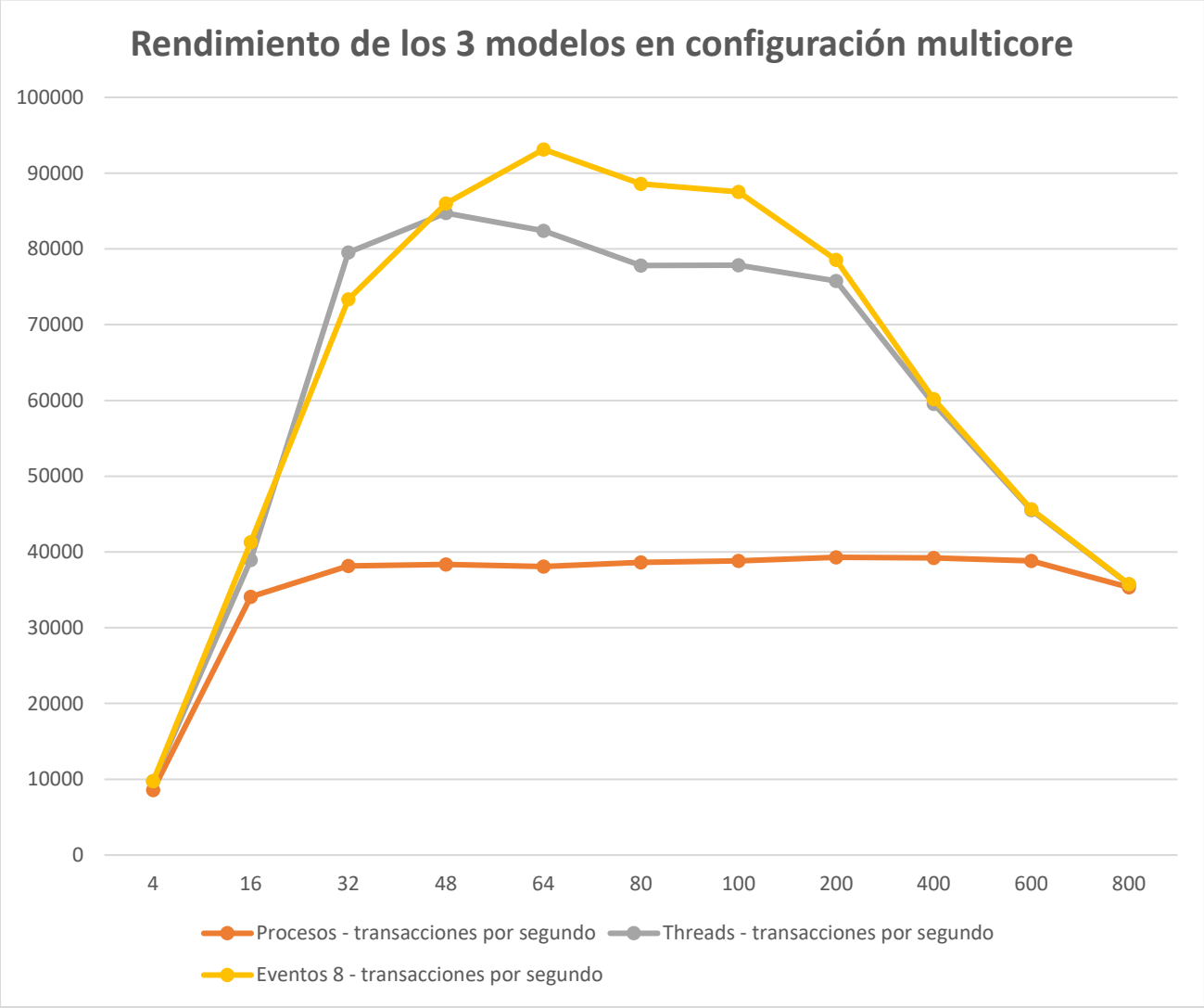

Fig. 12. Gráfico de rendimiento de los 3 modelos en configuración multicore.

En la configuración multicore, podemos observar como el modelo de procesos alcanza un punto a partir de concurrencia 32 donde se aproxima a las 40.000 transacciones/segundo, pero no es capaz de ir más allá, debido a que el coste de la creación de procesos y cambios de contexto limita el rendimiento del modelo.

Sin embargo, el modelo de *threads*, cuyo coste de creación de *threads* y cambio de contexto es inferior al modelo de procesos, escala mucho mejor, es más de dos veces más rápido que el modelo de procesos, ya que alcanza las 85.000 transacciones por segundo aproximadamente.

Finalmente, podemos ver como el modelo de eventos, con 8 procesos en diferentes puertos TCP es el más rápido, superando las 92.000 transacciones por segundo y, además, para concurrencias superiores al modelo de *threads*.

Notemos también que con esta medición obtenemos además el nivel de concurrencia para el que cada modelo es más eficiente. En las pruebas hemos variado la concurrencia del cliente con la concurrencia del servidor fijada a 1000, pero además, también podemos variar la concurrencia del servidor. Así pues, para un escenario de producción, podríamos fijarla en el punto de máxima eficacia para cada uno, según esta gráfica de rendimiento.

Rendimiento del modelo de eventos en multicore: con 1 proceso, 4 procesos y 8 procesos

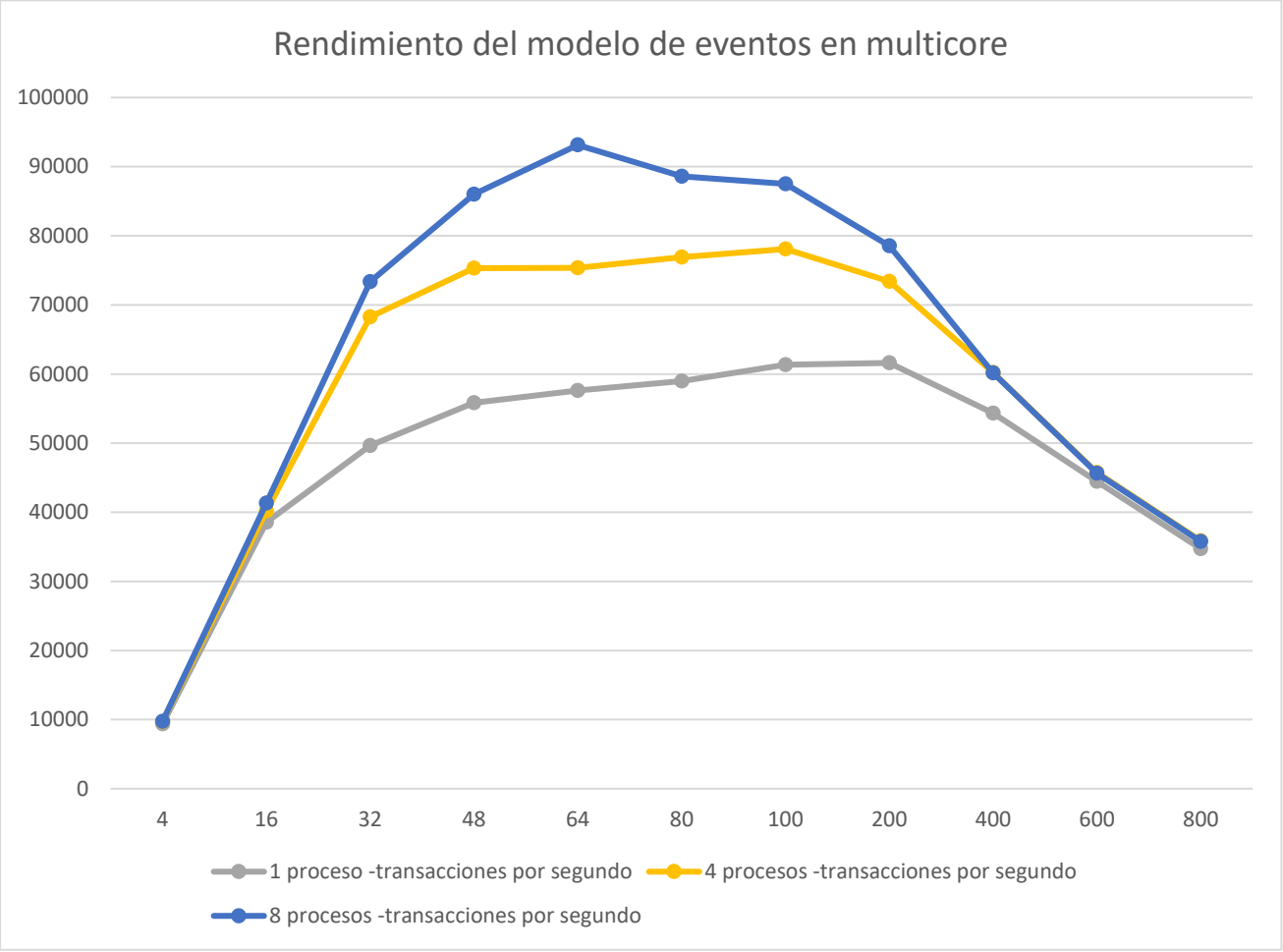

Fig. 13. Gráfico de rendimiento del modelo de eventos en configuración multicore.

Este gráfico ilustra cómo evoluciona el rendimiento del modelo de eventos a medida que aumentamos el número de procesos. Recordemos que el modelo de eventos, por diseño, ejecuta el bucle de eventos en un único *thread*, de forma que sólo es capaz de aprovechar un core. Para superar esta limitación, arrancamos varios procesos del servidor de eventos escuchando en diferentes puertos TCP y configuramos Siege para que lance peticiones a todos ellos, como se indica en el apartado “detalles de las pruebas”. Así, podemos observar como el rendimiento aumenta de manera significativa al pasar de 1 a 4 procesos y, finalmente, hasta 8 procesos. Crear más procesos del servidor a partir de este punto ya no incrementa el rendimiento en el hardware de pruebas empleado, puesto que de nuevo entran en consideración los costes del tiempo de cambio de contexto entre procesos.

Diferencia de rendimiento del modelo de eventos con 1 proceso en configuración 1 core y multicore

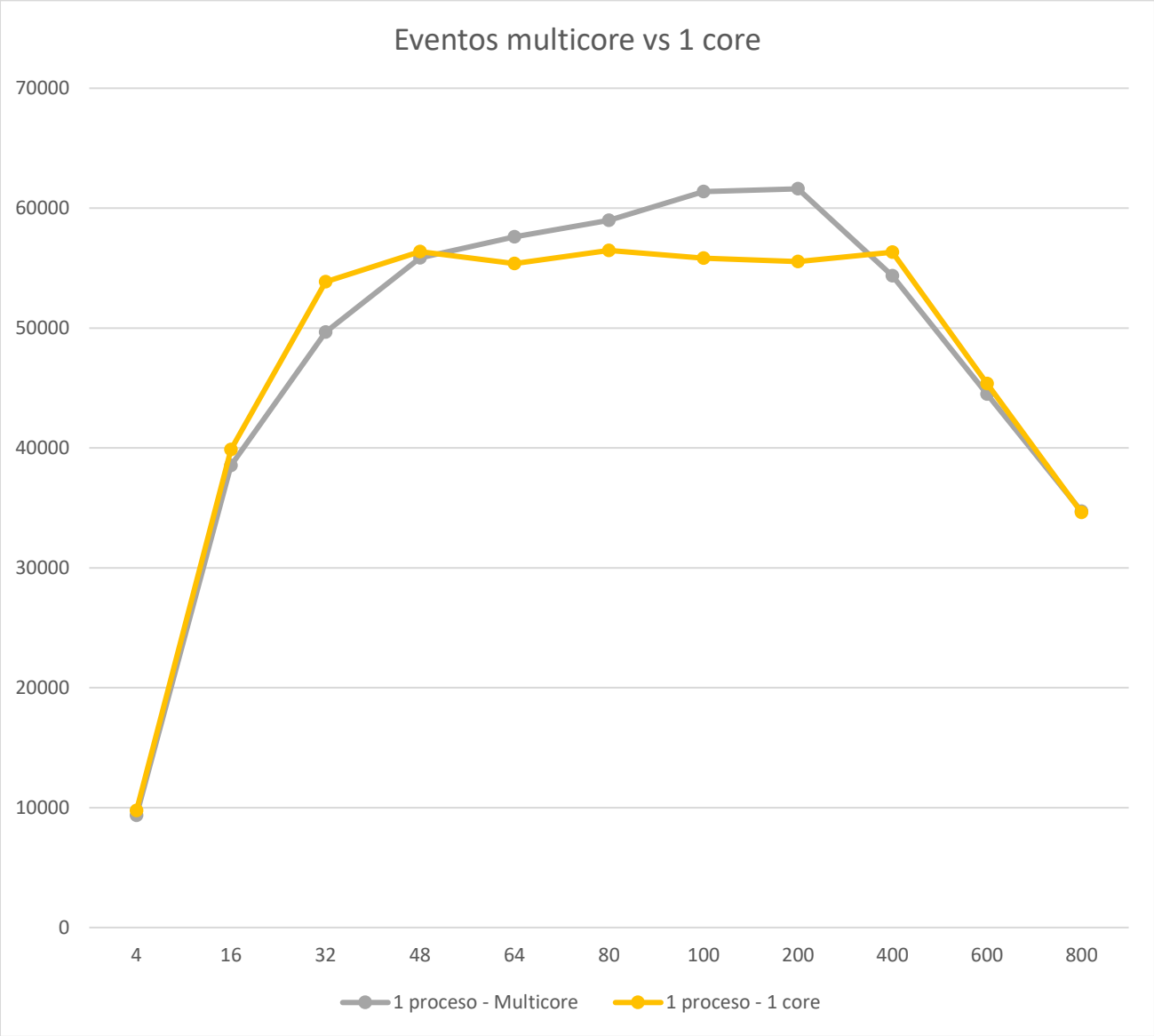

Fig. 14. Gráfico de rendimiento del modelo de eventos en multicore vs 1 core.

Es también interesante considerar el rendimiento del modelo de eventos cuando únicamente tenemos un core y cuando tenemos varios cores disponibles. Como se puede ver en la gráfica, existe una diferencia significativa de rendimiento (aproximadamente un 10%) que se debe a que cuando sólo tenemos un core, éste debe atender todos los procesos del sistema mientras que cuando tenemos varios cores, el sistema puede dedicar un core enteramente al servidor basado en eventos.

Rendimiento de los 3 modelos en configuración 1 core

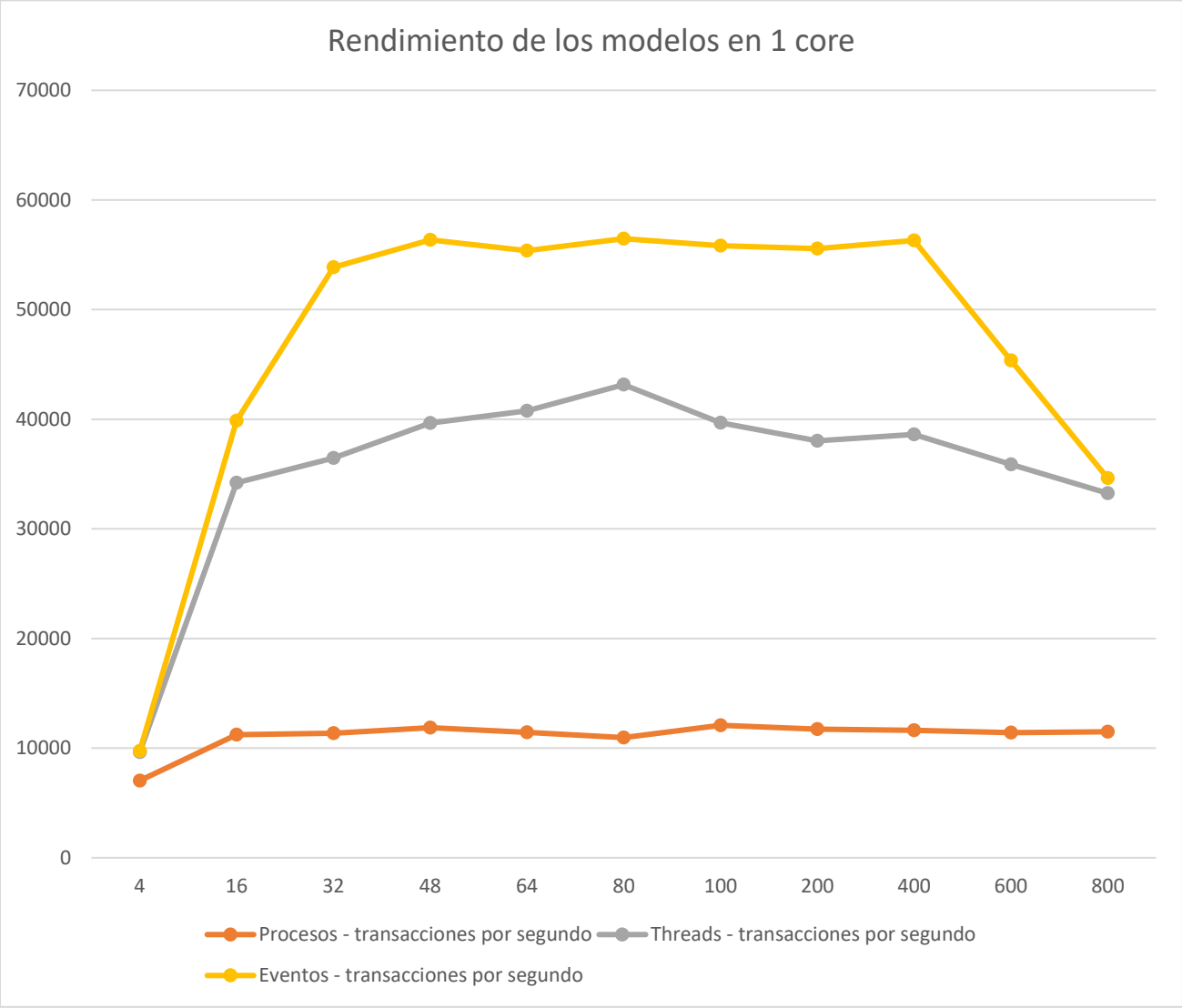

Fig. 15. Gráfico de rendimiento de los 3 modelos en 1 core.

En cuanto a la gráfica de los 3 modelos funcionando en un único core del servidor, podemos apreciar como el del modelo de eventos, que es un modelo que por diseño funciona en un único *thread* es el más eficiente con mucha diferencia, llegando casi a duplicar al de *threads* y a quintuplicar al de procesos. En este modelo, todos los eventos se añaden a una cola que se va procesando en orden en el bucle de eventos dentro del mismo *thread*, sin necesidad de cambios de contexto.

Como el sistema solo tiene un core, sólo puede haber un proceso o un *thread* ejecutándose a la vez. Al tener un único core, en los modelos de procesos y *threads*, cada petición nueva implica obligatoriamente un cambio de contexto, al menos entre el proceso / *thread* principal que acepta las conexiones TCP y los procesos/*threads* secundarios que atienden las peticiones HTTP. En cambio, en el sistema multicore si hay un core libre, sólo impactaría el coste de creación de proceso/*thread* (que se crearía en ese core libre) y no el cambio de contexto.

Así pues, podemos ver como el modelo de procesos es el más lento puesto que crear un proceso o cambiar de contexto entre procesos es muy costoso al tener que considerar tanto el espacio de memoria como el de código. En cambio, el modelo de *threads* es más eficiente que el de procesos puesto que todos los *threads* de un proceso comparten el espacio de memoria y de código, no es por tanto necesario copiarlos al crear el *thread* ni considerarlos en el cambio de contexto de un *thread* a otro.

Tiempo de la transacción más larga en configuración multicore

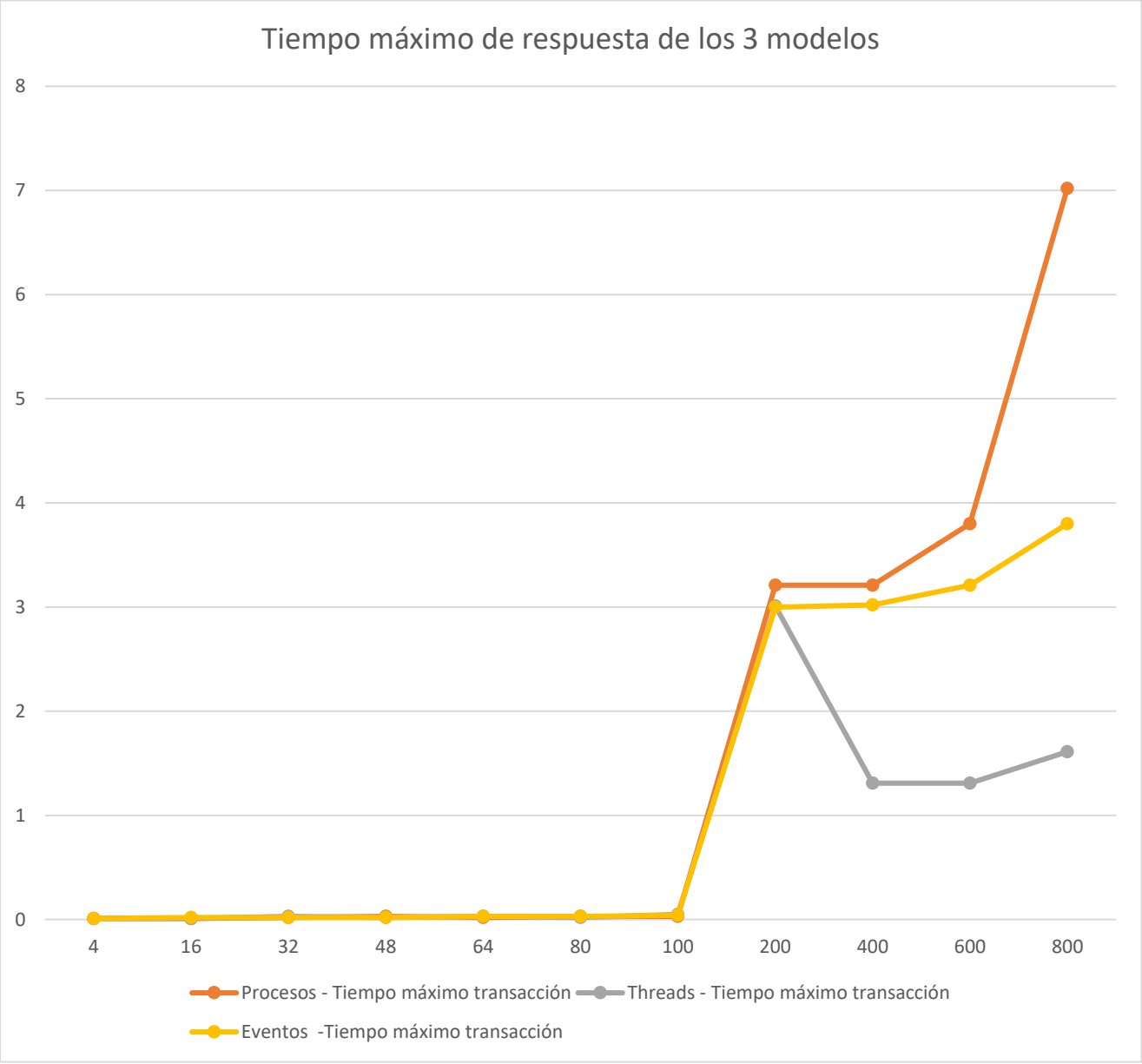

Fig. 16. Gráfico de tiempo de respuesta de los 3 modelos en configuración multicore.

Este gráfico representa las transacciones que más tiempo han tardado en ser respondidas para cada modelo, en general, los tiempos elevados corresponden únicamente a una o dos transacciones de entre los millones de transacciones que responde cada modelo durante las pruebas, no obstante, reflejan el peor caso posible en tiempo de atención de una petición.

En la imagen podemos apreciar como el tiempo máximo para responder una petición se mantiene prácticamente constante en valores muy próximos a 0 hasta alcanzar concurrencia 200, donde ya algunas peticiones necesitan varios segundos para recibir respuesta debido a la elevada carga del servidor. Observamos que el modelo de procesos, que como hemos visto en los otros gráficos es el menos eficiente, es el que tarda más en responder algunas transacciones (hasta 7 segundos en el peor de los casos). El modelo de eventos, si bien se comporta mejor, también presenta algunas transacciones con bastante retraso. Finalmente, el modelo de *threads* es el que, en general, presenta un mejor comportamiento en cuanto a las transacciones que más tardan, no superando en el peor de los casos los 3 segundos y manteniéndose como mucho entre 1 y 2 segundos incluso para los valores máximos de concurrencia.

Uso de CPU de los 3 modelos en configuración multicore con concurrencia 800

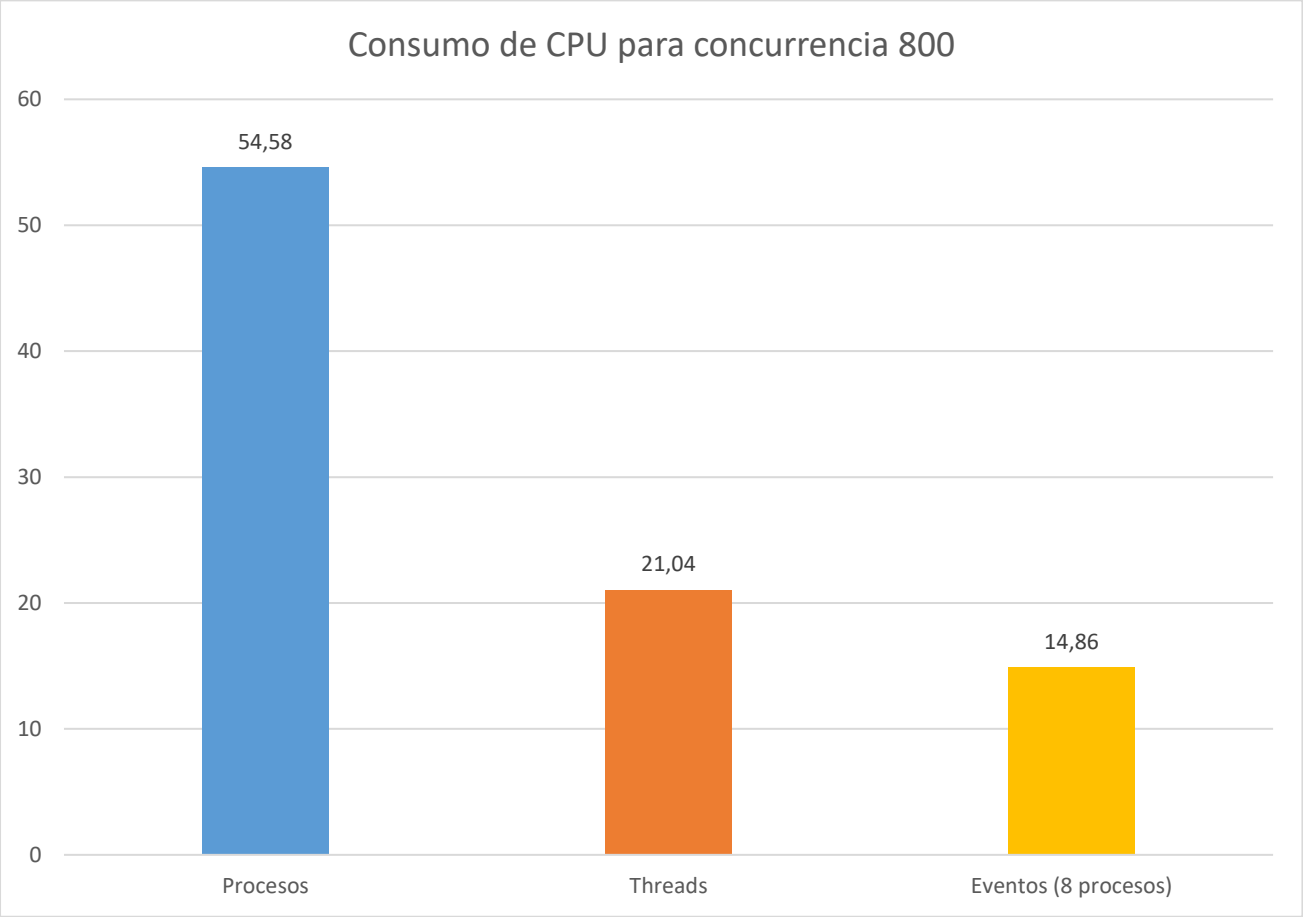

Fig. 17. Consumo de CPU de los 3 modelos en multicore.

Finalmente, tenemos la comparación de uso de CPU para concurrencia 800 que es cuando el número de transacciones se iguala para todos los modelos en un valor de aproximadamente 35.000 transacciones/segundo. Podemos observar como el modelo de procesos es el que más CPU consume con bastante diferencia, con más de un 54%. A continuación, tenemos el modelo de *threads* con 21% y, por último, el modelo de eventos con 8 procesos con únicamente un 15% aproximadamente.

Como vemos, estos valores concuerdan con el rendimiento en transacciones por segundo observado en el primer gráfico de resultados (*Fig. 12. Gráfico de rendimiento de los 3 modelos en configuración multicore*). El modelo de procesos consume más CPU que los demás para el mismo número de transacciones por segundo y en el gráfico de la Fig.12. queda en última posición. A continuación, el modelo de *threads*, en segunda posición en ambos gráficos y, por último, el modelo de eventos, el más rápido en transacciones por segundo y también el que menos CPU consume.

En la Fig. 12. Tanto *threads* como eventos son mucho más rápidos que procesos en transacciones por segundo, pero tienen una diferencia menor entre ellos respecto a la que tienen con procesos, lo que se corresponde con el consumo de CPU que vemos en el gráfico de la Fig. 17.

Conclusiones

En este TFG hemos podido comprobar como la predicción teórica del rendimiento de cada modelo de concurrencia se ha cumplido, siendo el modelo de procesos el más lento debido a su elevado coste de creación y de cambios de contexto. A continuación, el modelo de *threads* es significativamente más rápido que el de procesos y, finalmente, el modelo de eventos el más rápido de todos. Además, los servidores creados son robustos y cumplen todos los requisitos establecidos.

No obstante, en cuanto a facilidad de desarrollo y depuración, el modelo de procesos es el más sencillo de desarrollar, puesto que no hay que tener en cuenta *race conditions* y *deadlocks* al no haber comunicación entre procesos (para el caso del servidor web desarrollado). En segundo lugar, se situaría el modelo de eventos, donde hay que tener en cuenta que en los *callbacks* no puede existir ninguna función bloqueante o que espere una operación susceptible de ser bloqueante y, por último, el modelo de *threads*, donde hay que considerar *mutex* para evitar *race conditions* y *deadlocks*.

Considerando únicamente el rendimiento, como hemos visto, el modelo de eventos es el más rápido en cuanto a transacciones por segundo y también el más eficiente en cuanto a consumo de CPU en el servidor, no obstante, el modelo de *threads* presenta también un buen rendimiento y un consumo de CPU similar. Sin embargo, el modelo de procesos es claramente más lento y mucho menos eficiente en consumo de CPU.

Seguidamente, si consideramos la facilidad para escalar de cada modelo, tanto el modelo de procesos como el de *threads* tienen ventaja respecto al de eventos, ya que los primeros escalan automáticamente al aumentar el número de cores. En cambio, en el caso de eventos hay que dimensionar adecuadamente el número de procesos a medida que aumenta la capacidad del hardware. Si unimos a esto el elevado rendimiento del modelo de *threads*, que casi duplica al de procesos y está muy cerca del de eventos, y, por último, que es el que mejor comportamiento tiene en cuanto a las transacciones que más tardan en responderse, llegamos a la conclusión de que el modelo de *threads* es el que escala mejor y con más facilidad.

Futuras evoluciones

Debido a las restricciones de tiempo, el trabajo abarcado en este TFG es limitado, no obstante, existen una serie de mejoras a considerar para el proyecto, que son:

- Evolución del modelo de *threads*: Si se crea inicialmente un pool de *threads* que posteriormente se vayan asignando a conexiones, se eliminaría el coste de creación de *thread*. De esta forma, sería posible incrementar el rendimiento del modelo de *threads*, a costa de complicar el flujo de ejecución ya que haría falta añadir una estructura intermedia (por ejemplo, una cola) para gestionar las conexiones hasta que se asignen a un *thread* del pool.

- Evolución del modelo de eventos: Sería interesante detectar el número de cores del sistema y levantar automáticamente el número de procesos adecuados de servidor (el doble del número de cores físicos según nuestras pruebas de rendimiento).

- Desarrollo de un balanceador TCP para el modelo de eventos: Un balanceador TCP con un algoritmo de tipo round-robin que se ejecutara en un proceso separado a los servidores, facilitaría el escalado del modelo de eventos, eliminando la necesidad de una pieza externa para hacer esa labor.

- Nuevas características en los servidores web:
    - Ejecución de CGIs: Extender la funcionalidad del servidor web permitiendo que sirva páginas dinámicas generadas por un CGI ejecutado en el servidor.
    - Listado de directorios: Cuando se realice una petición al servidor web referida a una ruta que existe dentro del servidor pero que no tiene index.html, se puede presentar un listado de ficheros que contiene el directorio.

Anexo A – Ficheros resumen de pruebas de rendimiento

Los ficheros incluidos en este anexo contienen únicamente los valores de transaction rate y longest transaction filtrados a partir de los ficheros completos de las pruebas de rendimiento:

salida_eventos_1core_21_12_2016_20_04.LONGEST.txt

Longest transaction: 0.01
Longest transaction: 0.01
Longest transaction: 0.01
Longest transaction: 0.01
Longest transaction: 0.01
Longest transaction: 0.01
Longest transaction: 0.01
Longest transaction: 0.01
Longest transaction: 0.01
Longest transaction: 0.02
Longest transaction: 0.02
Longest transaction: 0.02
Longest transaction: 0.01
Longest transaction: 0.02
Longest transaction: 0.02
Longest transaction: 0.01
Longest transaction: 0.01
Longest transaction: 0.02
Longest transaction: 0.02
Longest transaction: 0.02
Longest transaction: 0.02
Longest transaction: 3.01
Longest transaction: 3.01
Longest transaction: 3.01
Longest transaction: 3.20
Longest transaction: 7.22
Longest transaction: 3.01
Longest transaction: 1.38
Longest transaction: 1.40
Longest transaction: 1.01
Longest transaction: 1.98
Longest transaction: 1.21
Longest transaction: 1.91

salida_procesos_1core_21_12_2016_19_02.LONGEST.txt

Longest transaction: 0.01
Longest transaction: 0.02
Longest transaction: 0.01
Longest transaction: 0.02
Longest transaction: 0.02
Longest transaction: 0.02
Longest transaction: 0.03
Longest transaction: 0.03
Longest transaction: 0.02
Longest transaction: 0.03
Longest transaction: 0.03
Longest transaction: 0.02
Longest transaction: 0.07
Longest transaction: 0.04
Longest transaction: 0.02
Longest transaction: 0.19
Longest transaction: 0.07
Longest transaction: 0.05
Longest transaction: 0.23
Longest transaction: 0.05
Longest transaction: 0.04
Longest transaction: 3.02
Longest transaction: 3.02
Longest transaction: 1.21
Longest transaction: 3.24
Longest transaction: 3.03
Longest transaction: 3.41
Longest transaction: 7.03
Longest transaction: 3.23
Longest transaction: 3.29
Longest transaction: 7.05
Longest transaction: 3.44
Longest transaction: 7.09

salida_eventos_1core_21_12_2016_20_04.RATES.txt

Transaction rate: 9746.10 trans/sec
Transaction rate: 9751.15 trans/sec
Transaction rate: 9755.16 trans/sec
Transaction rate: 39852.55 trans/sec
Transaction rate: 39988.49 trans/sec
Transaction rate: 39753.55 trans/sec
Transaction rate: 53993.39 trans/sec
Transaction rate: 53678.08 trans/sec
Transaction rate: 53940.34 trans/sec
Transaction rate: 57011.61 trans/sec
Transaction rate: 56045.85 trans/sec
Transaction rate: 56037.14 trans/sec
Transaction rate: 55515.92 trans/sec
Transaction rate: 55486.79 trans/sec
Transaction rate: 55125.33 trans/sec
Transaction rate: 56374.28 trans/sec
Transaction rate: 56386.09 trans/sec
Transaction rate: 56659.76 trans/sec
Transaction rate: 56310.81 trans/sec
Transaction rate: 56210.31 trans/sec
Transaction rate: 54968.37 trans/sec
Transaction rate: 56108.91 trans/sec
Transaction rate: 56286.89 trans/sec
Transaction rate: 54266.73 trans/sec
Transaction rate: 56480.18 trans/sec
Transaction rate: 55989.69 trans/sec
Transaction rate: 56492.70 trans/sec
Transaction rate: 45365.23 trans/sec
Transaction rate: 45205.81 trans/sec
Transaction rate: 45559.72 trans/sec
Transaction rate: 34836.52 trans/sec
Transaction rate: 34477.37 trans/sec
Transaction rate: 34554.63 trans/sec

salida_procesos_1core_21_12_2016_19_02.RATES.txt

Transaction rate: 6939.01 trans/sec
Transaction rate: 6805.31 trans/sec
Transaction rate: 7378.38 trans/sec
Transaction rate: 10103.00 trans/sec
Transaction rate: 10887.19 trans/sec
Transaction rate: 12726.13 trans/sec
Transaction rate: 9748.45 trans/sec
Transaction rate: 10986.19 trans/sec
Transaction rate: 13372.87 trans/sec
Transaction rate: 11285.49 trans/sec
Transaction rate: 11093.79 trans/sec
Transaction rate: 13188.89 trans/sec
Transaction rate: 10276.75 trans/sec
Transaction rate: 10833.37 trans/sec
Transaction rate: 13245.19 trans/sec
Transaction rate: 9942.64 trans/sec
Transaction rate: 10777.38 trans/sec
Transaction rate: 12206.51 trans/sec
Transaction rate: 11211.31 trans/sec
Transaction rate: 11612.81 trans/sec
Transaction rate: 13432.57 trans/sec
Transaction rate: 11350.65 trans/sec
Transaction rate: 10933.03 trans/sec
Transaction rate: 12919.42 trans/sec
Transaction rate: 10193.29 trans/sec
Transaction rate: 11201.00 trans/sec
Transaction rate: 13452.61 trans/sec
Transaction rate: 10933.97 trans/sec
Transaction rate: 10854.71 trans/sec
Transaction rate: 12435.81 trans/sec
Transaction rate: 9776.93 trans/sec
Transaction rate: 11352.95 trans/sec
Transaction rate: 13370.21 trans/sec

salida_threads_1core_21_12_2016_18_35.LONGEST.txt

Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.03
Longest transaction:	0.03
Longest transaction:	0.03
Longest transaction:	0.03
Longest transaction:	0.03
Longest transaction:	0.03
Longest transaction:	0.03
Longest transaction:	0.12
Longest transaction:	0.04
Longest transaction:	0.03
Longest transaction:	3.00
Longest transaction:	1.02
Longest transaction:	1.40
Longest transaction:	3.01
Longest transaction:	3.20
Longest transaction:	3.41
Longest transaction:	3.41
Longest transaction:	3.02
Longest transaction:	3.21
Longest transaction:	3.02
Longest transaction:	3.22
Longest transaction:	3.03

salida_eventos_21_12_2016_16_11.LONGEST.txt

Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.02
Longest transaction:	0.01
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.02
Longest transaction:	0.01
Longest transaction:	0.03
Longest transaction:	0.01
Longest transaction:	0.02
Longest transaction:	0.01
Longest transaction:	0.03
Longest transaction:	0.04
Longest transaction:	0.03
Longest transaction:	0.03
Longest transaction:	3.00
Longest transaction:	1.20
Longest transaction:	1.21
Longest transaction:	3.02
Longest transaction:	3.01
Longest transaction:	3.02
Longest transaction:	3.01
Longest transaction:	3.01
Longest transaction:	3.21
Longest transaction:	3.21
Longest transaction:	3.80
Longest transaction:	3.21

salida_threads_1core_21_12_2016_18_35.RATES.txt

Transaction rate:	9619.59 trans/sec
Transaction rate:	9631.60 trans/sec
Transaction rate:	9682.94 trans/sec
Transaction rate:	33754.15 trans/sec
Transaction rate:	34390.78 trans/sec
Transaction rate:	34478.54 trans/sec
Transaction rate:	33261.69 trans/sec
Transaction rate:	39320.48 trans/sec
Transaction rate:	36809.24 trans/sec
Transaction rate:	44852.14 trans/sec
Transaction rate:	37475.13 trans/sec
Transaction rate:	36630.03 trans/sec
Transaction rate:	36512.95 trans/sec
Transaction rate:	39280.89 trans/sec
Transaction rate:	46483.59 trans/sec
Transaction rate:	43299.46 trans/sec
Transaction rate:	46107.43 trans/sec
Transaction rate:	40054.25 trans/sec
Transaction rate:	37114.41 trans/sec
Transaction rate:	39144.71 trans/sec
Transaction rate:	42791.64 trans/sec
Transaction rate:	36345.13 trans/sec
Transaction rate:	41760.79 trans/sec
Transaction rate:	35984.18 trans/sec
Transaction rate:	35489.43 trans/sec
Transaction rate:	39423.40 trans/sec
Transaction rate:	40960.90 trans/sec
Transaction rate:	32267.95 trans/sec
Transaction rate:	34066.50 trans/sec
Transaction rate:	41281.40 trans/sec
Transaction rate:	34183.19 trans/sec
Transaction rate:	32855.39 trans/sec
Transaction rate:	32720.00 trans/sec

salida_eventos_21_12_2016_16_11.RATES.txt

Transaction rate:	9392.21 trans/sec
Transaction rate:	9354.96 trans/sec
Transaction rate:	9345.04 trans/sec
Transaction rate:	38964.27 trans/sec
Transaction rate:	38096.50 trans/sec
Transaction rate:	38538.44 trans/sec
Transaction rate:	49295.30 trans/sec
Transaction rate:	49333.84 trans/sec
Transaction rate:	50319.32 trans/sec
Transaction rate:	53910.31 trans/sec
Transaction rate:	55743.54 trans/sec
Transaction rate:	57869.87 trans/sec
Transaction rate:	57426.33 trans/sec
Transaction rate:	55401.50 trans/sec
Transaction rate:	59999.00 trans/sec
Transaction rate:	59734.94 trans/sec
Transaction rate:	57216.42 trans/sec
Transaction rate:	60030.93 trans/sec
Transaction rate:	60786.49 trans/sec
Transaction rate:	60449.75 trans/sec
Transaction rate:	62871.57 trans/sec
Transaction rate:	61346.55 trans/sec
Transaction rate:	61486.89 trans/sec
Transaction rate:	61996.70 trans/sec
Transaction rate:	54606.81 trans/sec
Transaction rate:	54150.35 trans/sec
Transaction rate:	54287.49 trans/sec
Transaction rate:	45878.86 trans/sec
Transaction rate:	45804.61 trans/sec
Transaction rate:	41780.66 trans/sec
Transaction rate:	34512.84 trans/sec
Transaction rate:	34471.34 trans/sec
Transaction rate:	35193.48 trans/sec

salida_eventos_url4_21_12_2016_16_43.LONGEST.txt

Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.03
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.03
Longest transaction:	0.04
Longest transaction:	0.03
Longest transaction:	0.02
Longest transaction:	0.03
Longest transaction:	0.04
Longest transaction:	3.00
Longest transaction:	1.20
Longest transaction:	1.01
Longest transaction:	1.01
Longest transaction:	0.76
Longest transaction:	1.06
Longest transaction:	1.04
Longest transaction:	1.09
Longest transaction:	1.04
Longest transaction:	1.75
Longest transaction:	1.49
Longest transaction:	1.50

salida_eventos_url8_21_12_18_21.LONGEST.txt

Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.01
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.01
Longest transaction:	0.03
Longest transaction:	0.02
Longest transaction:	0.03
Longest transaction:	0.06
Longest transaction:	0.05
Longest transaction:	0.05
Longest transaction:	0.07
Longest transaction:	0.07
Longest transaction:	0.04
Longest transaction:	0.27
Longest transaction:	0.43
Longest transaction:	0.41
Longest transaction:	0.68
Longest transaction:	0.41
Longest transaction:	0.50
Longest transaction:	1.17
Longest transaction:	1.27
Longest transaction:	1.08
Longest transaction:	1.66
Longest transaction:	1.50
Longest transaction:	1.64

salida_eventos_url4_21_12_2016_16_43.RATES.txt

Transaction rate:	9716.24 trans/sec
Transaction rate:	9687.49 trans/sec
Transaction rate:	9684.38 trans/sec
Transaction rate:	41163.46 trans/sec
Transaction rate:	41414.11 trans/sec
Transaction rate:	38015.02 trans/sec
Transaction rate:	70029.13 trans/sec
Transaction rate:	68144.24 trans/sec
Transaction rate:	66667.67 trans/sec
Transaction rate:	77423.83 trans/sec
Transaction rate:	75167.57 trans/sec
Transaction rate:	73301.00 trans/sec
Transaction rate:	74654.86 trans/sec
Transaction rate:	76407.11 trans/sec
Transaction rate:	74970.38 trans/sec
Transaction rate:	76127.73 trans/sec
Transaction rate:	77985.19 trans/sec
Transaction rate:	76625.02 trans/sec
Transaction rate:	76939.54 trans/sec
Transaction rate:	77645.15 trans/sec
Transaction rate:	79693.59 trans/sec
Transaction rate:	71879.08 trans/sec
Transaction rate:	73959.62 trans/sec
Transaction rate:	74385.19 trans/sec
Transaction rate:	60736.64 trans/sec
Transaction rate:	59997.00 trans/sec
Transaction rate:	59917.82 trans/sec
Transaction rate:	46111.31 trans/sec
Transaction rate:	45322.35 trans/sec
Transaction rate:	45812.83 trans/sec
Transaction rate:	35942.89 trans/sec
Transaction rate:	35780.16 trans/sec
Transaction rate:	35920.04 trans/sec

salida_eventos_url8_21_12_18_21.RATES.txt

Transaction rate:	9759.81 trans/sec
Transaction rate:	9758.06 trans/sec
Transaction rate:	9760.26 trans/sec
Transaction rate:	42395.90 trans/sec
Transaction rate:	39193.20 trans/sec
Transaction rate:	42348.15 trans/sec
Transaction rate:	74133.34 trans/sec
Transaction rate:	75159.46 trans/sec
Transaction rate:	70751.85 trans/sec
Transaction rate:	87862.06 trans/sec
Transaction rate:	83754.26 trans/sec
Transaction rate:	86417.62 trans/sec
Transaction rate:	92482.78 trans/sec
Transaction rate:	93917.02 trans/sec
Transaction rate:	93009.91 trans/sec
Transaction rate:	88873.17 trans/sec
Transaction rate:	88161.66 trans/sec
Transaction rate:	88744.15 trans/sec
Transaction rate:	87663.47 trans/sec
Transaction rate:	87304.11 trans/sec
Transaction rate:	87584.48 trans/sec
Transaction rate:	78922.20 trans/sec
Transaction rate:	78116.62 trans/sec
Transaction rate:	78621.75 trans/sec
Transaction rate:	60439.14 trans/sec
Transaction rate:	60039.74 trans/sec
Transaction rate:	60042.14 trans/sec
Transaction rate:	45607.41 trans/sec
Transaction rate:	45715.23 trans/sec
Transaction rate:	45644.09 trans/sec
Transaction rate:	35883.57 trans/sec
Transaction rate:	35738.78 trans/sec
Transaction rate:	35671.14 trans/sec

salida_procesos_21_12_2016_16_25.LONGEST.txt

Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.03
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.03
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.03
Longest transaction:	1.20
Longest transaction:	3.21
Longest transaction:	3.00
Longest transaction:	3.01
Longest transaction:	3.02
Longest transaction:	3.21
Longest transaction:	3.80
Longest transaction:	3.21
Longest transaction:	3.02
Longest transaction:	7.02
Longest transaction:	3.41
Longest transaction:	3.21

salida_threads_21_12_2016_18_36.LONGEST.txt

Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.01
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.03
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.03
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.02
Longest transaction:	0.05
Longest transaction:	0.03
Longest transaction:	3.01
Longest transaction:	3.01
Longest transaction:	1.20
Longest transaction:	1.31
Longest transaction:	0.59
Longest transaction:	0.63
Longest transaction:	1.31
Longest transaction:	0.56
Longest transaction:	1.24
Longest transaction:	1.31
Longest transaction:	1.56
Longest transaction:	1.61

salida_procesos_21_12_2016_16_25.RATES.txt

Transaction rate:	8645.63 trans/sec
Transaction rate:	8586.99 trans/sec
Transaction rate:	8550.15 trans/sec
Transaction rate:	34160.66 trans/sec
Transaction rate:	33542.44 trans/sec
Transaction rate:	34510.51 trans/sec
Transaction rate:	37505.41 trans/sec
Transaction rate:	37550.05 trans/sec
Transaction rate:	39432.03 trans/sec
Transaction rate:	38319.82 trans/sec
Transaction rate:	37427.03 trans/sec
Transaction rate:	39344.45 trans/sec
Transaction rate:	37668.47 trans/sec
Transaction rate:	36973.68 trans/sec
Transaction rate:	39629.33 trans/sec
Transaction rate:	38159.56 trans/sec
Transaction rate:	37523.15 trans/sec
Transaction rate:	40161.96 trans/sec
Transaction rate:	38286.49 trans/sec
Transaction rate:	37842.74 trans/sec
Transaction rate:	40331.53 trans/sec
Transaction rate:	38926.93 trans/sec
Transaction rate:	38286.99 trans/sec
Transaction rate:	40675.38 trans/sec
Transaction rate:	39204.71 trans/sec
Transaction rate:	38333.67 trans/sec
Transaction rate:	40151.30 trans/sec
Transaction rate:	38385.77 trans/sec
Transaction rate:	38480.46 trans/sec
Transaction rate:	39662.83 trans/sec
Transaction rate:	35138.71 trans/sec
Transaction rate:	34960.82 trans/sec
Transaction rate:	35954.81 trans/sec

salida_threads_21_12_2016_18_36.RATES.txt

Transaction rate:	9757.76 trans/sec
Transaction rate:	9750.05 trans/sec
Transaction rate:	9754.86 trans/sec
Transaction rate:	38950.95 trans/sec
Transaction rate:	38957.66 trans/sec
Transaction rate:	38926.93 trans/sec
Transaction rate:	79123.02 trans/sec
Transaction rate:	79541.34 trans/sec
Transaction rate:	79931.03 trans/sec
Transaction rate:	84820.22 trans/sec
Transaction rate:	84725.62 trans/sec
Transaction rate:	84640.84 trans/sec
Transaction rate:	84397.70 trans/sec
Transaction rate:	82397.20 trans/sec
Transaction rate:	80434.54 trans/sec
Transaction rate:	79023.92 trans/sec
Transaction rate:	77453.66 trans/sec
Transaction rate:	76903.30 trans/sec
Transaction rate:	78302.30 trans/sec
Transaction rate:	77633.63 trans/sec
Transaction rate:	77671.27 trans/sec
Transaction rate:	75815.41 trans/sec
Transaction rate:	75681.57 trans/sec
Transaction rate:	75831.37 trans/sec
Transaction rate:	59951.55 trans/sec
Transaction rate:	59161.26 trans/sec
Transaction rate:	59594.70 trans/sec
Transaction rate:	45521.42 trans/sec
Transaction rate:	45423.75 trans/sec
Transaction rate:	45650.50 trans/sec
Transaction rate:	35645.39 trans/sec
Transaction rate:	35784.27 trans/sec
Transaction rate:	35792.39 trans/sec

Los siguientes ficheros contienen los resultados de las pruebas de consumo de CPU de los diferentes modelos:

cpu-eventos.txt

Linux 4.4.0-21-generic (kraken) 12/21/2016 _x86_64_ (8 CPU)

10:51:48 PM	CPU	%user	%nice	%system	%iowait	%steal	%idle
10:51:49 PM	all	0,25	0,00	0,75	0,13	0,00	98,87
10:51:50 PM	all	2,28	0,00	9,64	0,00	0,00	88,07
10:51:51 PM	all	2,29	0,00	10,04	0,00	0,00	87,67
10:51:52 PM	all	2,30	0,00	9,58	0,26	0,00	87,87
10:51:53 PM	all	2,77	0,00	10,33	0,13	0,00	86,78
10:51:54 PM	all	2,42	0,00	10,18	0,00	0,00	87,40
10:51:55 PM	all	2,80	0,00	9,91	0,00	0,00	87,29
10:51:56 PM	all	2,67	0,00	9,92	0,00	0,00	87,40
10:51:57 PM	all	3,13	0,00	10,76	0,00	0,00	86,11
10:51:58 PM	all	8,29	0,00	9,82	0,51	0,00	81,38
10:51:59 PM	all	16,29	0,00	12,03	0,00	0,00	71,68
10:52:00 PM	all	15,72	0,00	11,41	0,00	0,00	72,88
10:52:01 PM	all	14,16	0,00	11,00	0,00	0,00	74,84
10:52:02 PM	all	2,39	0,00	11,31	0,00	0,00	86,31
10:52:03 PM	all	2,14	0,00	11,22	0,38	0,00	86,25
10:52:04 PM	all	2,76	0,00	11,03	0,00	0,00	86,22
10:52:05 PM	all	2,65	0,00	10,48	0,00	0,00	86,87
10:52:06 PM	all	2,66	0,00	10,38	0,00	0,00	86,96
10:52:07 PM	all	2,89	0,00	10,80	0,00	0,00	86,31
10:52:08 PM	all	2,54	0,00	9,91	0,38	0,00	87,17
10:52:09 PM	all	2,66	0,00	10,25	0,00	0,00	87,09
10:52:10 PM	all	2,92	0,00	10,14	0,00	0,00	86,95
10:52:11 PM	all	2,64	0,00	10,71	0,00	0,00	86,65
10:52:12 PM	all	2,17	0,00	10,08	0,00	0,00	87,76
10:52:13 PM	all	9,42	0,00	12,19	0,25	0,00	78,14
10:52:14 PM	all	16,29	0,00	11,11	0,00	0,00	72,60
10:52:15 PM	all	16,06	0,00	11,38	0,00	0,00	72,57
10:52:16 PM	all	12,55	0,00	11,17	0,00	0,00	76,29
10:52:17 PM	all	2,79	0,00	10,03	0,00	0,00	87,18
10:52:18 PM	all	3,02	0,00	10,45	0,38	0,00	86,15
10:52:19 PM	all	0,50	0,00	0,99	0,00	0,00	98,51
10:52:20 PM	all	0,00	0,00	0,12	0,00	0,00	99,88
Average:	all	5,14	0,00	9,65	0,07	0,00	85,14

cpu-procesos.txt

Linux 4.4.0-21-generic (kraken) 12/21/2016 _x86_64_ (8 CPU)

10:49:27 PM	CPU	%user	%nice	%system	%iowait	%steal	%idle
10:49:28 PM	all	5,90	0,00	9,66	0,00	0,00	84,44
10:49:29 PM	all	19,25	0,00	46,25	0,00	0,00	34,50
10:49:30 PM	all	20,22	0,00	53,76	0,12	0,00	25,89
10:49:31 PM	all	15,91	0,00	45,58	0,00	0,00	38,51
10:49:32 PM	all	7,81	0,00	48,33	0,12	0,00	43,74
10:49:33 PM	all	7,68	0,00	48,11	0,00	0,00	44,21
10:49:34 PM	all	7,79	0,00	47,77	0,13	0,00	44,32
10:49:35 PM	all	7,05	0,00	48,11	0,00	0,00	44,84
10:49:36 PM	all	7,40	0,00	47,58	0,00	0,00	45,03
10:49:37 PM	all	7,55	0,00	48,05	0,13	0,00	44,28
10:49:38 PM	all	7,07	0,00	48,74	0,00	0,00	44,19
10:49:39 PM	all	7,14	0,00	48,50	0,00	0,00	44,36
10:49:40 PM	all	8,55	0,00	47,30	0,00	0,00	44,15
10:49:41 PM	all	7,62	0,00	48,25	0,00	0,00	44,12
10:49:42 PM	all	7,35	0,00	48,80	0,13	0,00	43,73
10:49:43 PM	all	9,98	0,00	47,88	0,12	0,00	42,02
10:49:44 PM	all	20,75	0,00	43,60	0,00	0,00	35,65
10:49:45 PM	all	19,14	0,00	44,71	0,13	0,00	36,02
10:49:46 PM	all	20,25	0,00	52,70	0,13	0,00	26,92
10:49:47 PM	all	11,82	0,00	51,45	0,00	0,00	36,73
10:49:48 PM	all	8,13	0,00	48,92	0,25	0,00	42,69
10:49:49 PM	all	7,46	0,00	47,79	0,00	0,00	44,75
10:49:50 PM	all	8,38	0,00	47,38	0,00	0,00	44,25
10:49:51 PM	all	6,75	0,00	49,25	0,00	0,00	44,00
10:49:52 PM	all	7,85	0,00	48,07	0,00	0,00	44,08
10:49:53 PM	all	7,68	0,00	47,98	0,25	0,00	44,08
10:49:54 PM	all	7,72	0,00	48,32	0,00	0,00	43,96
10:49:55 PM	all	6,95	0,00	48,67	0,00	0,00	44,37
10:49:56 PM	all	6,50	0,00	49,38	0,00	0,00	44,12
10:49:57 PM	all	7,48	0,00	48,38	0,00	0,00	44,14
10:49:58 PM	all	5,62	0,00	13,88	0,12	0,00	80,38
10:49:59 PM	all	14,34	0,00	0,62	0,25	0,00	84,79
Average:	all	10,05	0,00	44,48	0,06	0,00	45,42

cpu-threads.txt

Linux 4.4.0-21-generic (kraken) 12/21/2016 _x86_64_ (8 CPU)

10:50:28 PM	CPU	%user	%nice	%system	%iowait	%steal	%idle
10:50:29 PM	all	14,20	0,00	5,28	0,13	0,00	80,40
10:50:30 PM	all	16,82	0,00	13,45	0,00	0,00	69,73
10:50:31 PM	all	16,88	0,00	14,47	0,00	0,00	68,65
10:50:32 PM	all	5,31	0,00	13,91	0,13	0,00	80,66
10:50:33 PM	all	4,48	0,00	15,67	0,00	0,00	79,85
10:50:34 PM	all	3,56	0,00	14,87	0,38	0,00	81,19
10:50:35 PM	all	4,30	0,00	14,56	0,00	0,00	81,14
10:50:36 PM	all	4,28	0,00	15,09	0,00	0,00	80,63
10:50:37 PM	all	3,81	0,00	15,23	0,00	0,00	80,96
10:50:38 PM	all	3,72	0,00	13,99	0,00	0,00	82,28
10:50:39 PM	all	3,56	0,00	15,01	0,00	0,00	81,42
10:50:40 PM	all	4,39	0,00	15,29	0,38	0,00	79,95
10:50:41 PM	all	3,33	0,00	14,62	0,00	0,00	82,05
10:50:42 PM	all	4,06	0,00	14,47	0,00	0,00	81,47
10:50:43 PM	all	10,68	0,00	16,46	0,00	0,00	72,86
10:50:44 PM	all	16,60	0,00	16,60	0,00	0,00	66,79
10:50:45 PM	all	16,97	0,00	15,30	0,26	0,00	67,48
10:50:46 PM	all	12,77	0,00	15,14	0,00	0,00	72,09
10:50:47 PM	all	4,49	0,00	15,73	0,00	0,00	79,78
10:50:48 PM	all	3,93	0,00	14,83	0,13	0,00	81,12
10:50:49 PM	all	3,96	0,00	14,18	0,00	0,00	81,86
10:50:50 PM	all	4,33	0,00	14,25	0,25	0,00	81,17
10:50:51 PM	all	4,35	0,00	13,83	0,00	0,00	81,82
10:50:52 PM	all	3,64	0,00	15,93	0,00	0,00	80,43
10:50:53 PM	all	4,80	0,00	14,52	0,00	0,00	80,68
10:50:54 PM	all	4,58	0,00	16,23	0,00	0,00	79,18
10:50:55 PM	all	4,65	0,00	14,95	0,38	0,00	80,03
10:50:56 PM	all	3,97	0,00	14,21	0,00	0,00	81,82
10:50:57 PM	all	4,09	0,00	14,05	0,00	0,00	81,86
10:50:58 PM	all	11,63	0,00	14,92	0,00	0,00	73,45
10:50:59 PM	all	12,91	0,00	0,38	0,00	0,00	86,72
10:51:00 PM	all	12,73	0,00	0,00	0,50	0,00	86,77
Average:	all	7,30	0,00	13,66	0,08	0,00	78,96

Anexo B – Ficheros de configuración Siege para modelo de eventos

urls4.txt

```
http://192.168.1.34:1080/
http://192.168.1.34:1081/
http://192.168.1.34:1082/
http://192.168.1.34:1083/
```

urls8.txt

```
http://192.168.1.34:1080/
http://192.168.1.34:1081/
http://192.168.1.34:1082/
http://192.168.1.34:1083/
http://192.168.1.34:1084/
http://192.168.1.34:1085/
http://192.168.1.34:1086/
http://192.168.1.34:1087/
```

Bibliografía

[1] Free of charge. Varios Autores. *Parallelism vs. Concurrency.* https://wiki.haskell.org/Parallelism_vs._Concurrency [En línea] Consultado a 2/10/2016.

[2] Creative Commons. Varios Autores. *Deadlock.* https://en.wikipedia.org/wiki/Deadlock [En línea] Consultado a 2/10/2016.

[3] Creative Commons. Varios Autores. *Race Condition.* https://en.wikipedia.org/wiki/Race_condition#Software [En línea] Consultado a 2/10/2016.

[4] © Hoare, C. A. R. (1978). *"Communicating sequential processes".* Communications of the ACM. http://www.cs.ucf.edu/courses/cop4020/sum2009/CSP-hoare.pdf [En línea] Consultado a 2/10/2016.

[5] © 2008 Intel Corporation. Matt Gillespie. *Amdahl's Law, Gustafson's Trend, and the Performance Limits of Parallel Applications* https://software.intel.com/sites/default/files/m/d/4/1/d/8/Gillespie-0053-AAD_Gustafson-Amdahl_v1__2_.rh.final.pdf [En línea] Consultado a 4/10/2016.

[6] © 2016 Mark Mitchell y otros – CodeSourcery LLC. ISBN 0-7357-1043-0. *Advanced Linux Programming. Chapter 4- Threads*. http://advancedlinuxprogramming.com/alp-folder/alp-ch04-threads.pdf[En línea] Consultado a 4/10/2016

[7] © 2010. University of Pennsylvania. Computer Science Lectures. Curso CIT 595 2010. https://www.seas.upenn.edu/~cit595/cit595s10/lectures/processvsthreads.pdf [En línea] Consultado a 5/10/2016

[8] © Hakim Weatherspoon. Cornell University. *Concurrency, Threads, and Events.* http://www.cs.cornell.edu/courses/cs6410/2010fa/lectures/03-concurrency.pdf [En línea] Consultado a 6/10/2016

[9] ©2016 Salesforce.com. *Optimizing Node.js Application Concurrency.* https://devcenter.heroku.com/articles/node-concurrency [En línea] Consultado a 6/10/2016

[10] © 2015 - KSI Graduate School. C.Y. Hsieh *Introduction to Computer Networks* http://pluto.ksi.edu/~cyh/cis370/ebook/ch05b.htm [En línea] Consultado a 7/10/2016

[11] © 2005-2016 Mozilla Developer Network. Varios Autores. *HTTP* https://developer.mozilla.org/en-US/docs/Web/HTTP [En línea] Consultado a 7/10/2016

[12] © 2016 W3C. *What is HyperText* https://www.w3.org/WhatIs.html [En línea] Consultado a 7/10/2016

[13] © The Internet Society (2005) T. Berners-Lee y otros. *Uniform Resource Identifier (URI): Generic Syntax* https://www.ietf.org/rfc/rfc3986.txt [En línea] Consultado a 9/10/2016

[14] 1996. T. Berners-Lee y otros. *Hypertext Transfer Protocol -- HTTP/1.0* https://tools.ietf.org/html/rfc1945 [En línea] Consultado a 9/10/2016

[15] 1999. R.Fielding y otros. *Hypertext Transfer Protocol -- HTTP/1.1* https://tools.ietf.org/html/rfc2616#page-170 [En línea] Consultado a 9/10/2016

[16] Balachander Krishnamurthy y otros. *Key Differences between HTTP/1.0 and HTTP/1.*1 http://www8.org/w8-papers/5c-protocols/key/key.html [En línea] Consultado a 9/10/2016

[17] Creative Commons. *SPDY: An experimental protocol for a faster web* https://www.chromium.org/spdy/spdy-whitepaper [En línea] Consultado a 10/10/2016

[18] Creative Commons. *Transitioning from SPDY to HTTP/2.* https://blog.chromium.org/2016/02/transitioning-from-spdy-to-http2.html [En línea] Consultado a 10/10/2016

[19] © 2015 IETF Trust. M.Belshe y otros. *Hypertext Transfer Protocol Version 2 (HTTP/2)* https://tools.ietf.org/html/rfc7540 [En línea] Consultado a 10/10/2016

[20] IETF HTTP Working Group. *HTTP/2 Frequently Asked Questions.* https://http2.github.io/faq/ [En línea] Consultado a 10/10/2016

[21] © Akamai. *Turn-on HTTP/2 today!* https://http2.akamai.com/ [En línea] Consultado a 10/10/2016

[22] © 2015 IETF Trust. R.Peon y otros. *HPACK: Header Compression for HTTP/2.* https://tools.ietf.org/html/rfc7541 [En línea] Consultado a 10/10/2016

[23] Creative Commons. Varios Autores. *Software Design Description.* https://en.wikipedia.org/wiki/Software_design_description [En línea] Consultado a 15/10/2016

[24] © Copyright 2010 - 2016 Toptal, LLC. Chris Fox. *Why Design Documents Matter.* https://www.toptal.com/freelance/why-design-documents-matter [En línea] Consultado a 15/10/2016

[25] © 2004. Donald Bell. *The sequence diagram.* http://www.ibm.com/developerworks/rational/library/3101.html [En línea] Consultado a 16/10/2016

[26] © 2003-2014 Scott W. Ambler. *UML 2 Sequence Diagrams: An Agile Introduction* . http://agilemodeling.com/artifacts/sequenceDiagram.htm [En línea] Consultado a 16/10/2016

[27] © 2003-2014 Scott W. Ambler. *UML 2 Activity Diagramming Guidelines* http://www.agilemodeling.com/style/activityDiagram.htm [En línea] Consultado a 17/10/2016

[28] © 2009-2016 uml-diagrams.org *UML Activity Diagrams Reference* http://www.uml-diagrams.org/activity-diagrams-reference.html [En línea] Consultado a 17/10/2016

[29] © W3C 1995. *The Common Logfile Format* https://www.w3.org/Daemon/User/Config/Logging.html#common-logfile-format [En línea] Consultado a 28/10/2016

[30] Creative Commons. Varios Autores. *Fork (system call)* https://en.wikipedia.org/wiki/Fork_(system_call) [En línea] Consultado a 1/11/2016

[31] © 1996 Institute of Electrical and Electronics Engineers, Inc. *IEEE Standards Interpretations for IEEE Std 1003.1c™-1995 IEEE Standard for Information Technology--Portable Operating System Interface (POSIX®) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)* http://standards.ieee.org/findstds/interps/1003-1c-95_int/ [En línea] Consultado a 1/11/2016

[32] © 2016 Free Software Foundation, Inc. *POSIX Threads* https://www.gnu.org/software/libc/manual/html_mono/libc.html#POSIX-Threads [En línea] Consultado a 1/11/2016

[33] © Nick Mathewson and Niels Provos. *libevent – an event notification library* http://libevent.org/ [En línea] Consultado a 2/11/2016

[34] Creative Commons. Varios Autores. *errno.h* https://en.wikipedia.org/wiki/Errno.h [En línea] Consultado a 4/11/2016

[35] © 2015 Jeffrey Fulmer. *Siege Home.* https://www.joedog.org/siege-home/ [en línea] Consultado a 19/12/2016

[36] © 2015 By Jeffrey Fulmer. *How Does Siege Calculate Concurrency?* https://www.joedog.org/siege-faq/#a17a [en línea] Consultado a 19/12/2016

[37] GPL. Sebastien Godard. *Welcome to the Sysstat utilites home page.* http://sebastien.godard.pagesperso-orange.fr/ [En línea] Consultado a 21/12/2016

[38] © Copyright IBM Corporation 1994, 2016. *Easy system monitoring with SAR* http://www.ibm.com/developerworks/aix/library/au-unix-perfmonsar.html [En línea] Consultado a 21/12/2016