Aplicación de la simulación en tiempo real para mejorar la calidad de servicio del Middleware

Ramon Nou Castell

Director: Jordi Torres Viñals

Tesis propuesta para la obtención del doctorado en informática

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Departamento de Arquitectura de Computadores
Universidad Politécnica de Catalunya
Barcelona, España
2008

Dedicado a Natalia Fos.

Agradecimientos

Tras algún intento de colaboración con el grupo eDragon que empezó en una “aula de empresa” dedicada al CEPBA (seguramente hacia el 2002), realizar mi proyecto final de carrera bajo la dirección de Jordi Torres y un amago de desvinculación de la universidad (de pocos meses) trabajando fuera, volví a la universidad, mientras seguía trabajando, para empezar el doctorado. No exagero si digo que una de las principales razones fue poder participar de nuevo en el concurso de programación de la ACM/IBM. Sin embargo, la fortuna no me sonrió: ese año introdujeron restricciones y los estudiantes de doctorado no podían participar. Por suerte tenía de mi parte la más grande motivación de todas: me gustaba la investigación.

Los primeros meses fueron muy duros. Después de las 6-7 horas de jornada laboral, tenía otras tantas de cursos de doctorado, experimentos, programación y reuniones. Pero finalmente el esfuerzo dio su fruto: la fortuna, esta vez sí, me sonrió al conseguir una beca FPI (paupérrima) para quedarme al menos 4 años bajo el techo del grupo eDragon sin tener necesidad de esta cansada dualidad de trabajos. Aunque a priori la FPI fue una buena idea, el hecho de tener que marcarme yo mismo los plazos produjo una situación contraria a la esperada; demasiadas jornadas largas e intensas, sobre todo con mi segundo “director” en algunas fases de la tesis, Samuel Kounev, mientras trabajábamos para sacar adelante algunos artículos ;)

Me gustaría mostrar mi agradecimiento al grupo eDragon en su totalidad y a Jordi Torres en especial, por su apoyo, consejos y buenos ratos que me ha ofrecido, además de permitirme hacer los horarios que quisiera (aunque muchas veces eso me hiciera superar ampliamente las 8 horas de jornada laboral, lo hacía con gusto). Su ayuda en las distintas fases de la tesis me ha ofrecido una visión distinta a la que yo suelo tener; además, el permitirme entrar y colaborar con el BSC dentro de su grupo ha tenido al final un balance positivo.

Finalmente, un saludo a mi compañero Ferran Julià por su ayuda y compañía en gran parte de este trabajo. También a Javier Alonso e Íñigo Igoiri por sus consejos y conversaciones. Agradecer también a Samuel Kounev su tutela y su ayuda en los artículos en los que ha colaborado; sin duda parte del éxito que pueda tener esta tesis es gracias a su “know-how”. Gracias además por permitirnos utilizar SimQPN. Tampoco querría dejar

de agradecer a Kevin Hogan toda la ayuda ofrecida con el inglés.

Un abrazo también a mi compañero Antonio Ramos “bigboss” por todos los años consumidos realizando el Mediaplayer de PS2Reality (por suerte no solapados con la tesis) y sus intentos, de momento infructuosos, para que me fuera al mundo empresarial.

Finalmente, un agradecimiento especial a Natalia Fos, que me ha apoyado todos estos años, difíciles, en los que muchas veces he tenido ganas de abandonar. Espero que me siga apoyando en los siguientes.

ACTA DE CALIFICACIÓN DE LA TESIS DOCTORAL

Reunido el tribunal integrado por los abajo firmantes para juzgar la tesis doctoral:

Título de la tesis: APLICACION DE LA SIMULACION EN TIEMPO REAL PARA MEJORAR LA CALIDAD DE SERVICIO DEL MIDDLEWARE.....
Autor de la tesis: RAMON NOU CASTELL.....

Acuerda otorgar la calificación de:

- ☐ No apto
- ☐ Aprobado
- ☐ Notable
- ☐ Sobresaliente
- ☐ Sobresaliente Cum Laude

Barcelona, …… de ….....…….. de .....….

El Presidente

.....
(nombre y apellidos)

El Secretario

.....
(nombre y apellidos)

El vocal

.....
(nombre y apellidos)

El vocal

.....
(nombre y apellidos)

El vocal

.....
(nombre y apellidos)

Aplicación de la simulación en tiempo real para mejorar la calidad de servicio del Middleware

Ramon Nou Castell

Abstract

La simulación es una técnica utilizada para predecir el comportamiento de sistemas en multitud de áreas. Las simulaciones de componentes hardware son muy comunes, dado el coste de construcción de los sistemas simulados (procesadores, memorias...). Sin embargo, el uso de la simulación en entornos complejos, como es el middleware, y su aplicación en gestores de recursos tiene un uso muy bajo. En esta tesis se propone la utilización de métodos de simulación para incrementar el rendimiento y la calidad de servicio de estos sistemas. Por otro lado, se incluye una ampliación de un sistema de monitorización global para aplicaciones mixtas (JAVA y C) que nos ofrece la posibilidad de conseguir información de lo que ocurre en el middleware y de relacionarlo con el sistema. Se propone y se muestra la creación de un gestor de recursos capaz de repartir los recursos en un entorno heterogéneo utilizando la predicción para tener en cuenta diferentes parámetros de calidad de servicio. En la tesis se muestran los mecanismos de creación de los distintos simuladores, las herramientas de obtención de datos y monitorización, así como mecanismos autónomos que pueden alimentarse de la predicción para producir mejores resultados. Los resultados obtenidos, con gran impacto en la QoS en el gestor creado para Globus, demuestran que los métodos aplicados en esta tesis pueden ser válidos para crear gestores de recursos inteligentes, alimentados de las predicciones del sistema para tomar decisiones. Finalmente, utilizamos las simulaciones realizadas incorporándolas dentro de un prototipo de gestor de recursos heterogéneo capaz de repartir los recursos entre un entorno transaccional y un entorno Grid dentro del mismo servidor.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

La simulación es una de las herramientas de las que dispone la investigación actual. Mediante ella podemos representar sistemas y obtener resultados sin necesidad de disponer del sistema real, en algunas situaciones inexistente. Podemos encontrar numerosos ejemplos en la investigación de *hardware*, como pueden ser el estudio de procesadores o memorias caché. También encontramos el uso de la simulación para probar configuraciones o aportaciones en planificación, pero su uso en entornos en tiempo real es nulo. Podemos encontrar técnicas similares utilizando métodos analíticos, como aparece en [Menascé et al., 2005], pero los métodos analíticos pierden su funcionalidad cuando trabajamos con sistemas no estables. Además, con ellos no podemos realizar o introducir técnicas como controles de admisión, puesto que significaría que las entradas no son iguales a las salidas.

Uno de los principales problemas con la simulación es su elevado coste, pero muchas veces es más costoso el sistema real; en la simulación de *hardware* el coste de implementar un simulador y de ejecutarlo es menor que el de realizar una modificación en el *hardware* (muchas veces imposible de realizar). En la actualidad el *handicap* del tiempo de ejecución se ve reducido gracias al incremento en potencia de cálculo y sobre todo en el uso de multiprocesadores; podemos reservar un procesador para ejecutar simulaciones del entorno mientras éste está funcionando para intentar mejorar su rendimiento mediante la modificación de alguno de sus parámetros. El coste puede bajar aún más si utilizamos una infrastructura *Grid*, tanto por la mayor disponibilidad de procesadores como por la posibilidad de paralelizar la simulación. El coste de construcción es proporcional a la precisión y velocidad que queremos obtener, y es reducible cuando tenemos módulos ya construídos para los subsistemas más comunes (Conexiones *SSL*, procesadores con *PS* (*Processor Sharing*), bases de datos...).

Aunque existen multitud de aplicaciones o entornos (como pueden ser *streaming*, sistemas *batch*, indexación, cálculo intensivo, transaccionales, grid...), nos centraremos en mejorar el rendimiento del *middleware* transaccional y *Grid*. Dentro de ellos hemos seleccionado *middleware* representativo y ampliamente usado: Tomcat para aplicaciones transaccionales y Globus Toolkit para el *middleware Grid*.

1.1. Propuesta y contribución

Nuestra propuesta es la de mejorar la calidad de servicio (*QoS*) del *middleware* utilizando la simulación *online*. Esta calidad de servicio se traducirá en la posibilidad de validar SLA (*Service Level Agreement*) con el cliente y/o la obtención de un rendimiento acorde con los recursos adquiridos. Para ello planteamos el uso de la simulación de entornos extendidos, como puede ser un servidor de aplicaciones o una red *Grid* en la que se ejecutan servicios y trabajos. Existen otro tipo de entornos o aplicaciones que podríamos utilizar como sistemas batch, aplicaciones centradas en la E/S, aplicaciones intensivas... Aún así nos centraremos en las transaccionales y en las *Grid* por ser las más interesantes, variadas y utilizadas tanto en el mundo empresarial como en el de investigación. Esta capa de predicción será parte de un entorno autonómico (*Autonomic System*). De este modo, el objetivo de la tesis es demostrar que la predicción es una herramienta válida para utilizarse en este tipo de entornos y que su utilización introduce claras ventajas. Tenemos los siguientes subobjetivos:

Creación de modelos de simulación para aplicaciones complejas. Hemos realizado la simulación de un servidor de aplicaciones Tomcat sobrecargado, así como la simulación de una serie de nodos *Grid*. Para crear los modelos necesitamos conocer con detalle el funcionamiento de las aplicaciones, y la monitorización nos facilita la tarea. Hemos ampliado *JIS* [Carrera et al., 2003] para poder utilizar la monitorización en cualquier aplicación Java. La ampliación de *JIS*, *BSC-MF*, ha permitido utilizar dicha monitorización en entornos externos a nuestro grupo de investigación. Las simulaciones creadas nos han permitido predecir con precisión el comportamiento de los middlewares estudiados. Hemos utilizado dos métodos de simulación; uno más cercano a un método analítico, *QPN* con *SimQPN*, y otro utilizando un motor de simulación, *OMNeT++*. En el primer método hemos contado con la ayuda de Samuel Kounev y hemos conseguido recrear un entorno *Grid*, no saturado, con gran calidad en las predicciones. Con la simulación hemos conseguido reproducir el comportamiento de un entorno saturado, transaccional con Tomcat, y la aplicación tanto de controles de admisión como de cargas dinámicas. Además, hemos realizado también la simulación del entorno *Grid* mediante *OMNeT++* para su comparación con

QPN. La velocidad de los dos métodos es similar, si bien la calidad de los resultados es estadísticamente más correcta y completa con *QPN*, aun así para nuestras necesidades nos sirve cualquier método.

Estudio de las capacidades de autogestión de las aplicaciones complejas. Para poder utilizar la predicción que hemos creado anteriormente, hemos elaborado una serie de componentes autogestionables para un nodo *Grid*, que incrementan el rendimiento del sistema. En primer lugar, creamos un componente capaz de gestionar los trabajos que entran en el sistema, de manera que se limitan al número de procesadores disponibles en la máquina. Esto produce una reducción en el número de trabajos *Grid* que fallan (pérdidas, cancelaciones, errores) en el sistema. Posteriormente, ampliamos esta propuesta para crear una solución en la que además de reducir las pérdidas se mejore el rendimiento (o se asemeje al sistema funcionando correctamente). Esta nueva propuesta, incremental, tiene como problema el poco dinamismo de la solución. Para resolverlo, proponemos finalmente un prototipo de sistema autogestionable para un nodo de entrada *Grid*, que resuelve muchos de los problemas y se adapta al sistema y a la carga de forma dinámica. Todos estos componentes pueden alimentarse de los componentes de predicción creados. La autogestión es la única forma de mejorar el rendimiento salvando la complejidad de estos entornos. Gracias a estas mejoras del *middleware* hemos aumentado el rendimiento y la calidad del *middleware Grid*. Finalmente, modificamos un gestor de recursos para servidores compartidos y aplicaciones homogéneas para que funcionara con aplicaciones heterogéneas. Gracias a este gestor de recursos conseguimos que dos aplicaciones saturadas y con distintas necesidades convivan en un mismo servidor, mientras mantienen una buena calidad de servicio. En los siguientes trabajos lo ampliaremos para utilizar la predicción aumentando la versatilidad del gestor.

Creación de entornos de autogestión y aplicación de la predicción. Primero creamos un gestor de recursos con *QoS* para Globus utilizando predicción. La predicción es la que se ha creado con *QPN* y *OMNeT++* en el primer subobjetivo. Gracias a este gestor conseguimos que todos los *SLA* de los clientes (aceptados) se cumplan. Los *SLA* consisten en mantener un tiempo de respuesta mínimo en las distintas peticiones de servicio que realizan al servidor. Disponemos también de un *SLA* por parte del servidor para mantener la carga del sistema dentro de un límite; gracias a este límite podemos utilizar el resto del servidor o mantener el servicio en unos límites controlados donde el servidor funcione correctamente (no está sobrecargado, por ejemplo). También introducimos la posibilidad de la reconfiguración dinámica cuando el entorno cambia: si desaparecen servidores (por culpa de fallos, *software-aging*, ...) o necesitamos más servidores (carga insuficiente, ventajas económicas

para coger un nuevo servidor, ...) podemos mover o cancelar trabajos para adaptarnos al nuevo entorno. Posteriormente, y elevando el nivel de abstracción, integramos un servidor de aplicaciones y un servidor *Grid* en un nodo heterogéneo. Para su correcto funcionamiento y distribución de la carga, utilizamos las dos simulaciones creadas para alimentar a un gestor de recursos que controle las aplicaciones que contiene el nodo. Con él mantenemos la *QoS* de las dos mientras repartimos los recursos de forma inteligente. Utilizar entornos heterogéneos autogestionados es de vital importancia en entornos virtualizados; para ello integramos las simulaciones realizadas en el gestor de recursos heterogéneo creado.

1.1.1. Aportaciones de la tesis

La tesis la podemos estructurar según la figura 1.1, en la que mostramos los diferentes componentes donde hemos trabajado y las aportaciones de la tesis de Jordi Guitart [Guitart, 2005].

Enumeraremos a continuación las diferentes aportaciones y las publicaciones que han generado.

1. Crear simuladores capaces de predecir el comportamiento de aplicaciones y *middleware* complejo. Los simuladores incluyen la simulación de un entorno *Grid* y de un servidor de aplicaciones. Simularemos estos entornos teniendo como premisa la velocidad de la simulación y precisión de los resultados.

   - Nou, R., Guitart, J., Beltran, V., Carrera, D., Montero, L., Torres, J., and Ayguadé, E. (2005). **Simulating Complex Systems with a Low-Detail Model**. In XVI Jornadas de Paralelismo, pages 201–308, Granada, Spain.
   - Nou, R., Guitart, J., Carrera, D., and Torres, J. (2006a). **Experiences with simulations - a light and fast model for secure web applications**. In ICPADS (1), pages 177–186. IEEE Computer Society.
   - Nou, R., Guitart, J., and Torres, J. (2006b). **Simulating and modeling secure web applications**. In Alexandrov, V. N., van Albada, G. D., Sloot, P. M. A., and Dongarra, J., editors, International Conference on Computational Science (1), volume 3991 of Lecture Notes in Computer Science, pages 84–91. Springer.

2. Crear y mejorar los mecanismos para obtener información del sistema. Hemos realizado mejoras en *JIS* y se ha creado una framework de monitorización, especializado en Java, llamado *BSC-MF*.

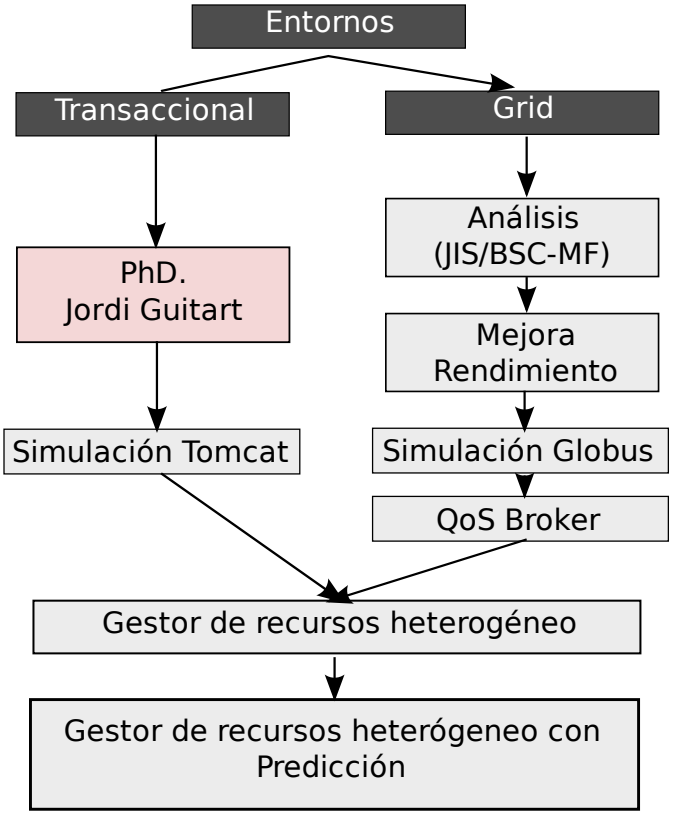


Figura 1.1: Estructura de la tesis y sus aportaciones.

- Nou, R., Julià, F., Carrera, D., Hogan, K., Caubet, J., Labarta, J., and Torres, J. (2006c). **Monitoring and analysing a grid middleware node**. Proc. 7th IEEE/ACM International Conference on Grid Computing 2006, GRID 2006, Barcelona, Spain.
- Nou, R., Julià, F., Carrera, D., Hogan, K., Caubet, J., Labarta, J., and Torres, J. (2007a). **Monitoring and analysis framework for grid middleware**. In PDP, pages 129–133. IEEE Computer Society.
- Caubet, J., Hogan, K., Nou, R., Labarta, J., and Torres, J. (2006). **Supporting the Introduction of Autonomic Computing in Middleware**. Engineering Conference VII, IBM Academy of Technology Conference IBM Hursley ,

England , October 2006.

3. Analizar el *middleware Grid* (Globus) para mejorar su rendimiento. Hemos construido mecanismos de autogestión que se pueden alimentar de la simulación y de sus predicciones para mejorar su comportamiento. Esto es necesario para evitar pérdidas de rendimiento en sistemas sobrecargados.

   - Nou, R., Julià, F., and Torres, J. (2007c). **Should the grid middleware look to self-managing capabilities?** The 8th International Symposium on Autonomous Decentralized Systems (ISADS 2007) Sedona, Arizona.
   - Nou, R., Julià, F., and Torres, J. (2007d). **The need for self-managed access nodes in grid environments** The 4th IEEE Workshop on Engineering of Autonomic and Autonomous Systems (EASe 2007), Tucson, Arizona.

4. Creación de un gestor de recursos para un entorno *Grid*, mediante simulación. El QoS Broker, programado utilizando dos entornos de simulación distintas, permite mantener la *QoS* y los *SLA* del entorno.

   - Nou, R., Kounev, S., and Torres, J. (2007e). **Building online performance models of grid middleware with fine-grained load-balancing: A globus toolkit case study**. In Wolter, K., editor, European Performance Engineering Workshop (EPEW'07), volume 4748 of Lecture Notes in Computer Science, pages 125–140. Springer.
   - Kounev, S., Nou, R., and Torres, J. (2007). **Autonomic QoSaware resource management in grid computing using online performance models**. Second International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS-2007), Nantes, France.
   - Nou, R. Kounev, S., Julià, F.,and Torres, J. (2008). **Autonomic QoS control in enterprise Grid environments using online simulation**. Journal of Systems and Software (Pendiente de publicación).

5. Construcción de una capa de autogestión para repartir los recursos en un entorno heterogéneo. En un primer lugar no utilizaremos simulación, sino que modificaremos las aplicaciones para que se comuniquen con el gestor. Finalmente, uniremos las simulaciones creadas para que tomen decisiones sin necesidad de modificar las aplicaciones, pero añadiendo políticas de control de admisión o de carga.

   - Nou, R., Julià, F., Guitart, J., and Torres, J. (2007b). **Dynamic resource provisioning for self-adaptive heterogeneous workloads in SMP hosting platforms**. International Conference on E-business (2nd) ICE-B 2007, Barcelona, Spain.

1.2. Organización de la tesis

Hemos dividido esta tesis en los siguientes capítulos (señalamos también las referencias a los artículos que los componen):

2 - Estudio de Tomcat y su posterior simulación cuando está en un entorno no estable. [Nou et al., 2005, Nou et al., 2006b, Nou et al., 2006a].

3 - Estudio de Globus, su monitorización para obtener información del entorno y diferentes propuestas de autogestión. [Nou et al., 2006c, Nou et al., 2007a, Caubet et al., 2006].

4 - Estudio de plataformas compartidas, en auge gracias a la virtualización, y utilización de mecanismos de autogestión a más alto nivel. [Nou et al., 2007b, Nou et al., 2007c, Nou et al., 2007d].

5 - Simulación de Globus y creación de un gestor de recursos que usa predicción en tiempo real para mejorar la calidad de servicio. [Nou et al., 2007e, Kounev et al., 2007], Journal of Systems and Software (Pendiente, Julio 2008).

6 - Creación de un gestor de recursos para un entorno heterogéneo (funcionando en un entorno virtualizado) utilizando las simulaciones creadas.

Finalmente en el capítulo 7 presentamos las conclusiones de la tesis y el trabajo futuro.

1.3. Reconocimientos

Para la realización de este trabajo hemos contado con la estrecha colaboración de Samuel Kounev para poder utilizar SimQPN como método alternativo para la simulación. Este trabajo ha sido financiado por el Ministerio de Ciencia y Tecnología de España y la Unión Europea en virtud de los contratos TIN2004-07739-C02-01 y TIN2007-60625. Partes de este trabajo han sido financiadas por la Comisión Europea en virtud del contrato 034286 (IST - SORMA).

Capítulo 2

Simulación de un middleware transaccional en un entorno complejo

En este capítulo, mostramos el trabajo relacionado que nos ha conducido a valorar la utilización de técnicas de simulación para predecir el comportamiento de un sistema. En nuestro caso concreto utilizamos como motivación y base los resultados de la tesis [Guitart, 2005]. Finalmente, veremos como podemos simular el comportamiento de los distintos parámetros del software para realizar, por ejemplo, autogestión utilizando predicción.

Nuestro primer objetivo será simular este entorno teniendo como premisa la velocidad de la simulación y precisión de los resultados para que puedan utilizarse en estudios de hipótesis y pruebas preliminares. Podemos usar un modelo de simulación para estimar el rendimiento del sistema. Esto nos permite tener una plataforma de análisis del estilo *what-if* o "¿qué pasaría si?" para ofrecernos soporte y ayuda para tomar decisiones.

2.1. Middleware transaccional

Podríamos definir una transacción como una interacción entre diferentes componentes (base de datos, cliente, servidor...) de forma atómica. El ejemplo más claro lo tenemos en una base de datos transaccional, donde las operaciones de inserción, consulta, borrado o modificación cumplen las propiedades *ACID (Atomicity, Consistency, Isolation, Durability)*. Podemos encontrar infinidad de ejemplos de aplicaciones transaccionales que funcionan sobre un *middleware* transaccional ofreciendo dichas aplicaciones a internet. Por ejemplo: páginas de bancos, subastas por Internet, agencias de viajes, compras de billetes.... Todas ellas páginas en las que se realizan millones de transacciones diarias y

que necesitan mantener el rendimiento y la *QoS* a un buen nivel.

En nuestro entorno experimental utilizaremos un *middleware* que permite la ejecución de aplicaciones transaccionales: Apache Tomcat.

Apache Tomcat, es un contenedor de servlets creado en la *Apache Software Foundation* que implementa las especificaciones de servlets y de *JSP* de Sun. Se trata de una implementación de referencia oficial; sin embargo, es un excelente contenedor de servlets y servidor de aplicaciones usado en multitud de empresas y sitios web.

Tomcat está creado en *Java* y utiliza código abierto, haciéndolo interesante para su estudio. Actualmente, Tomcat está en la versión 6 con más de 8 años de desarrollo a su espalda.

Los sistemas complejos, como un servidor de aplicaciones web, son ampliamente utilizados hoy en día. Sin embargo, cuando necesitamos añadir seguridad surgen problemas de rendimiento; para resolverlos necesitamos trabajar y probar diferentes alternativas o propuestas sobre ellos. Un ejemplo de aplicación web podría ser la página web de eBay: eBay recibe una gran cantidad de peticiones, que necesitan ejecutarse sin fallos y en un entorno seguro, y que se incrementan cuando una subasta está a punto de finalizar. El uso *in crescendo* de esta clase de aplicaciones y la necesidad de incrementar su rendimiento ha llevado a la creación de un conjunto de benchmarks, como puede ser *RUBiS* (*Rice University Bidding System*) [Amza et al., 2002]. *RUBiS* genera una carga similar a la ofrecida por los clientes de eBay, y es la que utilizaremos normalmente en los experimentos, por su configurabilidad y adaptabilidad a nuestros entornos.

2.2. Trabajo relacionado

Como hemos comentado anteriormente, la motivación de la tesis proviene de [Guitart, 2005]: En dicha tesis se detecta que estos sistemas complejos disminuyen su rendimiento cuando el sistema está sobrecargado. En el caso concreto de Tomcat hay una carga límite a partir de la cual el rendimiento cae en picado, como podemos observar en la figura 2.1.

Para describir el entorno de un servidor de aplicaciones podemos decir que las aplicaciones web dinámicas son aplicaciones de *n-capas* y están normalmente formadas por una capa cliente y una capa servidor que consiste en un servidor web al frente, un servidor de aplicaciones y una base de datos. La capa del cliente es responsable de la interacción con la aplicación y de generar las peticiones hacia el servidor. El servidor implementa la lógica de la aplicación y es responsable de tratar las peticiones que recibe del usuario (o usuarios).

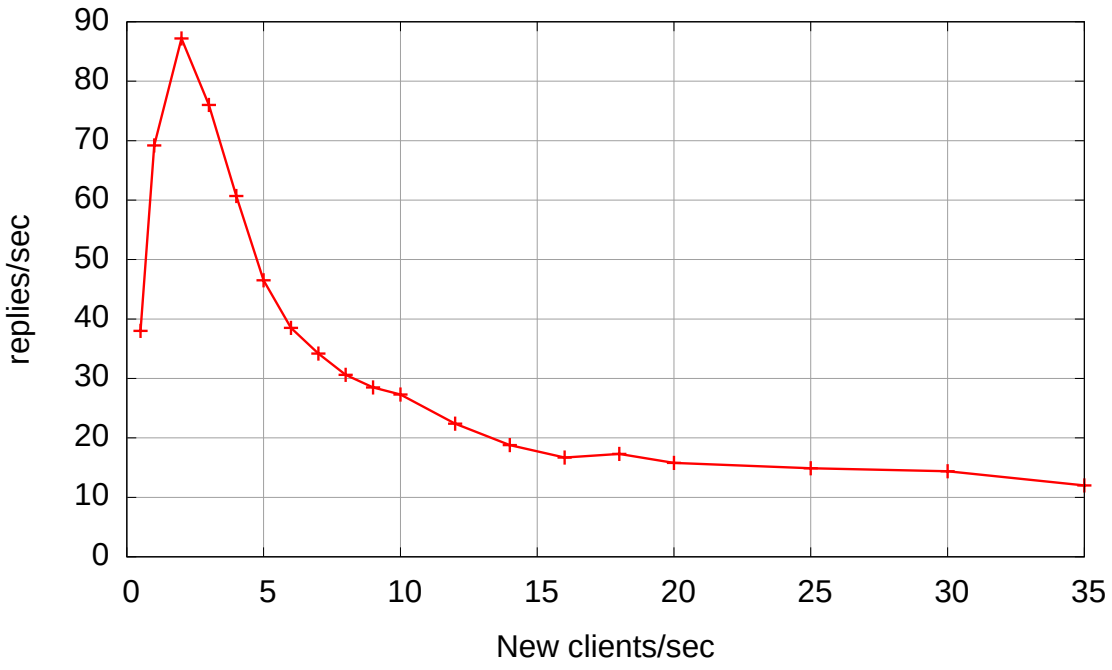


Figura 2.1: Rendimiento del servidor de aplicaciones Tomcat sobrecargado.

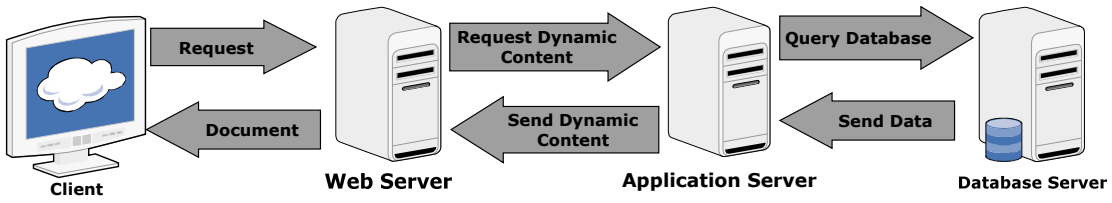


Figura 2.2: Ejemplo de servidor de 3-capas.

Cuando el cliente envía una petición *HTTP* al servidor web para obtener contenido dinámico, el servidor de aplicaciones ejecuta el código correspondiente, que puede necesitar acceder a una base de datos para generar la respuesta. El servidor de aplicaciones forma la respuesta en una página *HTML*, que se devuelve, finalmente, como una respuesta *HTTP* hacia el cliente. Podemos observar un ejemplo de flujo en la figura 2.2.

El entorno experimental consiste de una configuración típica de servidor web: un servidor de 4 procesadores con un Tomcat usando seguridad *SSL*, un servidor de 2 proce-

Cuadro 2.1: Tiempos observados de las distintas fases de una petición, para un Pentium Xeon, 4-way 1.4 GHz. [Guitart, 2005]

(a) Tiempos de una conexión *SSL*.

	Tiempo de CPU
***SSL handshake* (Primera conexión)**	170 ms.
***SSL handshake* (Reusado)**	2 ms.
Conexión, petición	3 ms.

(b) Tiempos de proceso de los recursos más comunes de *RUBiS*.

Recurso	**Frecuencia**	**Procesado**
RUBiSlogo.jpg	35.6 %	3.65 ms.
Bidnow.jpg	26.1 %	0.17 ms.
SearchItems	14.3 %	21 ms.

sadores con una base de datos *MySQL* [WebPage, 2008] que ofrece datos al sistema, y finalmente un cliente *httperf* [Mosberger and Jin, 1998] generando una carga típica de *RUBiS* (*Rice University Bidding System* [Amza et al., 2002]. Tanto el cliente como la base de datos no son cuellos de botella.

Httperf permite crear un flujo continuo de peticiones *HTTP/S* enviadas desde uno o más clientes y procesados por un servidor. La prueba está configurada para tener un *timeout* de 10 segundos por cliente; cuando aparece un *timeout* se crea una nueva petición (el cliente reintenta). Esta nueva petición produce un nuevo negociado de *SSL* (*SSL handshake*) y provoca una reducción del rendimiento general.

La situación comentada la podemos analizar con Paraver [Jost et al., 2003] utilizando trazas de la aplicación generadas con *JIS* (base del actual *BSC-MF*). *JIS* nos permite ver todo el sistema de forma vertical, detectando que la sobrecarga proviene del gran número de peticiones de negociado *SSL*.

El protocolo *SSL* (detalle en [Dierks and Allen, 1999,Freier et al., 1996]) proporciona privacidad en las comunicaciones de Internet; el protocolo permite que las aplicaciones cliente/servidor se comuniquen de manera que no se permitan escuchas, cambios de mensaje y falsas autentificaciones. A pesar de estas ventajas, *SSL* incrementa el consumo de recursos para servir una conexión por el uso de la criptografía.

Analizando el tiempo de CPU de estas conexiones *SSL* nos encontramos con los datos del cuadro 2.1(a): Observemos como el tiempo que tarda en procesar un nuevo *SSL* es

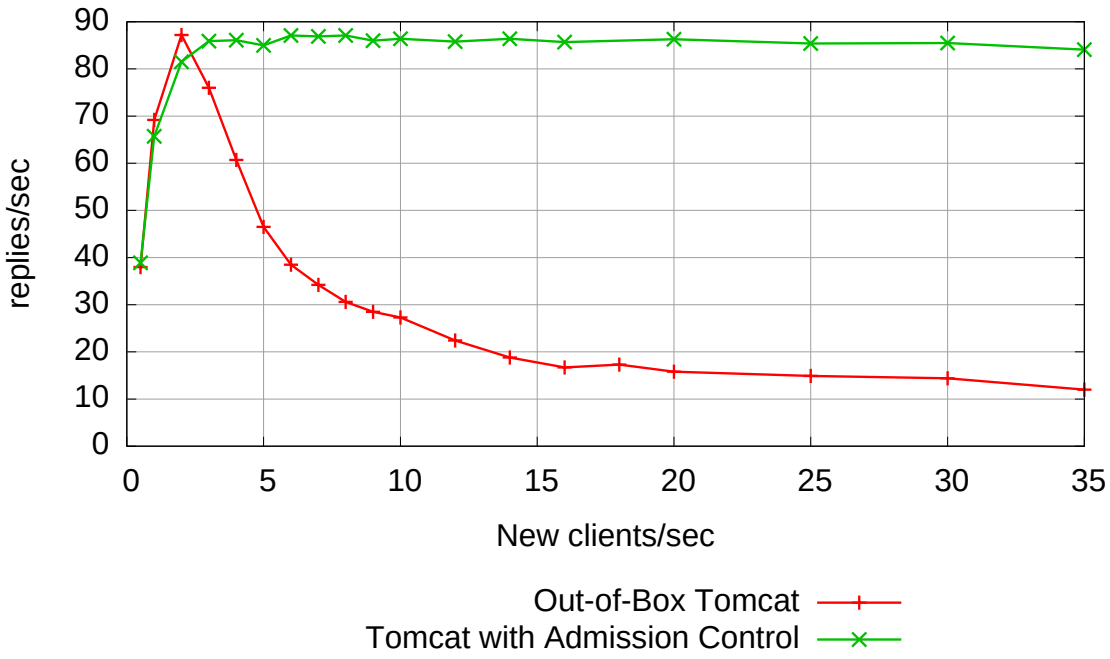


Figura 2.3: Rendimiento del servidor de aplicaciones utilizando un control de admisión. [Guitart, 2005]

100 veces superior al tiempo de procesado que necesita para procesar una conexión ya existente. Esta diferencia de tiempo afecta al rendimiento del servidor, como se puede ver en [Guitart et al., 2005a]. Este estudio concluye que el rendimiento máximo obtenido mediante conexiones *SSL* es 7 veces menor que usando conexiones normales sin seguridad. Según estos datos, se plantea el uso de un mecanismo de control de admisión [Guitart et al., 2005b]; el control impide la bajada de rendimiento diferenciando las peticiones realizadas por nuevos clientes (que necesitan un negociado *SSL* nuevo) de las realizadas por los antiguos clientes (que reutilizan una conexión *SSL*). Para destacar el consumo de tiempo de procesador en el procesado *SSL* podemos ver en el cuadro 2.1(b) los tiempos necesarios para obtener/procesar un recurso de *RUBiS*; claramente es muy costoso.

Utilizando la propuesta anterior se consigue el perfil de la figura 2.3, donde se observa que el servidor es capaz de mantener el rendimiento de la aplicación en el punto más alto.

2.3. Propuesta de simulación

2.3.1. Introducción

Tal como explicamos, nuestro primer entorno incluye algunas características que lo hacen difícil de simular utilizando técnicas analíticas (redes de colas, cadenas de *Markov*, *petri nets*); entre ellas destacamos que el sistema no está en equilibrio ($entradas \neq salidas$) y además no es estable. Al tener un sistema sobrecargado y con pérdidas no podemos utilizar técnicas clásicas (para encontrar puntos más alla de un determinado punto de carga).

La ejecución de *benchmarks* para probar configuraciones y cambios en el sistema es costoso: requieren tiempo y recursos. Para ejecutarlos necesitamos normalmente un cliente y un servidor (normalmente con más de un procesador). Además, las pruebas requieren ejecutarse en tiempo real en un sistema exclusivo para reducir las interferencias en los resultados. Los recursos son difíciles de adquirir, por eso es importante encontrar otra manera de ejecutar estas pruebas de una manera rápida y sin consumir recursos (sólo los necesarios para la simulación). ¿Cómo podríamos obtener medidas de rendimiento cuando el sistema no existe o no podemos conseguir los recursos necesarios? En este caso podemos usar un modelo para estimar el rendimiento del sistema. Esto nos permite tener una plataforma de análisis del estilo (*what-if*) o "¿que pasaría si?" para ofrecernos soporte y ayuda para tomar decisiones.

2.3.2. Simulación y predicción de entornos complejos

En entornos *e-business* podemos encontrar diferentes propuestas y alternativas en la literatura para el análisis de rendimiento y la creación de modelos de predicción. El sistema más común es utilizar modelos analíticos en los cuales el análisis está basado en Cadenas de *Markov* [Bolch et al., 1998]. La teoría de redes de colas [Xu et al., 2005] y de *Petri Nets* [Kounev and Buchmann, 2003](*QPN*) son los más comunes.

Pero dada la complejidad de los sistemas actuales, los modelos analíticos no son viables. Sólo simplificando el sistema podemos obtener sistemas tratables; sin embargo esta simplificación puede llevarnos a resultados erróneos. Nos encontramos también con serios problemas (o la imposibilidad) para modelar el comportamiento de los *timeouts*. Este comportamiento es muy importante; los *timeouts* sobrecargan el sistema y producen efectos negativos en entornos seguros (*SSL*).

Un modelo de simulación [Law and Kelton, 2003, Bratley et al., 1987] es una representación abstracta de los elementos del sistema y de sus interacciones; es una alternativa

a los modelos analíticos matemáticos. La principal ventaja de la simulación es que supera las limitaciones de los complejos modelos teóricos; además, la metodología de la simulación y el análisis de sus resultados está sustentado por principios estadísticos desarrollados los últimos 30 años. Aún así su principal problema, en determinados casos, es la estabilidad de los resultados obtenidos. En la simulación normalmente definimos el tiempo que vamos a simular, por lo que determinadas leyes, como por ejemplo *Little* [Little, 1961], donde el tiempo de respuesta medio es independiente del planificador, se pueden llegar a invalidar.

Durante las próximas secciones describiremos, junto con su metodología, la construcción de un modelo para sistemas complejos basados en un servidor de aplicaciones. Este modelo nos ofrecerá resultados y tendrá un bajo coste de ejecución. Además, presentaremos los resultados obtenidos, así como una serie de trabajo futuro creado a partir de la simulación de hipótesis. Finalmente, veremos que la simulación se adapta bien al sistema real, un sistema complejo, pudiéndose utilizar en distintos entornos donde la predicción puede ofrecerle ventajas.

Además, mostraremos como la simulación nos da la flexibilidad y capacidad para extraer datos y ejecutar algunas pruebas que serían imposibles utilizando recursos reales: Podemos medir qué rendimiento (*throughput*) obtenemos añadiendo más procesadores, *threads* o cambiando el número de clientes. No necesitamos utilizar el sistema real para esto, sólo la simulación.

Como trabajo relacionado en este entorno podemos encontrar a [McGuinness et al., 2004, McGuinness and Murphy, 2005], que utilizan *Ptolemy II* y *Workbench* como entorno de simulación. Estas simulaciones muestran el rendimiento de *EJB* (*Enterprise Java Beans*), no usan *timeouts* ni tienen en cuenta los problemas de seguridad *SSL* como en nuestra propuesta. *Workbench* es un entorno comercial y *Ptolemy II* es un framework basado en Java que nos permite seleccionar diferentes métodos de resolución (*FSM*, eventos discretos, tiempo discreto...), con lo que son herramientas similares a *OMNeT++* [Vargas, 2001], que es la que hemos utilizado.

En [Xu et al., 2005] puede encontrarse otro método de resolución; en él sólo se describe el comportamiento de *EJB* sin seguridad ni *timeouts* y utiliza un método basado en redes de cola por capas (*Layered Queueing Networks*). Enfoques con métodos tradicionales pueden encontrarse en [Kounev and Buchmann, 2003, Stewart and Shen, 2005, Uragonkar et al., 2005], pero no incluyen seguridad ni la simulación del comportamiento de los *timeouts*. El método utilizado por S. Kounev [Kounev and Buchmann, 2003] (con el cuál hemos colaborado) con *SimQPN* y *QPN* lo utilizaremos más adelante, ya que nos permite describir contención de software o hardware (limitación de número de *threads*, por ejemplo) de forma sencilla. Aún así, tampoco nos permite simular entornos sobrecargados.

2.3.3. Métodos de predicción

Resumamos nuestro objetivo: tenemos una serie de experimentos que necesitan ejecutarse sobre un servidor, típicamente un sistema multiprocesador (*SMP*). Estos experimentos pueden durar varias horas para completarse; además, los resultados podrían ser buenos o malos dependiendo de los parámetros elegidos al inicio. También nos encontramos que estos experimentos necesitan un periodo de tiempo continuo para ejecutarse y no se pueden interrumpir porque funcionan sobre una red cliente-servidor. ¿Por qué no buscar otra alternativa para predecir y aproximarse a estos resultados de manera rápida?

Nuestra propuesta se basa en encontrar algún método analítico o basado en simulación para ayudarnos en estas fases de experimentación.

Primera propuesta

Como hemos comentado en la sección 2.3.2, hay diferentes alternativas para realizar análisis de rendimiento y crear modelos para predecir sistemas *e-business*. *QN* y *PN* son los formalismos más comunes y más utilizados, pero dada la complejidad de los entornos actuales los métodos analíticos sólo pueden utilizarse simplificando el sistema: necesitamos obtener modelos tratables [Buch and Pentkovski, 2001, Chu, 1998, Uragonkar et al., 2005]. Otras propuestas usan *PN*, como en [Bause et al., 1995], o ampliaciones como las *QPN* (*Queue Petri Nets*) [Kounev and Buchmann, 2003], las cuáles tienen limitaciones para modelos largos (resueltos utilizando redes jerárquicas) y su uso es *off-line* (aunque como mostraremos en posteriores capítulos hemos conseguido utilizar *QPN on-line*).

Nuestra primera propuesta para el sistema presentado fue usar un modelo analítico, un modelo de colas de *Markov* $G/G/n$ (llegadas y tiempos de servicio generales). El principal problema de esta propuesta es que los modelos analíticos sólo funcionan en sistemas no saturados y estables, de modo que esta clase de colas fallan cuando aparecen *timeouts* y aparecen procesadores sobrecargados. Además, esta alternativa carece de flexibilidad y no nos ofrece el detalle necesario.

Segunda propuesta

Para el segundo método hicimos una búsqueda de herramientas de simulación; necesitamos facilidad de uso y la habilidad de añadir nuestro propio código al simulador. Seleccionamos *OMNeT++* (*Objective Modular Network Testbed in C++*) [Vargas, 2001].

OMNeT++ es un framework de simulación que nos ofrece paso de mensajes, simulación basada en eventos y simulación basada en el tiempo con facilidad.

Gracias al entorno podemos programar los distintos módulos o servicios que necesitamos. En nuestro caso hemos construido todo nuestro código y los servidores, pero es una tarea sencilla realizable en *C++*. La flexibilidad del entorno se traduce en una forma sencilla para añadir nuestro código y añadir dinamismo; además, podemos encontrar en la web de *OMNeT++* experiencias de utilización de código real sin modificaciones en la simulación. *OMNeT++* ofrece también herramientas para visualizar en tiempo real lo que sucede dentro del sistema simulado.

2.3.4. Creación del modelo

Obtener información del sistema

Hemos usado Paraver [Jost et al., 2003] y el Java Instrumentation Suite (*JIS* [Carrera et al., 2003], ampliado en esta tesis) para obtener los tiempos de servicio de los procesos más relevantes que tenemos en nuestro sistema (reflejados en el cuadro 2.1(a). Los tiempos importantes son los requeridos para procesar un *SSL* handshake (nuevo y reusado), el tiempo de servicio para crear una conexión hacia el *HTTPProcessor* y el tiempo para procesar la petición estática o dinámica (mostradas en el cuadro 2.1(b)).

Además, hemos obtenido la carga del cliente mediante un análisis estadístico del *workload* original (archivo con peticiones realizadas a un servidor por *RUBiS*) para obtener su composición. Éste consiste en un conjunto de peticiones realizadas simultáneamente (una página y dos imágenes, por ejemplo), un (*thinktime*) y el % de peticiones estáticas entre las dinámicas. Finalmente, también necesitamos el número de peticiones realizadas por un cliente antes de dejar el sistema. Si quisiéramos podríamos utilizar la tabla de probabilidades (Cadena de *Markov*) que define el benchmark *RUBiS*, pero una simplificación de las variables nos ofrece flexibilidad a la hora de modificar el comportamiento del generador y aplicar otras cargas (*burst*, no *SSL*, cargas estáticas...).

Separar e identificar los componentes del sistema

En este momento hemos de decidir qué puntos han de ser modelados: nuestra intención es obtener una simulación del sistema general, no una emulación de cada bit que se va a cambiar; por lo tanto, necesitamos encontrar un modelo abstracto del sistema que identifique los componentes importantes del sistema y los experimentos. Por ejemplo, si no necesitamos ninguna información del cliente no deberíamos modelarlo con gran detalle, puesto que aumentaría el tiempo necesario para su ejecución. En nuestro caso sólo

necesitamos el *workload* o la carga que genera el cliente hacia el servidor. Resumiendo, sólo debemos modelar lo que necesitamos.

En el servidor necesitamos identificar los componentes importantes utilizando un proceso iterativo usando la información obtenida del sistema. Por ejemplo, en nuestro caso necesitamos modelar la cola de *backlog* del sistema operativo, ya que nuestro sistema está saturado y es un componente importante. Si nuestras pruebas se realizaran en un entorno no saturado podríamos omitir esta cola, ya que no nos aportaría nada. Si no modelamos esta cola veremos resultados que comparados con los reales consiguen un rendimiento inferior, ya que la cola de backlog tiene un tamaño típico de 1.024 y mantiene los paquetes *SYN TCP* [Backlog, 2008]. La cola sirve de primer filtro "natural" para las conexiones.

En el caso de la implementación del servidor de aplicaciones no necesitamos modelar el sistema a nivel de *thread*; si modelamos los *HTTPProcessors* será suficiente y obtendremos buenos resultados. Como en el caso anterior, si necesitamos cambiar políticas de comportamiento de los *threads*, entonces necesitariamos modelarlos. Es un compromiso entre lo que necesitamos (o lo que queremos) y la dificultad, el incremento de complejidad y el tiempo de programar nuestra simulación (en el capítulo 6 veremos como modelamos con detalle esta parte). Para decidir la complejidad del sistema, el tiempo de simulación es un factor importante: algunos componentes o características se diluyen o filtran cuando realizamos ejecuciones largas; si necesitamos simular periodos cortos necesitaríamos habilitar estos componentes en el modelo (otros aspectos como el planificador incrementan su importancia en periodos cortos).

Modelar el cliente (carga)

Nuestro cliente es simple pero aun así es importante, ya que genera la carga que se envía al servidor. Hemos usado un enfoque estadístico para la carga, ofreciéndonos la posibilidad de probar otros tipos de carga fácilmente (*SSL*, diferentes ráfagas, contenido mixto). Además podemos introducir, si fuera necesario, la misma carga que en el experimento. En nuestros experimentos utilizaremos la carga representativa que nos ofrece el benchmark *RUBiS*.

Validación y calibración del modelo

Uno de los procedimientos más importantes cuando tenemos el modelo completo es su calibración; necesitamos ejecutar simulaciones y validar las predicciones usando resultados reales. Con esta validación, además deberíamos detectar cualquier fallo de nuestro

modelo y ajustarlo. Finalmente, la calibración es también importante cuando no simulamos el sistema en detalle: debemos incluir en las variables de entrada detalles como la E/S o la sobrecarga que se produce y que no se percibe con las herramientas de captura de datos. En nuestro caso, al decidir no modelar los *threads* o la contención del sistema, podemos remplazar este subsistema con un enfoque matemático que reproduzca este comportamiento.

Iterar y refinar el modelo

Si nuestro modelo no se adapta a los resultados reales deberíamos revisarlo y añadir más detalle o separar procesos. En nuestro caso, detectamos en esta fase que en el modelo de Tomcat necesitamos más detalle para diferenciar entre tiempo de E/S y tiempo de CPU. La incorporación de la E/S permite que otro proceso entre en el sistema e incrementa de manera global el rendimiento.

Diseñando las pruebas

Si queremos obtener gráficos de tiempo de respuesta, necesitamos modelar todos los procesos o los bloques que afectan este valor (orden de las peticiones, persistencia, etc.). En nuestro caso, nos hemos centrado en obtener medidas de rendimiento (peticiones y respuestas por segundo), así que sólo necesitamos modelar los **HTTPProcessors**, las colas de *backlog* y la persistencia en detalle, ya que todos estos componentes tienen un importante impacto sobre el throughput. Seleccionar qué modelar y qué dejar fuera es importante para reducir la complejidad del modelo y el tiempo de ejecución de la simulación.

Añadir métricas

Para finalizar, debemos añadir algunos contadores de rendimiento para evaluar nuestras pruebas. Por ejemplo: si necesitamos conseguir respuestas por segundo, debemos añadir un contador en el módulo del cliente; este bloque es el que conoce cuándo una petición sale y cuándo llega su respuesta.

2.3.5. Modelo final

El sistema se ha modelado utilizando cinco cajas negras (la figura 2.4 es una visión ampliada): **Client**, que simula *Httperf* y la carga de *RUBiS*; **accessProcessor**, que simula

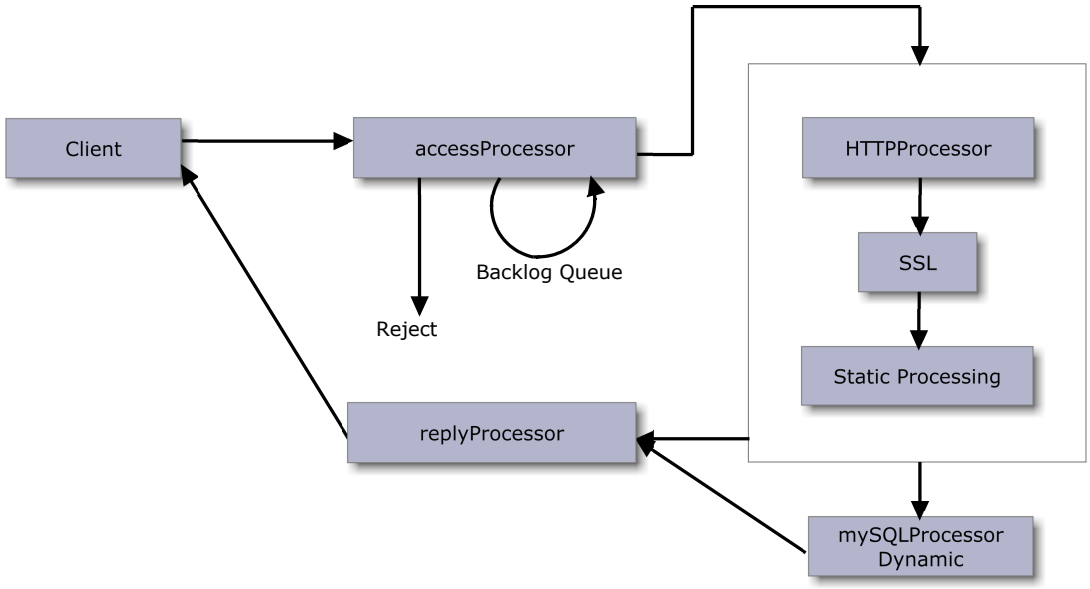


Figura 2.4: Flujo de datos del sistema simulado.

el backlog de conexiones del sistema operativo y permite colocar políticas de control de admisión; **HTTPProcessor** controla las conexiones usando *SSL*, también procesa las peticiones y envía peticiones al bloque **mySQL** (base de datos) si es necesario; por último, **replyProcessor** coge la respuesta de **mySQL** o del **HTTPProcessor** y la envía finalmente al **Client**.

Bloques modelados

Client (client): programado para generar estadísticamente la misma carga que el sistema real. Usando análisis estadístico encontramos cómo funciona el *workload* real. Este bloque controla también los *timeouts*, modelados facilmente con la *API* de *OMNeT++*. Como nota aparte podemos generar más clientes que en el sistema real, ya que no tenemos ninguna limitación ni cuello de botella.

accessProcessor (accproc): una cola *FIFO* envía pedidos hacia el **HTTPProcessor** (si se necesita, podemos realizar un control de admisión). Desde aquí es fácil programar controles, de modo que podemos probar diferentes propuestas cambiando el código. Dentro del bloque se ha modelado el comportamiento típico de los buffers de Linux (backlog).

HTTPProcessor (reqproc): Usa una planificación *Processor Sharing* (*PS*) para simular el comportamiento del procesador. En *OMNeT++* es menos intuitivo modelar una cola *PS* que otro tipo de colas (*FIFO*) que procesan un mensaje en cada instante. Podemos resolverlo creando una función de *poll* que controla el tiempo de simulación. Cuando una petición llega procesamos la conexión (*SSL*, conexión y el servicio de la petición) y procesamos los datos estáticos. Posteriormente, comprobamos si necesitamos procesar de forma dinámica la página; si es así, la enviamos hacia la base de datos. Esta separación en dos fases es utilizada para simular el comportamiento de los *timeouts* de *httperf*.

Este bloque además se consulta a través del **accessProcessor** para simular la persistencia y el comportamiento de *HTTP 1.1*.

mySQLProcessor (mysql): Emula el comportamiento de la base de datos; como nos centramos en Tomcat, este bloque no está modelada con mucho detalle. Utilizamos también un planificador en *PS* para simular el comportamiento original.

replyProcessor (replyproc): Este bloque envía la respuesta al cliente y no hace ningún procesamiento extra. En el sistema real este bloque estaría incluido dentro de Tomcat.

Cualquier otro bloque (por ejemplo, contención en *threads*, parseado de *HTTP*, contención en el log del sistema...) incluido añadiría más detalles y resultados más exactos. Aunque esto seria positivo, también incrementaría la complejidad, así que es importante decidir qué queremos conseguir.

2.4. Evaluación

2.4.1. Validación del modelo

En esta sección presentamos la validación del modelo comparando los resultados predecidos con los del sistema real. Podemos ver en la figura 2.5 y la figura 2.6 cómo los resultados son suficientemente cercanos (en un escenario complejo) para poderlos tomar en serio. El comportamiento normal del sistema puede verse en la primera figura, donde encontramos como el rendimiento cae (debería mantenerse aproximadamente en el nivel más alto en un comportamiento normal) debido a la sobrecarga creada por los nuevos clientes (y los *timeouts*) que necesitan un nuevo negociado *SSL*. Cuando usamos un control de admisión basado en esta clase de *SSL*, el rendimiento no se degrada y crea el perfil

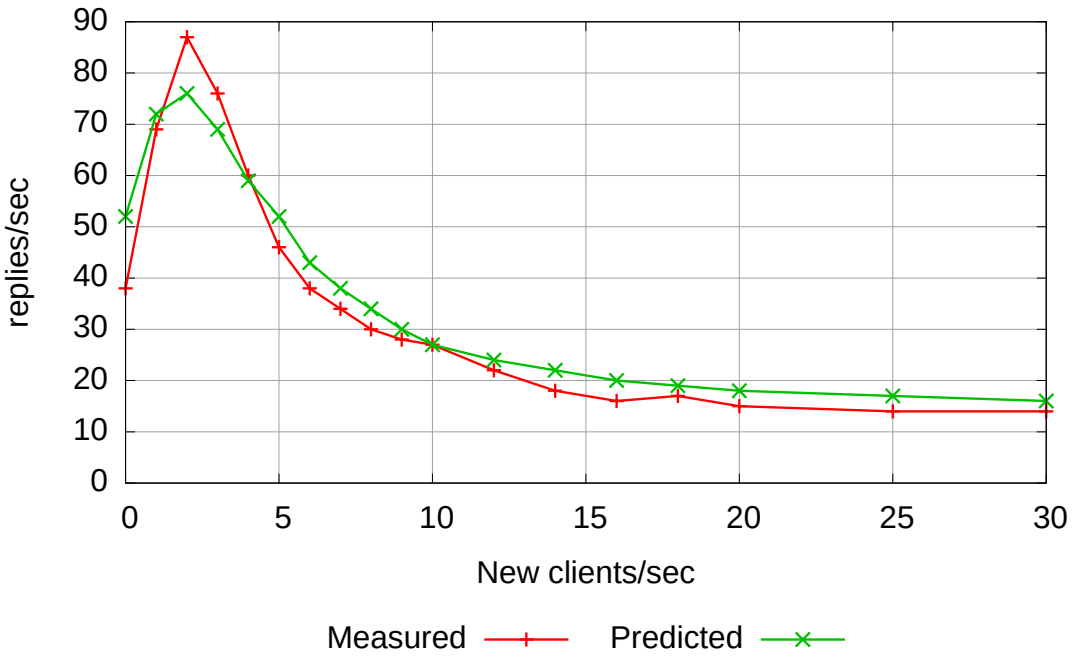


Figura 2.5: Throughput de Tomcat en el sistema experimental y en el modelo predecido (1 CPU).

esperado que podemos ver en la segunda figura. Más información sobre estas gráficas y su interpretación pueden encontrarse en [Guitart et al., 2005b].

En nuestra primera propuesta hemos utilizado 1 CPU; nuestra simulación representa el concepto de CPU simulada dividiendo el tiempo de proceso dentro del **HTTPProcessor** y del **mySQLProcessor** y aplicando una formula de contención (encontrada analizando el sistema experimental). A pesar de no tener un comportamiento exacto, como podemos ver en la subsección 2.4.2, donde mostramos gráficas para multiprocesadores, es una buena aproximación. En el capítulo 6 tenenos una simulación más exacta de este comportamiento.

En la figura 2.6 hemos aplicado el control de admisión para mejorar el rendimiento del servidor. Damos prioridad a las conexiones que reutilizan una sesión SSL existente, de manera que maximizamos el numero de sesiones que se completan igual que se raliza en [Guitart et al., 2005b].

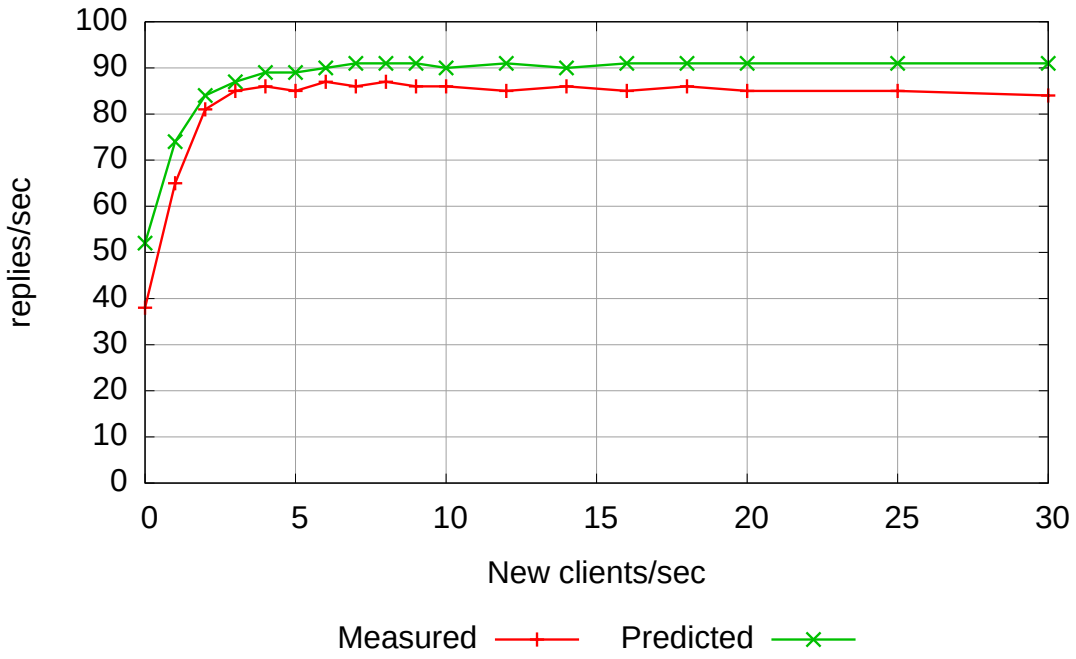


Figura 2.6: Throughput de Tomcat con control de admisión, comparación entre el sistema experimental y el modelo predecido (1 CPU).

2.4.2. Otras pruebas

Este modelo simple se adapta bien al sistema experimental, como hemos visto en la anterior subsección. Además, este tipo de simulación permite ejecutar pruebas sobre hardware o software que aún no está disponible y obtener medidas de rendimiento.

La baja complejidad de esta simulación nos permite simular tiempos de ejecución largos sin un uso computacional grande (más de 100 órdenes de mágnitud inferior). Un ejemplo puede ser simular una serie de diferentes políticas de control de admisión. Además, si fuera necesario, se podría extender este modelo de simulación en cualquier nivel de detalle para obtener otro tipo de información.

Para ilustrar las posibilidades que ofrece este tipo de simulación, en esta sección presentaremos algunos experimentos que no pueden ser ejecutados en un entorno real, ya sea por no estar disponible o porque el tiempo o coste de ejecución/preparación sería demasiado elevado. De esta manera demostramos que la simulación puede ayudar mucho a responder preguntas, confirmar o descartar hipótesis, o darnos pistas acerca de posibles propuestas, que validaríamos en el sistema real *a posteriori*. Como hemos comentado

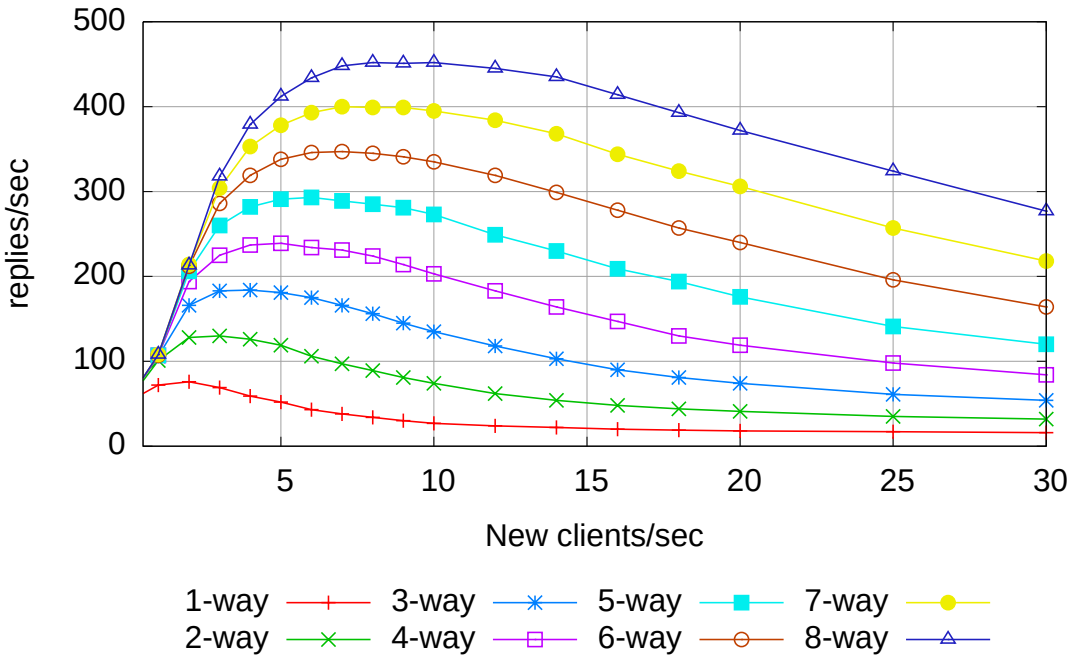


Figura 2.7: Throughput predecido de Tomcat en un multiprocesador.

anteriormente, la simulación nos ofrece un "*what-if*" para muchos escenarios.

Escalabilidad de la aplicación

Imaginemos que necesitamos conocer el comportamiento de nuestra aplicación en un sistema con más de 1 CPU. Tenemos la información para 1 procesador y hemos obtenido los parámetros de la ejecución. En la figura 2.7 mostramos el rendimiento obtenido usando de 1 a 8 CPUs. Ya que tenemos un sistema de 4 procesadores, podemos comprobar el comportamiento y comparar el resultado obtenido mediante predicción en la figura 2.8. Observamos como el comportamiento es similar, la validación del modelo en un entorno más complejo se puede observar en el capítulo 7.

Para finalizar este ejemplo, podemos ver que usando 8 CPUs prácticamente ha escalado correctamente. Esto viene provocado porque no hemos modelado las CPUs ni la contención del sistema operativo; si quisieramos encontrar el rendimiento para un número mayor de CPUs, deberíamos modificar el modelo introduciendo *threads* para reproducir el comportamiento exacto como veremos más adelante.

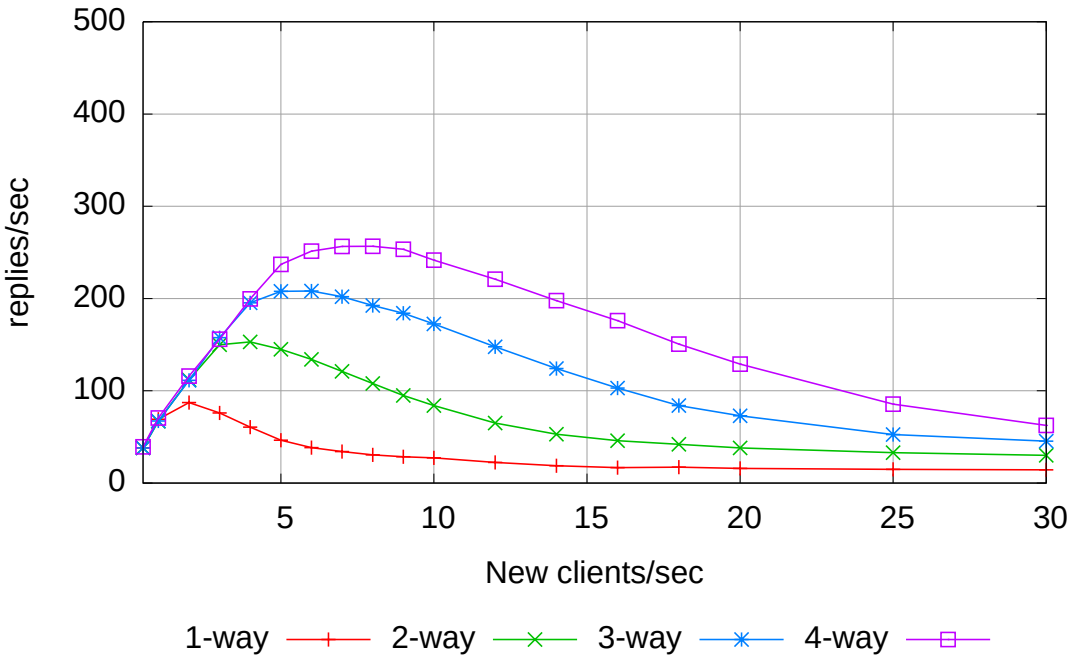


Figura 2.8: Throughput de Tomcat en un sistema experimental con multiprocesadores.

Simulando hipótesis

Las simulaciones pueden clarificar algunas propuestas y decirnos si son opciones correctas. Por ejemplo: hemos testeado algunas propuestas para incrementar el rendimiento en un servidor que no utiliza seguridad (*SSL*); hemos probado el comportamiento normal contra algunas modificaciones, como asignar *1 HTTPProcessor a 1 cliente*, *priorizar la cola de backlog* o asignar *más HTTPProcessors a los clientes más antiguos* (o a los nuevos). 1 cliente - 1 *HTTPProcessor* reduce el rendimiento del servidor mientras las otras lo mantienen, de este modo podemos descartar esta hipótesis (o ponerla al final de nuestras prioridades). De igual modo, debemos pensar constantemente en las causas de esta pérdida de rendimiento, para asegurarnos que el modelo sigue siendo válido. Una asignación 1 a 1 de un *workload* con peticiones simultáneas (*burst*), como pueden ser imágenes que son pedidas a la vez, produce congestión: no se pueden resolver en paralelo.

Si ejecutamos el mismo test midiendo el número de sesiones finalizadas, encontramos que hay una propuesta que hace mejorar el rendimiento como podemos ver en la figura 2.9; de este modo dotamos de prioridad a la cola de conexiones y dejamos que Tomcat trabaje con los clientes antiguos (típicamente finalizaran antes su sesión que los nuevos).

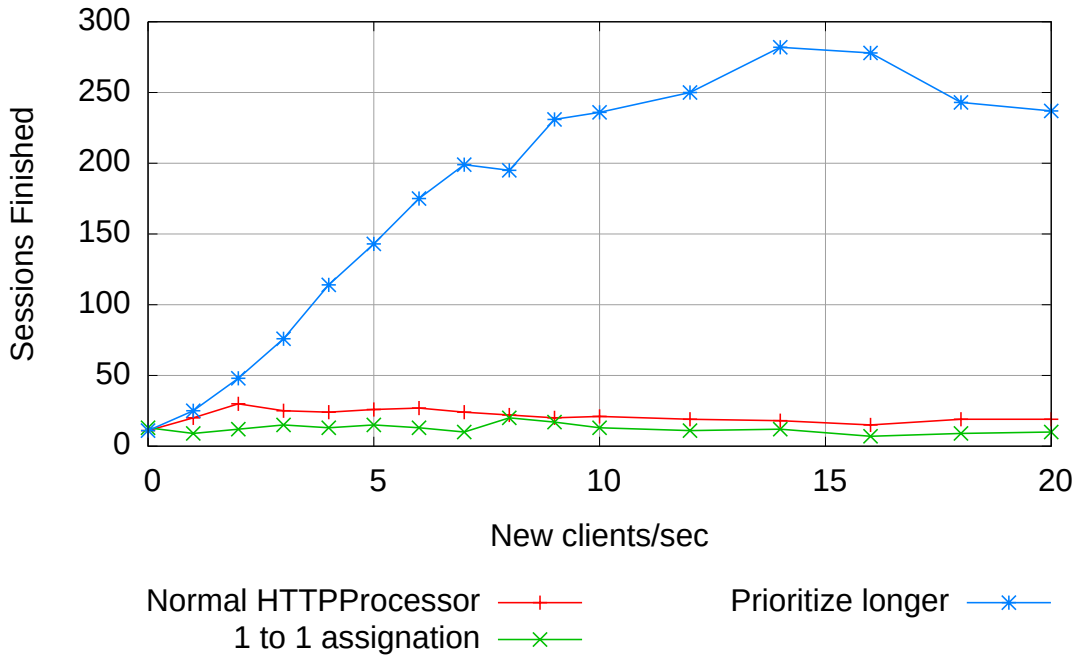


Figura 2.9: Probando diferentes propuestas para incrementar el rendimiento, métrica de sesiones finalizadas (Predicción).

De igual modo debemos pensar si el resultado es correcto: si los clientes antiguos finalizan antes, se verán menos afectados por los *timeouts*. Así que probablemente finalicen su sesión pronto y no se vean afectados por una sobrecarga del sistema.

Utilizando la simulación de la propuesta anterior hemos obtenido una mejora muy elevada del rendimiento en esta métrica (sesiones finalizadas). De este modo esta propuesta debería pasar a ser prioritaria para implementarla en un sistema real ya que tenemos argumentos para pensar que es buena.

Cambios en los parámetros de sistema

Hay una gran cantidad de parámetros que podemos modificar; testear todos puede consumir mucho tiempo (y requerir una gran cantidad de recursos). Con la ayuda de la simulación podemos testearlos de manera rápida siempre que tengamos en cuenta el parámetro en el modelado.

Uno de los parámetros que podemos modificar para conseguir más rendimiento es el tamaño del *Thread Pool* se refiere al número de *HTTPProcessors* que pueden crearse o

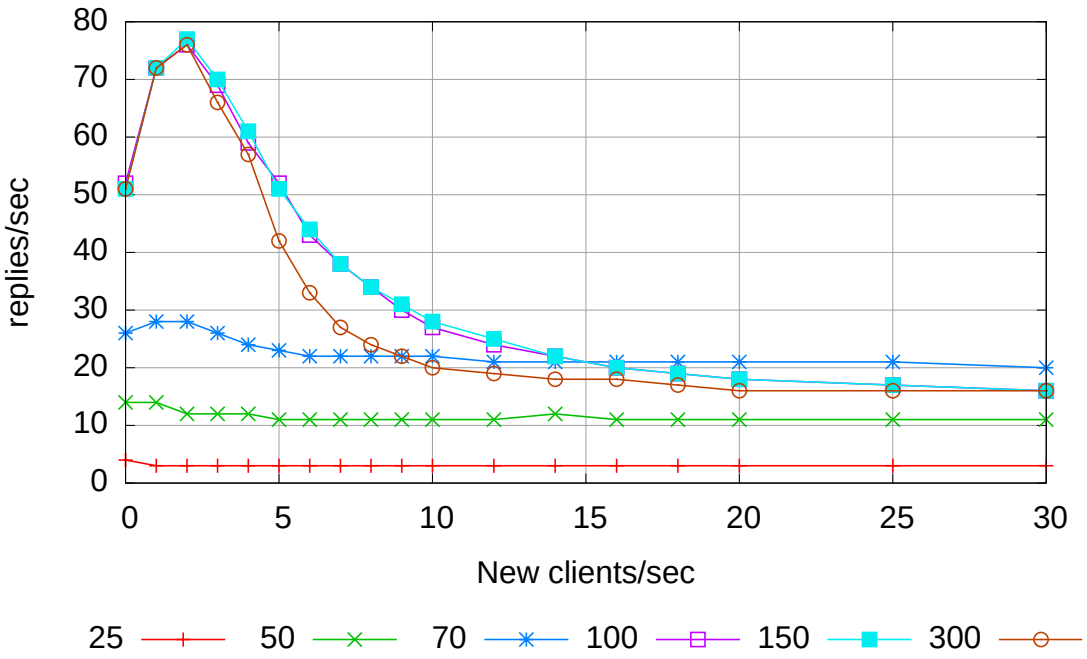


Figura 2.10: Modificación del tamaño de pool en 1 CPU, predicción.

mantenerse para procesar peticiones. Hemos ejecutado una predicción con diferentes medidas del pool de *thread*: queremos demostrar que la modificación del número de *HTTP-Processors* no tiene ningún beneficio en el *throughput*, así que no necesita ser modificado. Podemos ver los resultados en la figura 2.10 con diferentes medidas del *Pool*, con la simulación podríamos realizar estadísticas del uso del *Pool* que serían complicadas de realizar en el sistema real. Los resultados de la figura 2.10 coinciden con la suposición encontrada en otros experimentos realizados sobre el sistema real. Este hecho abre distintas puertas para poder usar esta simulación con el fin de evaluar distintas políticas de *QoS* con la modificación de parámetros en tiempo real.

Extracción de datos usando simulación

Con las simulaciones podemos adaptar el modelo para ayudarnos a obtener diferentes tipos de resultados. Estas pruebas o datos podrían costar mucho o incluso no poderse obtener en el sistema experimental. Las simulaciones nos dan la flexibilidad necesaria para añadir variables y contar eventos, como puede ser el máximo número de sesiones abiertas en un instante (*Sessions In-fly*). Con esta variable podemos ver como el control de admisión afecta el número de sesiones abiertas. Esta clase de resultados estadísticos se pueden

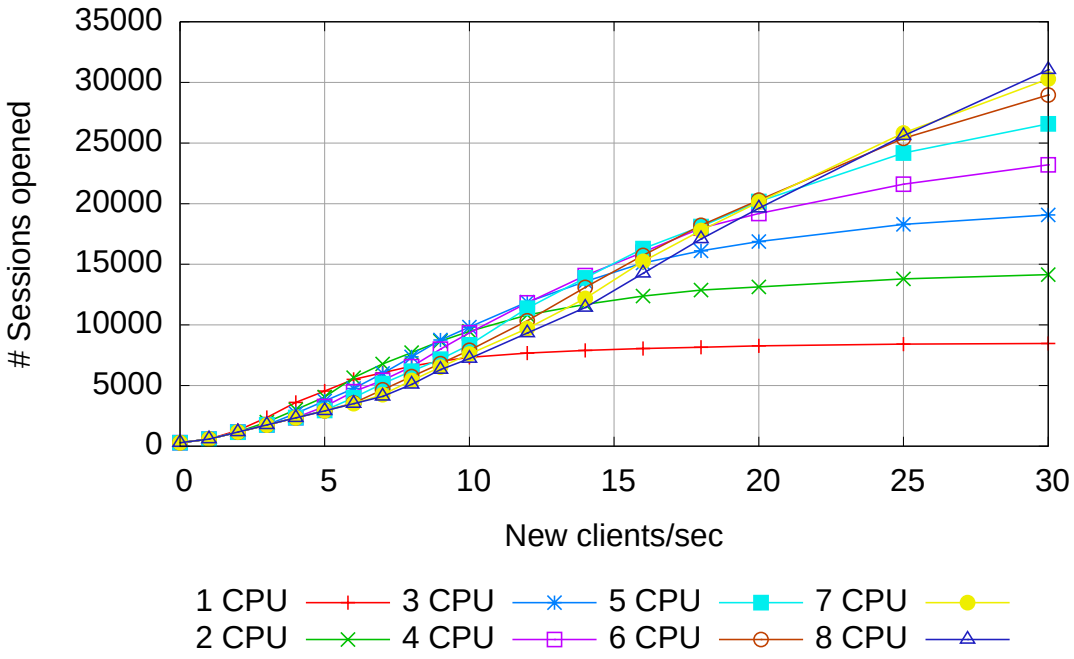


Figura 2.11: Predicción del máximo número de sesiones abiertas en un sistema multiprocesador sin control de admisión.

obtener con gran facilidad añadiendo código al modelo. Un vistazo a la gráfica de sesiones abiertas sin control de admisión (figura 2.11) y con control de admisión (figura 2.12) nos muestra que el numero de sesiones abiertas en el sistema están limitadas a un número fijo si usamos control de admisión y cómo escala si hay más de una CPU.

2.5. Conclusiones

En este capítulo hemos mostrado las bases para conseguir modelar y simular un sistema complejo, como es un servidor de aplicaciones transaccionales, con un modelo ligero para reproducir y crear nuevos resultados. Las pruebas sobre un sistema experimental necesitan mucho tiempo para ejecutarse y producir resultados; además, se necesitan multiprocesadores de forma exclusiva y en tiempo real, ya que requieren una interacción cliente-servidor. Muchas veces estos recursos pueden llegar a ser imposibles de conseguir o simplemente inexistentes.

La simulación nos ofrece información útil que puede usarse para seleccionar una

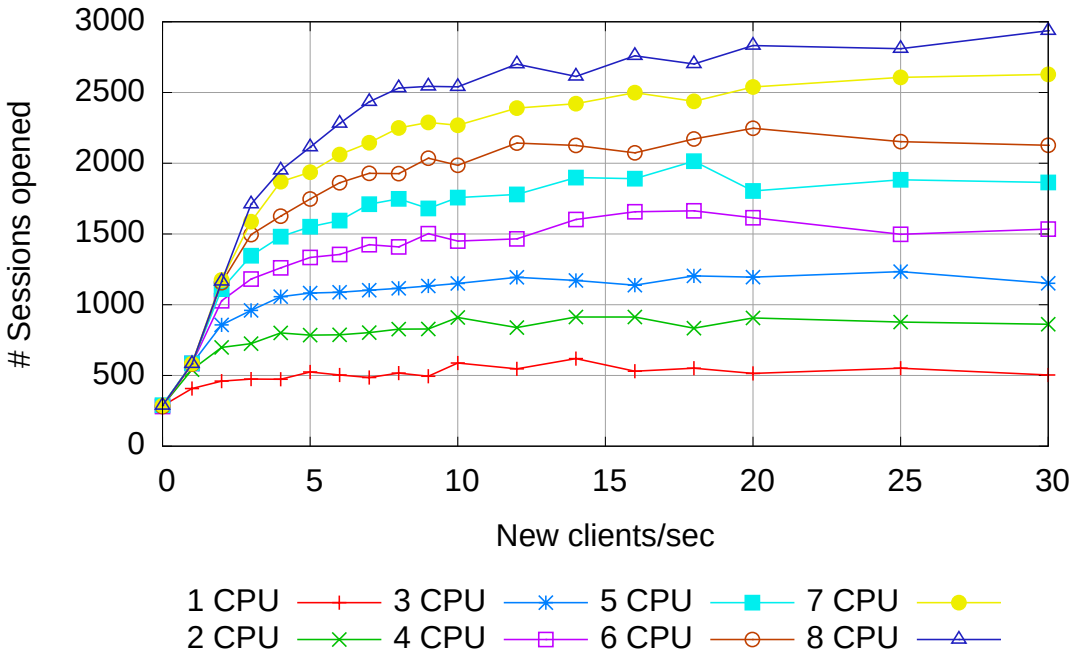


Figura 2.12: Predicción del máximo número de sesiones abiertas en un sistema con control de admisión.

hipótesis sobre otra usando los recursos necesarios para la simulación. Nuestro modelo usa *OMNeT++* para recrear el sistema con el detalle que necesitamos, y predice de manera satisfactoria el comportamiento y el perfil de los resultados del sistema real. Las propuestas analíticas, aunque son buenas herramientas, no son válidas en este escenario. Por otro lado, los resultados obtenidos animan a usar la simulación, pero se necesita tener precaución con los resultados. En nuestro modelo, por ejemplo, no se pueden tomar como buenos los resultados obtenidos con más de 8 procesadores, puesto que en el sistema real habría una gran contención que no se refleja en el modelo creado; para ello necesitaríamos ampliar el modelo y darle más detalle. Este detalle extra lo hemos utilizado en el capítulo 6 para conseguir la simulación del entorno cuando tenemos dinamismo en la asignación de recursos y en la carga del sistema. Como comprobaremos, la fiabilidad de los resultados de la simulación son elevados.

Desde el punto de vista de un investigador, poder testear las hipótesis con pocos recursos además de consumir poco tiempo, nos ofrece una herramienta para ayudarnos en nuestro trabajo. Podemos perder mucho tiempo probando una hipótesis en el sistema real que nos ofrecerá finalmente unos malos resultados; con la simulación lo podemos detectar antes de que suceda. Finalmente, es importante recordar que este tipo de simulación ha

de ser usada para lo que se ha creado, de este modo conseguiremos tiempos de ejecución muy bajos y produciremos predicciones más exactas. En nuestro caso (por ejemplo) no hemos modelado los bloques o comportamientos necesarios que afectan al tiempo de respuesta, por lo tanto no deberíamos usar el modelo para predecir esta métrica. Podemos encontrar en [Chao et al., 2003] propuestas similares en un entorno distinto.

Este capítulo está basado en las siguientes publicaciones:

- Nou, R., Guitart, J., Beltran, V., Carrera, D., Montero, L., Torres, J., and Ayguadé, E. (2005). **Simulating Complex Systems with a Low-Detail Model**. In XVI Jornadas de Paralelismo, pages 201–308, Granada, Spain.
- Nou, R., Guitart, J., Carrera, D., and Torres, J. (2006a). **Experiences with simulations - a light and fast model for secure web applications**. In ICPADS (1), pages 177–186. IEEE Computer Society.
- Nou, R., Guitart, J., and Torres, J. (2006b). **Simulating and modeling secure web applications**. In Alexandrov, V. N., van Albada, G. D., Sloot, P. M. A., and Dongarra, J., editors, International Conference on Computational Science (1), volume 3991 of Lecture Notes in Computer Science, pages 84–91. Springer.

Capítulo 3

Estudio y mejora de middleware Grid

En este capítulo realizaremos un estudio de las plataformas *Grid* y sus problemas de rendimiento; analizaremos el comportamiento de un nodo *Grid*, utilizando el *middleware* Globus Toolkit, realizando la monitorización y instrumentación usando *BSC-MF* (ampliación realizada de *JIS*) y plantearemos tres soluciones incrementales para obtener un sistema autónomo y auto-gestionable que pueda mantener la carga de un nodo de acceso para una *Grid*. Para finalizar, comentaremos los puntos donde podemos introducir la predicción para mejorar estos sistemas autónomos.

3.1. Introducción a Grid

La tecnología *Grid* ha permitido realizar clusters de una gran variedad de recursos y servicios distribuidos geográficamente. Habiéndose establecido como un paradigma de la computación para la ingeniería y la ciencia, la computación *Grid* va a convertirse en el paradigma de la futura computación para empresas y la integración de sistemas distribuidos [Foster et al., 2002b, OGF, 2008].

Las distintas capas del *middleware* que hacen de soporte para las aplicaciones *Grid* tienen más importancia en el rendimiento de dichas aplicaciones y tienen un impacto inmediato en la calidad de servicio (*QoS*). Dada la complejidad de estas capas, son más difíciles de analizar [Carrera et al., 2005].

La infraestructura *Grid* nos ofrece algunas soluciones integradas para crear aplicaciones distribuidas. Se han hecho esfuerzos para consolidar estas soluciones como iniciativas de software libre o código abierto, financiadas por la administración, o incluso soluciones

propietarias [Parashar and Browne, 2005]. Una de estas iniciativas es la *Globus Alliance* [Sotomayor and Childers, 2005], la cual tiene un proyecto junto determinados centros de investigación, universidades y otras organizaciones para crear tecnologías que utilizan *Grid*. Entre sus contribuciones destacamos la especificación de *Open Grid Services Architecture* [Foster et al., 2002a] y de *Open Grid Services Infrastructure* [GridForum, 2006].

Estas especificaciones definen una arquitectura global y una serie de interficies. La primera implementación de la *OGSI* fue adoptada en Globus Toolkit (primeras versiones) y la nueva especificación *WSRF* en Globus Toolkit 4.0. Utilizando este conjunto de recursos y servicios coordinados, seguros y flexibles, la computación *Grid* ofrece un gran número de ventajas a los entornos empresariales, como por ejemplo: mayor respuesta a sus cambios, mejor utilización y rendimiento a nivel servicio y unos menores costes de administración [OGF, 2008]. A pesar de esto, al incrementar su entrada en el ámbito comercial, el rendimiento y la *QoS* (Calidad de servicio) se están convirtiendo en las mayores preocupaciones. La complejidad inherente del sistema, su heterogeneidad y el dinamismo de la computación en los entornos *Grid* ofrecen dificultades para gestionar su capacidad y asegurar que los requerimientos en *QoS* son satisfechos de forma continua.

Las Grids empresariales están compuestas normalmente de componentes heterogéneos en distintos dominios administrativos y están altamente distribuidos bajo entornos dinámicos. Los mecanismos de asignación de recursos y de planificación de trabajos, tanto a nivel global como a nivel local, juegan un papel crítico en el rendimiento y la disponibilidad de las aplicaciones *Grid* [Menascé and Casalicchio, 2004]. Para prevenir la gestión de recursos y la no disponibilidad es esencial que se utilicen mecanismos de control de admisión por los gestores de recursos locales.

Resumiendo, aún existen muchos problemas a diferentes niveles para conseguir una infraestructura *Grid* global [Parashar and Lee, 2005]; uno de ellos es el rendimiento del *middleware*. Por ejemplo, cuando un nodo recibe trabajos, la sobrecarga introducida por el *middleware* puede tener un impacto importante en toda la ejecución. Procesar las conexiones, parsear las peticiones (*XML*) y trabajar con los protocolos de los servicios Web (*Web Services*) son tareas que requieren una gran cantidad de recursos.

3.2. Globus Toolkit 4

El principal problema que debe resolver *Grid* fue definido en [Foster et al., 2001] como la *"coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations"*. El objetivo de estos entornos virtuales dinámicos, *virtual organi-*

zations, es permitir que grupos dispares (organizaciones o individuos) compartan recursos de una manera controlada y que se beneficien de elementos de *QoS*, como pueden ser el rendimiento mejorado, la disponibilidad y la gestión de recursos, y de su utilización de manera óptima, seguridad, etc.

En [Foster et al., 2001] los autores presentan una arquitectura *Grid* extensible de 5 capas: *fabric*, *connectivity*, *resource*, *collective*, *application*. La capa *fabric* ofrece los recursos con acceso compartido utilizando los protocolos *Grid*; por ejemplo, recursos computacionales, espacio de almacenamiento, recursos de red o sensores. Un recurso puede ser también una entidad lógica, como un clúster *HPC* o un sistema de almacenamiento distribuido. La capa *connectivity* ofrece el núcleo de los protocolos de comunicación y de seguridad (autenticación y control de acceso) requeridos para las transacciones *Grid*. La capa *resource* define los protocolos, *APIs* y *SDK* para la negociación segura, la iniciación, la monitorización, el control y el pago de las operaciones para poder compartir los recursos individuales. La capa *collective* ofrece protocolos y servicios que coordinan las interacciones entre colecciones de recursos. Finalmente, la capa *application* engloba las aplicaciones del usuario que operan en el entorno *Grid*.

Una de las implementaciones de esta arquitectura multicapa la podemos encontrar en Globus Toolkit (GT) [Foster, 2005]: una arquitectura de software abierto, y un conjunto de servicios y librerías que soportan Grids y aplicaciones *Grid*. El Toolkit resuelve cuestiones de seguridad, gestión de recursos, gestión de datos, comunicación, detección de fallos y portabilidad, entre otros. Globus Toolkit se usa en multitud de entornos y en los mayores proyectos *Grid* del mundo. La última implementación de GT, la versión 4, basada en Web Services, mejora la anterior en términos de usabilidad, rendimiento, estándares y funcionalidad. En los siguientes párrafos mostramos algunos de los protocolos que ofrece Globus para la gestión de recursos (reserva y asignación de recursos) y *QoS*. Globus se utiliza en multitud de proyectos y organizaciones, y es una implementación de referencia.

El Toolkit incluye los servicios y librerías para monitorizar recursos, descubrirlos y utilizarlos. Además, incluye seguridad y herramientas de transferencia de ficheros. Sus servicios básicos, interficies y protocolos permiten a los usuarios el acceso remoto de recursos como si estos estuvieran en su propia máquina, mientras se preserva el control local de cada recurso para definir quién puede usar los recursos y cuándo.

La arquitectura de gestión de recursos de Globus [Foster and Kesselman, 2003] resuelve el problema de la calidad de servicio ofreciendo un acceso dedicado a colecciones de computadoras en sistemas distribuidos heterogéneos. La arquitectura consiste en tres componentes principales: un servicio de información, un gestor de recursos locales y varios tipos de agentes de asignación que implementan estrategias para descubrir y asignar los recursos necesarios para satisfacer los requerimientos de *QoS* de una aplicación.

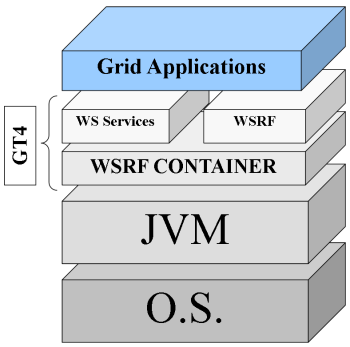


Figura 3.1: Estructura de un sistema simple con un middleware Grid.

Una aplicación que desee realizar una tarea le pasa una descripción a un agente de asignación, este agente usa una combinación de consultas al servicio de información, heurísticas y conocimiento específico de la aplicación para mapear las necesidades de la aplicación a los recursos, descubrirlos y finalmente asignarlos.

El software está compuesto de 5 componentes que ofrecen: seguridad, administración de datos, administración de las tareas, servicios de información y un entorno de ejecución unificado. Nos centraremos sólo en dos componentes: *WS-GRAM* [Alliance, 2005b] es una implementación de un protocolo de comunicación (cumple el *WSRF*): ofrece un conjunto de planificadores de recursos locales y nos ofrece la administración del entorno de ejecución; y finalmente, *Java WS-Core* [Alliance, 2005a], que mantiene el entorno de ejecución y ofrece un conjunto de librerías y herramientas (Java) que permite que los servicios web de GT4 sean independientes de la plataforma.

Podemos observar en la figura 3.1 y en la figura 3.2 una visión esquemática de un sistema GT4. El contenedor de *GT4* utiliza dos grupos de threads. El primero es el *ServiceDispatcher* (sólo un thread), el cual atiende las conexiones y envía las peticiones a la *RequestQueue*. El segundo grupo, llamado *ServiceThreads*, recibe las peticiones desde la RequestQueue y las procesa. El número de estos threads varía dinámicamente dependiendo de las necesidades del sistema. El proceso utiliza seguridad (*SSL*) y procesado de mensajes *SOAP* realizado a través de *Axis*.

El servicio *GRAM* permite la ejecución remota del trabajo. En nuestro caso no usamos el *RFT* (Remote File Transfer), ya que no enviamos ningún fichero, de modo que podemos dividir el servicio en dos partes abstractas: la primera se encarga del nivel Java preparando la ejecución del trabajo, y la segunda se encarga de la ejecución del trabajo e interactúa con él. En nuestro caso utilizamos por defecto el planificador de Globus.

WS-GRAM son básicamente dos servicios web: *ManagedJobFactoryService* y *Mana-*

gedJobService. El primero permite la creación de un trabajo (conjunto de descriptores) cuando queremos enviar un trabajo a una maquina remota; el segundo es un servicio al que llamamos para iniciar, parar o monitorizar el trabajo. Esto sería lo que vemos desde el lado del cliente. Si miramos dentro de *WS-GRAM* podemos ver cómo actúa sobre los trabajos en la máquina remota usando la *RunQueues*. Cada vez que llamamos a *ManagedJobFactoryService*, éste crea un recurso asociado a nuestro trabajo y lo envía a una de las *RunQueues* (conjunto de threads). Cuando el recurso está en ellas, va a través de distintos estados (que conforman el ciclo de vida de un trabajo [Alliance, 2005c]) hasta que se envía al planificador local. Cuando ha finalizado el trabajo, éste se envía al último estado y finalmente se envía la respuesta al cliente. Podemos observar la navegación de una petición por el nodo en el diagrama de la figura 3.2.

En detalle, el flujo de control de Globus al procesar trabajos es el siguiente. Cuando un cliente envía un trabajo a un servidor Globus (usando la interfície *globusrun-ws* en su modo de proceso no batch), el trabajo es adquirido por un *ServiceThread* que realiza un procesado de seguridad *SSL*. Después del handshake, la petición *SOAP* se analiza para preparar el trabajo para su ejecución. El trabajo se inicia y va a través de distintas fases, como muestra la figura 3.2. En cada fase el trabajo se introduce en una *RunQueue* y se procesa por un conjunto de threads. La fase más importante es cuando el trabajo se ejecuta a nivel del sistema operativo *submit*, que se realiza usando un hilo de ejecución separado (creado por Globus). Después de la ejecución, el trabajo va a través de algunas fases de limpieza para pasar otra vez a los *ServiceThreads* y generar la respuesta *SOAP* para enviarla al cliente.

Escogemos *Globus Toolkit* para nuestros experimentos por ser un entorno ampliamente utilizado (también en la *UPC* y el *BSC*) y de código abierto.

3.3. Rendimiento de Globus Toolkit 4

En esta sección presentaremos un ejemplo en el que se muestra como Globus Toolkit 4 (*GT4*) [Sotomayor and Childers, 2005] puede llegar a la indisponibilidad bajo algunas condiciones de sobrecarga. La sobrecarga se realiza en lo que sería un nodo de acceso de GT4.

Podemos ver el resultado de un experimento en la gráfica de la figura 3.3 donde tenemos una serie de clientes enviando muchos trabajos al mismo tiempo a un nodo simple. Aunque este test no es realista (en cuanto al tipo de trabajo que se envía), es suficiente para mostrar la efectividad de administrar los recursos (como veremos en sucesivos capítulos) y cómo la sobrecarga en la gestión del trabajo impide el correcto funcionamiento del sis-

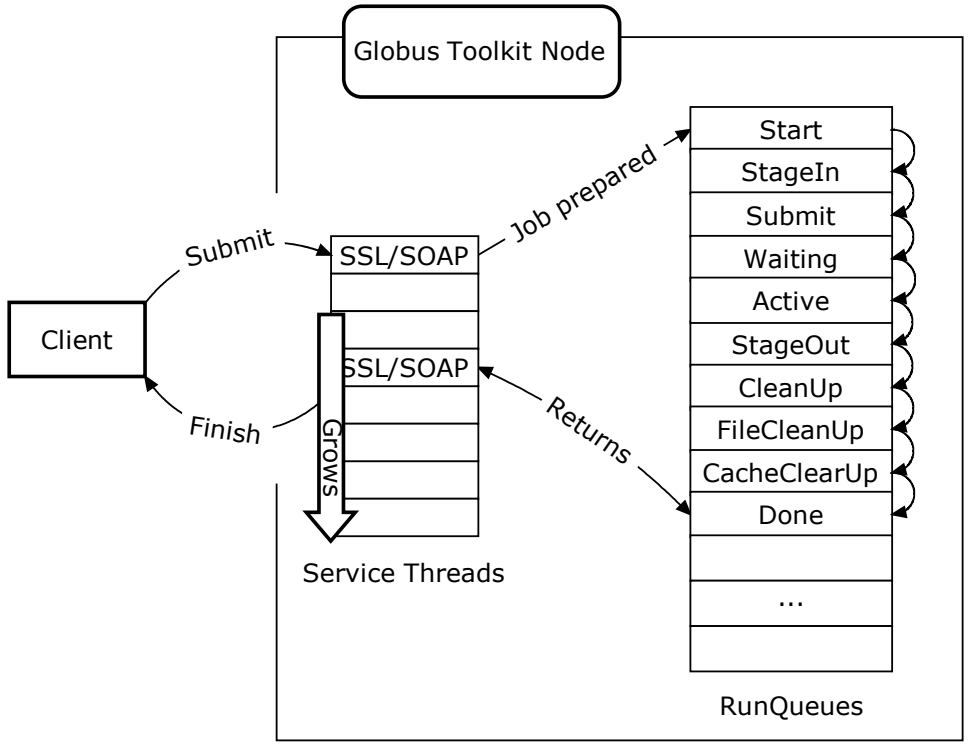


Figura 3.2: Diagrama simplificado de la navegación por el interior de un nodo GT4 de un trabajo.

tema. Cada trabajo que enviamos consume CPU; enviamos los trabajos al principio de la ejecución y creamos el número deseado de trabajos simultáneos según el eje x para simplificar las pruebas y el análisis. Hemos de destacar que las soluciones propuestas en los sucesivos capítulos funcionarían bajo cualquier comportamiento o tipo de carga. El tiempo de la prueba finaliza cuando $t = \infty$, es decir, cuando todos los trabajos enviados han finalizado o se han perdido.

En el lado del servidor, tenemos un contenedor Java de Globus Toolkit 4 que acepta las conexiones con una conexión segura (default) y envía las peticiones a los servicios correspondientes. Como cada conexión está asignada a un thread, el número de threads del servidor (*ServiceThreads*) se adapta a la carga del sistema.

Como podemos ver en la gráfica de la figura 3.3, cuando el número de trabajos concurrentes sobrecarga el servidor se llega a un punto donde no se obtiene servicio; el servidor está demasiado ocupado procesando los trabajos en ejecución. Esto ocurre no sólo por el

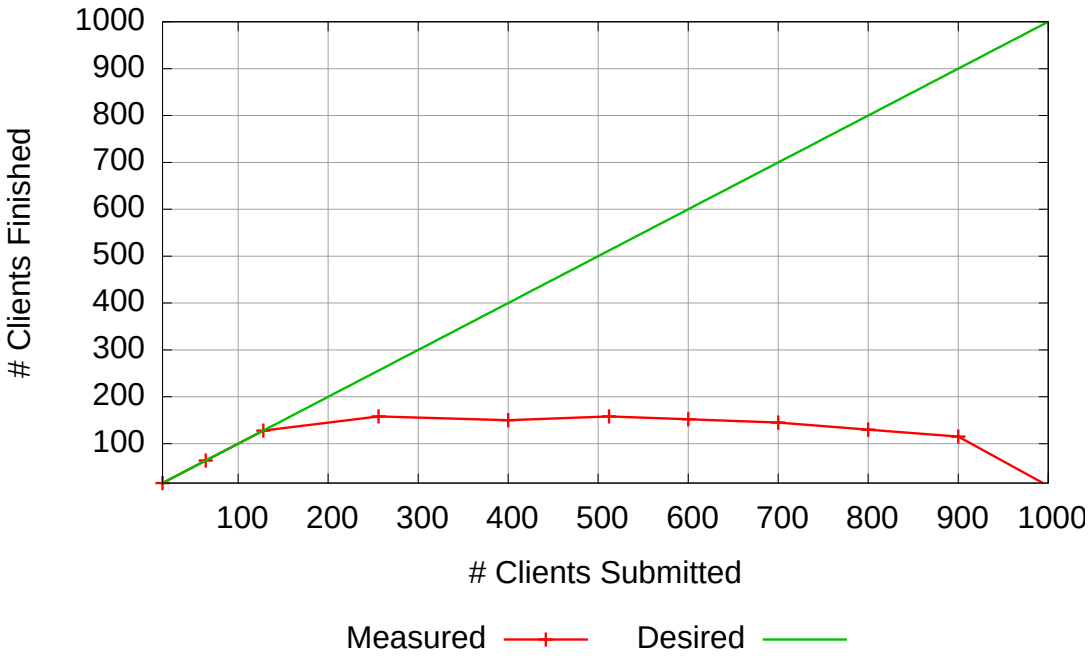


Figura 3.3: Trabajos completados en un nodo de acceso GT4.

consumo de procesador de cada uno (que sólo acelera el proceso) sino por las conexiones seguras entrantes. Por ello, habilitar una política correcta de administración de recursos podría resolver este problema.

3.4. Análisis y monitorización de un nodo Globus

En este apartado introduciremos *BSC-MF* nuestro entorno de monitorización para aplicaciones Java, así como un recorrido por el estado del arte de las distintas herramientas de monitorización disponibles para este tipo de entornos.

3.4.1. Introducción

Las aplicaciones Java se ejecutan, cada vez más, en entornos complejos. Esto provoca que cada vez sea más difícil optimizar los parámetros de instalación en los entornos de producción. Pensando en esto, *BSC-MF* ha sido desarrollado para monitorizar y medir el rendimiento de las aplicaciones Java junto con el sistema donde se ejecuta, y unir todos

los datos en una forma utilizable para su análisis. El uso principal es usar *BSC-MF* para detectar cualquier ineficiencia o problemas de contención que tenga el sistema (como un todo) y utilizarlo posteriormente para probar diferentes alternativas o cambios que puedan resolver el problema.

Las soluciones pueden pasar por reprogramar rutinas, cambiar configuraciones o cambiar la configuración del sistema (entre otras). Para dar sentido a los datos obtenidos y poder analizar la aplicación, la traza ha de ser lo más completa posible y contener detalles relevantes para los recursos del sistema (red, uso de CPU y uso de disco).

Este uso es más amplio que los demás profilers o herramientas de monitorización para Java, ya que se basan en la aplicación y observan las cosas desde esa perspectiva. Conceptos como el uso de memoria, el tiempo de proceso de los threads, las llamadas de sincronización de los hilos de ejecución, la creación de objetos o el número de invocaciones de un método son factores importantes para la aplicación. Mientras todas las herramientas Java son útiles en la fase de desarrollo de la aplicación, no son muy útiles durante la ejecución del sistema en el "mundo real". En un servidor de aplicaciones complejo hay demasiados procesos que pueden crear cuellos de botella, de modo que esta vista es demasiado simple e inadecuada para observar si las aplicaciones funcionan correctamente.

3.4.2. BSC-Monitoring Framework

BSC-MF consiste en tres partes diferenciadas que trabajan juntas para conseguir nuestro objetivo. La primera parte se llama Java Instrumentation Suite (*JIS*) y contiene herramientas para tracear aplicaciones Java que se ejecutan en una máquina virtual, además de herramientas de parseo y creación para distintos ficheros de trazas. *LTT* (*Linux Trace Toolkit*) [Yaghmour and Dagenais, 2000] es una conocida herramienta de sistema para monitorizar procesos que se ejecuten bajo un sistema operativo Linux. Finalmente, Paraver [Jost et al., 2003] es una herramienta flexible de visualización de trazas y una potente herramienta de análisis seleccionada como la mejor manera de visualizar las trazas finales. Ver diagrama 3.5 para una descripción de cómo funciona el sistema.

Además de los objetivos descritos arriba, se requiere que tenga un overhead muy bajo y que la aplicación pueda ejecutarse en el entorno final, lo que nos ha llevado a tener dos versiones: una para máquinas virtuales 5.0 y superiores, y otra para las inferiores que utiliza la antigua versión creada con *JIS*.

Algorithm 3.4.1: M(y)

$$CtClass.insertBefore(\text{"}...UserEvents.GenerateUserEvent((long)\text{"} + classID + \text{"},(long)\text{"} + methodID+);\text{"});$$
$$myCtClass.insertAfter(\text{"}...UserEvents.GenerateUserEvent((long)\text{"} + classID + \text{"},(long)0);\text{"},true);$$

Figura 3.4: Ejemplo de código en javassist.

3.4.3. Detalles de JIS

Para monitorizar cualquier aplicación Java usando *JIS*, éste necesita ejecutarse usando una clase de *Bootstrap*. Esta clase sustituye al *ClassLoader*, pudiendo modificar las clases cuando se cargan, de manera que puedan tracearse. Para prevenir que *JIS* monitorice todas las clases, añadiendo sobrecarga innecesaria, se incluye un fichero de configuración (filter.xml) en el que se colocan las clases y los métodos de interés. La librería *jdom* se utiliza para parsear el fichero.

Cuando el *Classloader* carga las clases, que están marcadas, son procesadas por otra clase llamada *InstrumentationModule* que las preparara para tracearlas. Como su nombre sugiere, se introduce nuestro *byte code* en puntos clave (por ejemplo cuando se entra o se sale de los métodos) para que se llame a los mecanismos de traceo de *JIS*.

Javassist [CHIBA, 1998], una API para modificar el *byte code*, se utiliza para inyectar nuestro código. Los dos métodos importantes que usamos son *insertBefore()* y *insertAfter()* para insertar código al principio o al final del método, como podemos ver en la figura 3.4.

La clase *myCtClass* es una instancia del objeto *CtClass* que hace de envoltorio de cualquier clase Java. El método *GenerateUserEvent* es parte de nuestro código JIS que usamos para enviar una llamada a JNI (*Java Native Interface*) hacia la librería de JIS. Tiene dos parámetros: *classID* y *methodID*. El primero es un número interno que representa la clase y se utiliza para referirse a la clase en la traza Paraver; finalmente *methodID* representa un método. Un valor de 0 en el *methodID* representa la finalización del método llamado.

El *byte code* inyectado en la aplicación con la ayuda de javassist hace llamadas a la librería nativa usando *JNI*. La interfície *JNI* permite a Java usar código nativo, pudiendo acelerar y ganar precisión en la generación de las trazas. En muchos casos el reloj que

usa Java tiene una precisión de unos 10 ms. Finalmente, se guarda el tiempo de inicio de los eventos, así como el estado de los threads (IDLE, RUNNING, STOPPED). Esto no es independiente del sistema, por ello la actual versión sólo funciona en Linux.

La interficie de *JVMPI* [user manual, 2008] es una forma muy eficaz de obtener información de una máquina virtual Java y de la aplicación que está ejecutando. Puede usarse para obtener una gran cantidad de información, pero esto produce un incremento de la sobrecarga del sistema, impidiendo la obtención de resultados reales. Por esta razón se ha decidido utilizar sólo *JVMPI* para obtener los eventos de creación o destrucción y de inicialización y finalización de la *JVM*. Estos eventos son los siguientes: JVMPI_EVENT_JVM_INIT_DONE, SHUT_DOWN, THREAD_START y THREAD_END.

Cuando la librería nativa *JIS* se inicia, una de las primeras cosas que hace es arrancar el *Linux Trace Toolkit* (*LTT*). *LTT* monitoriza los procesos del sistema en un sistema operativo Linux y requiere que se parchee el kernel para instrumentarlo. *LTT* tiene su formato binario propio y lo crea para cada una de las CPUs del sistema. Existe una versión nueva llamada *LTTng* (que tiene un formato distinto de traza) para los nuevos kernels, ya que la antigua versión ya no se desarrolla.

Como podemos ver en el diagrama de la figura 3.5, al final del periodo de traceado hay una serie de archivos en el directorio. Las trazas generadas por *JIS* tienen un descriptor de fichero *.jis*, y hay una creada para cada thread de la aplicación. Tal como hemos comentado en el párrafo anterior, hay un fichero binario de *LTT* para cada *CPU*. Para combinar todos los elementos juntos utilizamos un conversor para generar una traza Paraver.

Dejamos a Paraver la visualización y el análisis de la traza final. Paraver nos puede ofrecer un análisis cuantitativo detallado del rendimiento de una aplicación; es un sistema muy flexible con un elevado número de opciones que pueden usarse para obtener vistas específicas de la traza. Una de sus funciones más importantes es la capacidad para trabajar con trazas muy largas, como las que generamos en nuestros experimentos.

3.4.4. Coste de la instrumentación

En toda instrumentación se introduce consumo de CPU o esperas (*overheads*) que no existen en el sistema real; con *BSC-MF* se han intentado mantener en un nivel mínimo para que no produzcan interferencias en el comportamiento normal de una aplicación. El hecho de que los eventos que no son necesarios no sean monitorizados, y que el fichero de configuración (*filter.xml*) nos permita especificar las clases y los métodos que necesitamos, nos ayuda a reducir la sobrecarga del sistema. Finalmente, aunque imposibilita el análisis online sin realizar cambios, las trazas se unifican al final del proceso de captura. Hemos realizado algunos experimentos para obtener el *overhead* del sistema y éste

se ha mantenido entre un 1 % y un 10 % (en aplicaciones normales) del tiempo total de ejecución, suficientemente bajo para no afectar la aplicación.

3.4.5. Desarrollo futuro

Se pueden realizar muchos desarrollos para ampliar el entorno; actualmente (2008) tenemos una versión universal (para cualquier aplicación) que utiliza la nueva interficie *JVMTI* y que se ha ofrecido en dos proyectos externos. Uno de los aspectos importantes del entorno es la instrumentación del código; hay alternativas como las técnicas que utilizan *Aspect Object Programming* (*AOP*).

Tal como hemos dicho, el overhead de *JVMPI* es excesivo, pero el uso del mismo nos puede ser útil para momentos puntuales en los que necesitemos herramientas para, por ejemplo, ver cómo actúa el garbage collector de la máquina virtual en la aplicación.

Finalmente, el punto más importante es utilizar *BSC-MF* en otros entornos. Para poderlo hacer se deberán utilizar los mecanismos mencionados en la sección 3.4.6. Además, la librería nativa debería reescribirse/recompilarse para que las llamadas *JNI* continuaran funcionando.

3.4.6. Estado del arte

Hay un gran número de herramientas de monitorización de aplicaciones Java. Para facilitar la comparación las hemos agrupado en distintos tipos basados en los mecanismos que emplea. Hemos realizado un análisis en profundidad, pero al final ninguno nos ha servido para nuestro propósito, ya que no había ninguno que permitiera un análisis de el sistema entero. Mencionaremos los factores que hemos tenido en cuenta mientras los analizamos en las siguientes subsecciones.

hprof

Empezamos con *hprof* [Sun Microsystems, 2005], una herramienta que ha ofrecido Sun desde la versión 2.0 de SDK de Java para analizar un máquina virtual (*JVM*) mientras se está ejecutando. No ofrece información acerca del sistema operativo, pero puede mostrarnos los threads y realizar un análisis de la memoria (*heap memory*) del *JVM* usando el interfaz de *JVMPI*. El agente hprof necesita empezar cuando una aplicación Java se ha ejecutada para monitorizarla. Esto provoca una gran cantidad de sobrecarga (estimado sobre un factor superior a 20 en algunos casos, como se puede observar en la página web

JIP [JIP, 2006]), así que no es muy aceptable para medir rendimiento. Mejoras en Java 5.0 han provocado diferencias en el rendimiento, a pesar de esto, estas mejoras nos limita a usarlo sólo en las nuevas máquinas virtuales. Esta restricción provocaría la imposibilidad de comparar diferentes máquinas virtuales, como por ejemplo Java 4.0 o Java 5.0.

JVMPI/JVMTI

La interfaz *JVMPI* [user manual, 2008], que usa hprof, se ha usado como base para otras aplicaciones para crear profilers y monitores de rendimiento [WebPage, 2004, Lambert and Podgurski, 1999]. Desde su introducción en la máquina virtual Java 2 se ha etiquetado como experimental, pero se ha usado como el mecanismo básico para extraer datos del funcionamiento interno de una *JVM*, hasta su sustitución (*deprecated*) en Java 5.0 por *JVMTI* [Manual, 2008].

Es una interficie de doble comunicación, de modo que un agente puede controlar el nivel de detalle dado por la máquina virtual y hacer peticiones directamente. La *JVM* notifica al agente de numerosos eventos (el agente necesita suscribirse) cuando estos se producen. Finalmente, el agente se ejecuta al arrancar y puede crear una sobrecarga importante (directamente proporcional al numero de eventos que se ha suscrito el agente). Para minimizar la sobrecarga, sólo usaremos el mecanismo para obtener el evento de creación o destrucción de los diferentes hilos de ejecución (threads), así como la creación o finalización de la *JVM*, ya que son eventos poco repetidos. Sustituyendo *JVMPI* con *JVMTI* en la versión 5.0 de Java obtenemos un mejor rendimiento; hay herramientas que aprovechan esta mejora, como *Java Interactive Profiler* [JIP, 2006]. Los mismos problemas que se aplican a hprof (referentes al rendimiento o la incapacidad de usar máquinas virtuales antiguas) suceden aquí.

Traceado/análisis del sistema

Además del *Linux Trace Toolkit (LTT)* hay otras herramientas que pueden usarse para analizar el sistema operativo. Encontramos a *DTrace* [Microsystems, 2008] de SUN; *ktrace* [FreeBSD, 2008] de FreeBSD, *trace* de IBM para AIX [AIX, 2006] y la herramienta de *Microsoft Event Tracing* for Windows [Corp., 2002]. Estas herramientas son capaces de obtener buenos datos de lo que ocurre en el sistema, y algunas de ellas tienen buenas herramientas de visualización para ver y analizar los resultados [Yaghmour and Dagenais, 2000]. Sin embargo, ninguna de ellas nos ofrece datos de cómo funciona la máquina virtual de Java y no nos son útiles por sí solas. Medir el rendimiento y las políticas de las aplicaciones Java está fuera de su alcance y nos obliga a utilizar una herramienta combinada: una parte que se utilice para los datos de bajo nivel y otra que maneje la aplicación

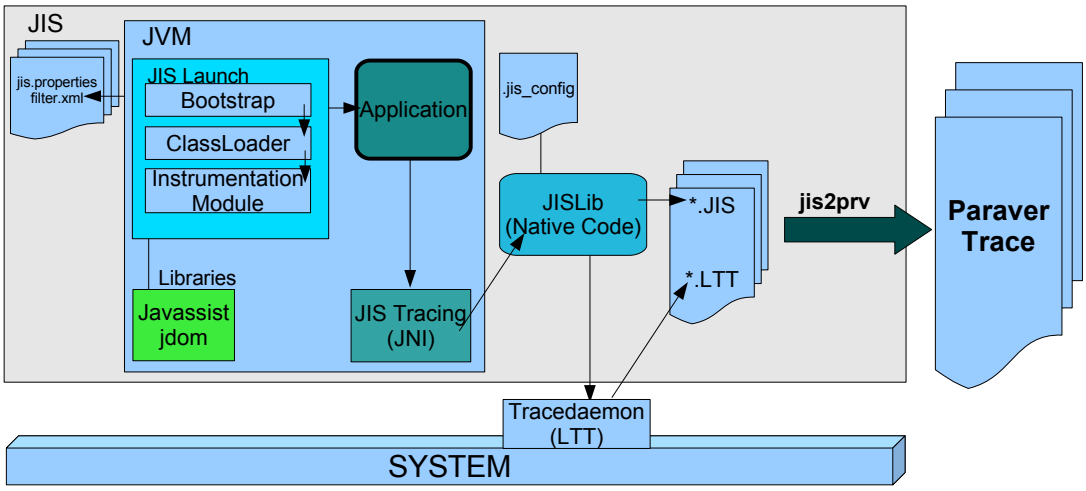


Figura 3.5: Estructura del *BSC-MF*.

Java ejecutándose bajo una máquina virtual. En particular, nosotros hemos usado *LTT* como nuestra herramienta de monitorización por su baja carga y por el hecho de ser *Open source*, haciendo posible modificarlo y cambiarlo como sea necesario. Además, *LTT* tiene un soporte amplio en Linux.

JVMs modificadas

Otra clase de monitorización en Java podemos encontrarlo en *JVM* modificadas. Podemos encontrar ejemplos en *JFluid* de Sun [Dmitriev, 2003], *JRocket* de BEA [WebPage, 2006] y *Tracing VM* [Wolczko, 1999]. Como el rendimiento puede verse muy afectado por el uso de una máquina virtual modificada (algunas veces incluso de manera positiva, JRocket dice ser un 20 % más rápida que la máquina virtual de Sun), se excluyen de nuestra solución para medir el rendimiento de manera correcta y sin grandes alteraciones. Este tipo de JVM tiende a usarse más en el desarrollo de una aplicación, ya que suelen encontrarse como extensión a una IDE. Es el caso de *JFluid*, que forma parte del paquete *Netbeans Profiler*.

Comerciales

Hay algunas propuestas comerciales en este campo. Las principales son: *OptimizeIT* creado por Borland [Borland, 2006], *Performance Management Suite* de Quest [Quest, 2008] y *Introscope* de WilyTech [Wilytech, 2006].

Introscope es una solución que ofrece baja sobrecarga cuando se monitoriza aplicaciones Java y está orientada para clientes que usen J2EE, lo que conlleva un coste (monetario) elevado. Introscope no monitoriza todo el sistema pero soporta algunas de las plataformas más importantes, como pueden ser WebLogic, Oracle o Websphere. Se ofrecen paquetes separados para este tipo de plataformas. Hay un visor común que muestra las distintas estadísticas en tiempo real. Finalmente, para plataformas desconocidas o aplicaciones no comunes el soporte es bajo.

Quest ofrece algo similar a Wily con su *Performance Management Suite*. Le pone énfasis en la monitorización de memoria. Es mejor para monitorizar aplicaciones no específicas, pero est orientado a empresas por su precio.

El coste de usar estas herramientas y el hecho de que ninguna de ellas ofrece una monitorización del sistema a múltiples niveles desaconsejan su uso.

Herramientas combinadas

Recientemente hemos encontrado una herramienta de monitorización para aplicaciones Java que usa tecnología y mecanismos similares a los nuestros. Su nombre es *Java Performance Monitoring Toolkit* (JPMT) y usa *JVMI* con código nativo, una librería externa para instrumentar el *byte code* de Java además de *LTT* para seguir y monitorizar los procesos del sistema operativo. A parte de la referencia [Harkema et al., 2003], no encontramos nada más en su momento, además de no ser descargable o publicado. A partir de los artículos vemos que la información del proceso obtenida mediante *LTT* se limita a la obtención de modelos de la aplicación Java.

Las herramientas que hemos mencionado representan un subconjunto de los numerosos profilers que hemos considerado.

3.5. Mejora básica del rendimiento

3.5.1. Entorno experimental

Nuestro entorno esta montado usando dos máquinas: un cliente y un servidor con características similares. Pentium Xeon a 2.4 Ghz con 2 procesadores y 2 Gb de memoria. Usan un *kernel* 2.6.8 y un 2.6.9, modificado para utilizar *LTT* para el servidor. En el nivel aplicación tenemos una instalación de un cliente GT4 y un servidor utilizando una máquina virtual 1.5 de Java. Aunque el entorno es simple (un solo nodo) nos servirá para ver puntos donde mejorar.

3.5.2. Estudio

Tras hacer pruebas sobre una instalación de *GT4* buscando el número máximo de trabajos que podemos enviar, encontramos que la cantidad de trabajos finalizados decrece cuando incrementamos los trabajos enviados (con un tiempo de espera infinito, de modo que los trabajos no se cancelan por culpa del cliente). Otro problema que detectamos fue el incremento del tiempo de respuesta en el lado del cliente. Para averiguar qué ocurría utilizamos un análisis completo en todos los niveles del sistema, ya que el problema podría encontrarse en la aplicación (*GT4*), la máquina virtual (*JVM*) o el sistema operativo. Cabe destacar el hecho de que los trabajos que se ejecutan lo hacen en el sistema operativo y no encima de la *JVM*.

Para acelerar el proceso de sobrecarga utilizamos un tipo de trabajo que consume tiempo de procesador; son bucles que consumen CPU durante 5 segundos (100 % de carga). Podríamos obtener los mismos resultados, aunque necesitaríamos más clientes, utilizando trabajos que no hicieran nada. En nuestro caso ejecutamos 128 de estos trabajos en paralelo, reduciendo el rendimiento a nivel sistema, provocando la reducción del rendimiento o la eficiencia en el nivel *middleware*, perdiendo trabajos en algunos experimentos (más de 150 trabajos en paralelo). La figura 3.6 muestra una traza Paraver mostrando la ejecución de este ejemplo. El eje horizontal representa el tiempo de ejecución en microsegundos y el eje vertical los threads. En la vista Paraver podemos ver en negro los momentos en los que el thread correspondiente consume procesador, podemos observar como la densidad es mayor en los trabajos (FOR_LOOPS) lo que significa que la utilización de la CPU se centra en los trabajos y se desplaza de las tareas de administración (*middleware*).

Tal como vemos en la traza, el contenedor de Globus sigue intentando ejecutar todos los trabajos tan pronto como sea posible (que es un comportamiento deseado en trabajos sin un consumo intensivo de CPU, pero no en los trabajos que estamos enviando, ya

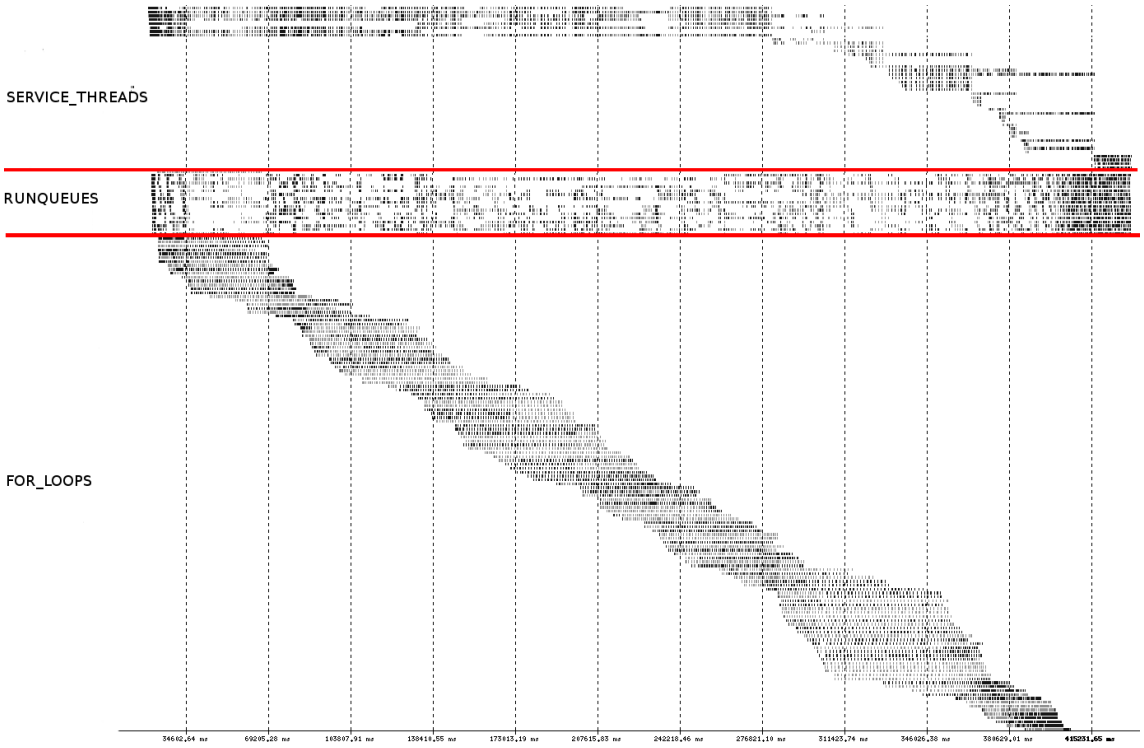


Figura 3.6: Traza Paraver que muestra el comportamiento del sistema cuando estamos enviando trabajos que sobrecargan la CPU.

que sobrecargan el sistema). Este comportamiento produce muchos efectos negativos: el *middleware* se bloquea y el tiempo de respuesta se incrementa. Además, el sistema operativo consumirá más tiempo en los trabajos (porque hay un número más elevado de ellos) que lo que estará en Globus (utilizando un planificador tradicional); adicionalmente, los cambios de contexto pueden llegar a agravar el problema.

Como podemos ver en la tabla 3.1 el tiempo de ejecución (en 128 trabajos) dura cerca de 52 segundos de media (para un trabajo de 5 segundos). También observamos como este valor medio disminuye cuando bajamos el número de trabajos ejecutados. No contemplamos el tiempo de respuesta para 256 y 512 trabajos, ya que la ejecución no se completó.

De estas trazas podemos sacar algunas vistas interesantes. En primer lugar, podemos ver como el sistema operativo distribuye CPU hacia los threads. En segundo lugar, podemos contar el número de cambios de contexto de cada thread (un total de 27.925 con una media de 218 por thread) y el tiempo medio de ráfaga. Aun con todos estos datos necesitamos realizar alguna prueba más simple para tomar algunas decisiones, por ello

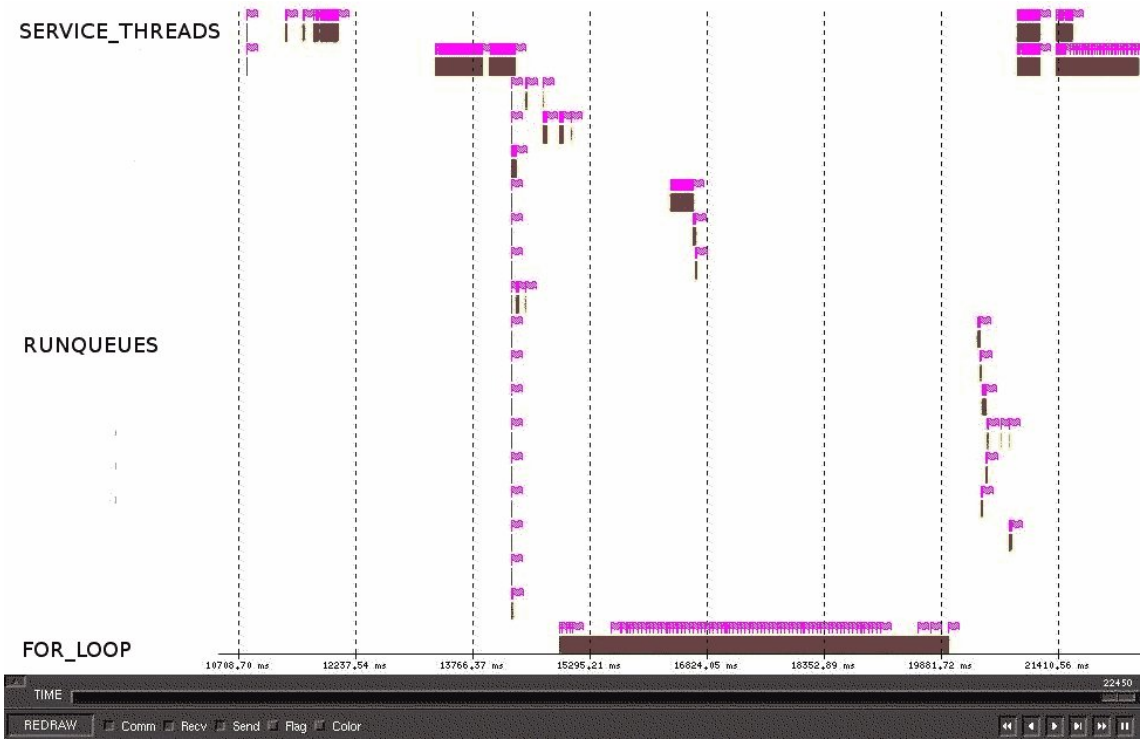


Figura 3.7: Traza con el comportamiento de la ejecución de 1 trabajo; los cambios de contexto están marcados con una bandera.

analizaremos la ejecución de 1 trabajo.

Ejecución de un trabajo

Para analizar el problema observaremos una traza en la que se ejecuta un trabajo. Con esta traza y repitiendo el análisis anterior podemos encontrar el tiempo de ejecución: 5.076 ms.

Nuestra propuesta quiere llegar a conseguir (en el lado del servidor) este tiempo de respuesta de 1 trabajo cuando enviamos 128 trabajos. Además del valor anterior encontramos que los cambios de contexto del trabajo disminuyen a 92.

Podemos observar la traza en la figura 3.7, en donde encontramos marcados con banderas los cambios de contexto.

Simplicando las trazas encontramos que para poder procesar un trabajo dentro de un sistema *Grid* necesitamos aceptar la conexión (*ServiceThreads*) y finalmente procesar el

Cuadro 3.1: Estadísticas para la ejecución de *x* trabajos en paralelo.

(a) Sin modificación.

# Trabajos	Tiempo de respuesta medio (s)	Trabajos ejecutados
16	40,73	16
32	43,83	32
64	47,44	64
128	52,67	128
256	∞	158
512	∞	158

(b) Con gestor de recursos.

# Trabajos	Tiempo de respuesta medio (s)	Trabajos ejecutados
16	5,34	16
32	5,74	32
64	10,65	64
128	15,8	128
256	∞	201
512	∞	238

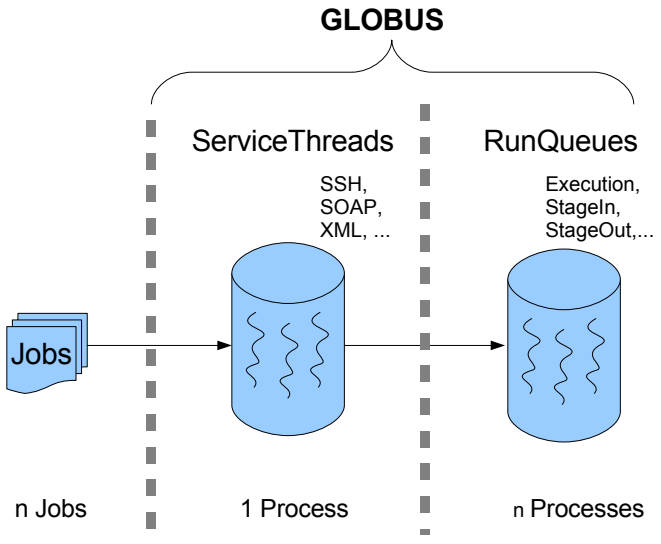


Figura 3.8: Globus Middleware, funcionamiento.

trabajo (RunQueues y el sistema operativo, de modo que pueda ser enviado a otro nodo). Todo este procesado se realiza en el mismo nodo, como muestra la figura 3.8, donde, a pesar de tener más de un *ServiceThread*, lo representaremos como 1 en términos de procesos en los siguientes pasos, ya que se mantiene constante en la ejecución. En la figura anterior podemos observar como hay una relación n - n entre los trabajos que entran al sistema y el número de procesos que se ejecutan cuando utilizamos la instalación de Globus por defecto.

Estudio analítico

Como hemos visto en las trazas, el comportamiento original de Globus es ejecutar todos los trabajos a la vez. Vamos a aplicar una simplificación en el escenario para mostrar el comportamiento con teoría de colas [Bolch et al., 1998] y extraer algunas conclusiones preliminares.

Observando la figura 3.9, el primer sistema *QN* (cola $M/M/n$) muestra qué sucede en un sistema cuando tenemos gran cantidad de trabajos en el nodo de entrada. Como tenemos un servidor *PS* (*Processor Sharing*), todos los trabajos se ejecutan a la vez usando un *quantum* (típicamente de *100 Hz*) para cambiar entre procesos y trabajos en el sistema. Además, necesitamos procesar los trabajos que llegan en el mismo servidor. De este modo sólo conseguimos para el servicio de entrada un porcentaje de CPU igual al valor de la ecuación (3.1) cuyo límite encontramos en la ecuación (3.2). Los trabajos se descartan o se pierden en el proceso de entrada cuando no tenemos la suficiente potencia de cálculo para ellos (el servidor se sobrecarga).

Podemos mejorar este escenario introduciendo una limitación en el número de trabajos que hay en el sistema de manera concurrente (simulado con una cola $M/M/n/k$). Si colocamos 2 trabajos dentro de la CPU, terminaran en el instante $2 * (service\ time)$. En este caso incrementamos la cantidad de CPU asignada al proceso de llegada y aceptación de trabajos y conseguimos la ecuación (3.3). Este valor, que no depende de $NumJobs$, necesita ser suficiente para procesar los trabajos que llegan al sistema; si no es así, acabaremos con el mismo problema que antes. Podemos generalizar este valor con una variable llamada "límite" obteniendo la ecuación (3.4).

Sin embargo, otro problema aparece con esta modificación. Dado el incremento constante del tamaño de la cola, necesitaremos incluir algún control de admisión. Vamos a ignorar este problema en el prototipo que presentaremos, centrándonos en cargas tipo burst por períodos limitados de tiempo.

¿Qué valor sería bueno para *límite*? Probablemente no tendremos el mismo nivel de llegadas todo el tiempo; además, nuestros recursos podrían modificarse en cualquier mo-

$$\frac{1}{numJobs+1} \quad \text{CPU \% por proceso} \tag{3.1}$$

$$\lim_{numJobs\to\infty} \frac{1}{numJobs+1} = 0 \quad \text{CPU \% por proceso} \tag{3.2}$$

$$\frac{1}{2+1} = 0{,}33 \quad \text{CPU \% por proceso} \tag{3.3}$$

$$\frac{1}{limite+1} \quad \text{CPU \% por proceso} \tag{3.4}$$

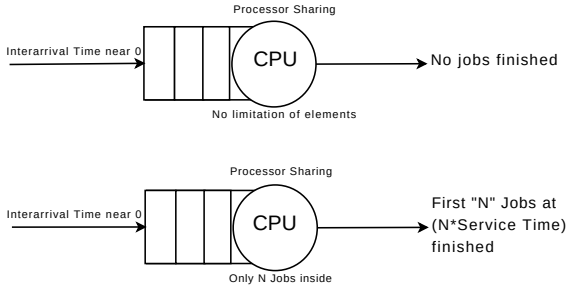
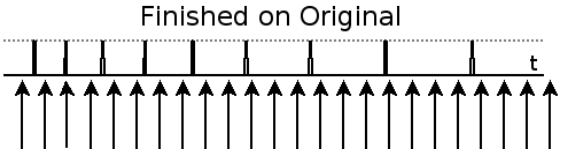


Figura 3.9: Cola Simplificada, arrival rate de $(\lambda) = \infty$.

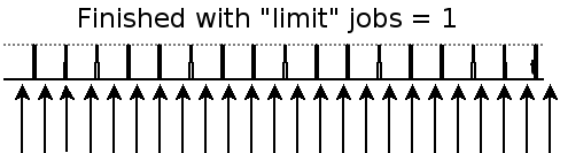
mento (como veremos en el capítulo siguiente). Necesitamos obtener, de forma continua, un valor para *límite* donde la ecuación (3.4) sea mayor o igual a la necesidad de consumo para aceptar trabajos. Está claro que el valor de esta variable está relacionada con el número de procesadores del sistema (o con la potencia de las mismas).

Mostremos otro escenario con un nivel de envíos diferente; en esta ocasión estamos enviando trabajos con un $\lambda = \frac{1}{2}$ trabajos/segundo y un tiempo de servicio de 5 segundos. Usando una *QN* sin limitación (figura 3.10(a)), el tiempo entre trabajos finalizados se incrementa y el servicio se va a sobrecargar en algún punto. Además, al tener *timeout*, los trabajos empezarán a perderse en algún momento; este *timeout* limita el tamaño de la cola (como un control de admisión).

Por otro lado, con una *QN* con una limitación de 1 trabajo simultáneo (figura 3.10(b)), el tiempo entre trabajos finalizados es constante. En este caso el servicio no se sobrecar-



(a) Sin limitación $\lambda = \frac{1}{2}$ trabajos/segundo.



(b) Con limitación $\lambda = \frac{1}{2}$ trabajos/segundo.

Figura 3.10: Análisis analítico del problema.

gará, pero deberemos tener controlado el tamaño de la cola de espera.

Resultado del análisis

Para crear una política de gestión de recursos debemos comparar los distintos parámetros obtenidos con el análisis.

Cuando la capa del *middleware* (o del sistema operativo) necesita CPU, enviará el trabajo (o trabajos) que se está ejecutando fuera del procesador; esto incrementa los cambios de contexto del mismo, perdiendo todas las ventajas de continuar su ejecución, como puede ser la afinidad. Para reducirlos podemos incrementar la prioridad de los trabajos y así conseguir un tiempo de ejecución más continuo. Otra medida para optimizar los recursos es ejecutar sólo tantos trabajos como procesadores libres tengamos en el sistema, y esto, como hemos observado en los cronogramas, puede ofrecernos mejoras sustanciales. Nuestra primera propuesta será exactamente está, reducir el número de trabajos que se ejecutan simultáneamente en el sistema.

3.5.3. Propuesta de limitación de concurrencia

Nuestra primera idea y propuesta es la de ofrecer la posibilidad de limitar el número de trabajos que pueden ejecutarse en el sistema; además, incrementaremos la prioridad de estos trabajos para disminuir el número de cambios de contexto sobre los procesos.

Esto nos permitirá incrementar el rendimiento del servidor permitiéndonos pasar más tiempo haciendo cosas útiles (aumentando también la afinidad del sistema).

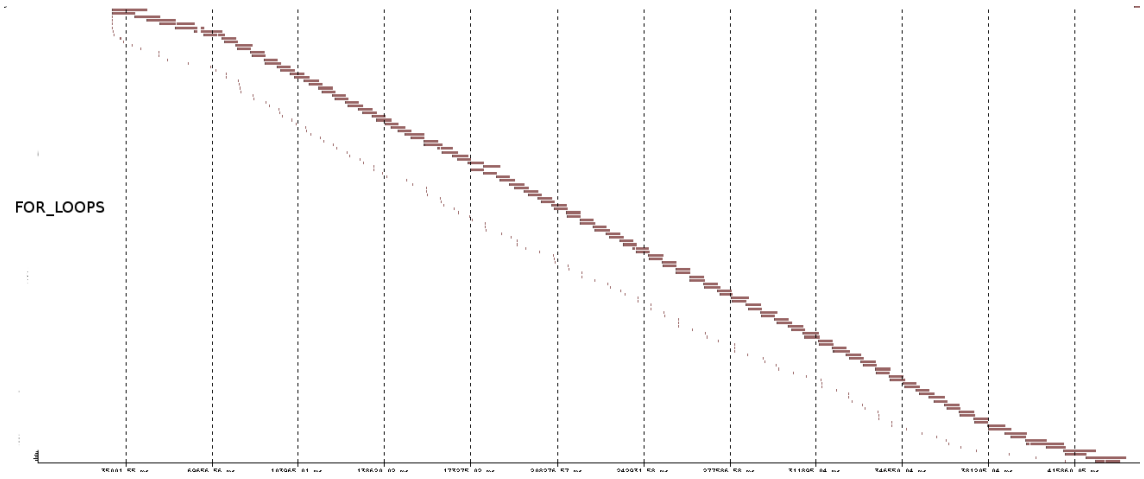


Figura 3.11: Traza mostrando el funcionamiento del gestor de recursos.

Como en el apartado anterior, analizaremos la propuesta con Paraver para visualizar el sistema de manera global y poder mejorar esta primera propuesta.

Evaluación

Vamos a analizar la ejecución de 128 trabajos con gestión de recursos: podemos ver en la figura 3.11 como estamos ejecutando sólo 2 trabajos en paralelo; utilizando las herramientas de análisis estadístico de las trazas anteriores podemos ver como el tiempo de respuesta medio ha mejorado (cuadro 3.1(b)), y sobre todo que hemos incrementado el número de trabajos ejecutados satisfactoriamente. Si el tiempo de respuesta en esta propuesta, el número de trabajos también aumenta; éste es un parámetro que podríamos utilizar en un entorno de *Autonomic Computing*, como veremos más adelante. Incrementar la prioridad de los trabajos (ejecutando 2 a la vez) puede ayudarnos a conseguir 200 trabajos finalizados cuando enviamos 1.000 en un punto donde el entorno original no nos ofrece ninguna respuesta.

La disminución de los cambios de contexto también es notable consiguiendo una reducción de 27.395 hasta sólo 1.788. De media cada trabajo cambia 14 veces de contexto (218 usando la propuesta original).

3.5.4. Propuesta de gestión en dos fases

Para mostrar lo importante que es la auto-gestión propondremos una ampliación del anterior sistema utilizando una gestión en dos fases. No es una propuesta autónoma todavía, pero se puede extender para que lo sea. Utilizaremos el mismo ejemplo que el caso anterior, pero enviando dos níveles de carga (128 y 256) de trabajos concurrentes al servidor. En el primer grupo intentamos mantener el sistema con una carga alta pero sin sobrecargarlo. En el segundo nivel intentamos sobrecargar el sistema. Los trabajos sintéticos usados en el experimento consisten, como en la sección anterior, en un trabajo de 5 segundos que consume un procesador. Veremos que faltan una gran cantidad de recursos en el servidor para servir los pedidos. Resumiendo, el objetivo es mostrar la necesidad de introducir capacidades de auto-gestión dentro del *middleware*, pudiendo usar técnicas de *Dynamic Provisioning* [Andrzejak et al., 2002, Chandra et al., 2003a] para mejorar la *QoS*.

Estudio: Comportamiento sobrecargado y no sobrecargado

Los resultados obtenidos en las propuesta de limitación de los trabajos (3.5.3) pueden verse en la figura 3.3 mostrada anteriormente, donde el eje X representa el número de trabajos enviados y el eje Y representa el número de trabajos finalizados correctamente. Podemos ver como el resultado es bastante inferior al deseado; con los datos obtenidos de esta prueba llegamos a la conclusión que en nuestra plataforma de pruebas se pueden procesar 128 trabajos concurrentes sin problemas en nuestro entorno experimental.

A partir de este punto, algunos trabajos empiezan a fallar y no superan los 170 trabajos finalizados. Cuando el nivel de sobrecarga es extremo, el funcionamiento del servidor es inaceptable para el tipo de servicio que debería ofrecer un servidor *Grid*.

Identificando las causas de las pérdidas de trabajos Como ya hemos visto anteriormente, bajo condiciones extremas el servidor empieza a dejar de aceptar nuevos trabajos (o los pierde). Necesitamos determinar qué está causando este problema y ver si podemos superar o hacer menos caótica esta situación.

Para conseguirlo, seguimos usando *BSC-MF* para poder realizar un análisis en detalle de la sobrecarga del servidor *GT4* y compararlo con el comportamiento de un servidor bajo condiciones normales. Para hacerlo utilizamos dos niveles de carga de los que hemos utilizado en 3.3 que nos proporcionan dos escenarios: uno para un servidor no sobrecargado (128 trabajos concurrentes) y otro escenario, o nivel de carga, que sobrecarga el sistema (256 trabajos concurrentes), obteniendo dos trazas. Recordemos que el escenario

es el de una máquina con 2 procesadores (por lo que el nivel máximo de utilización de CPU es de un 200 % en términos absolutos).

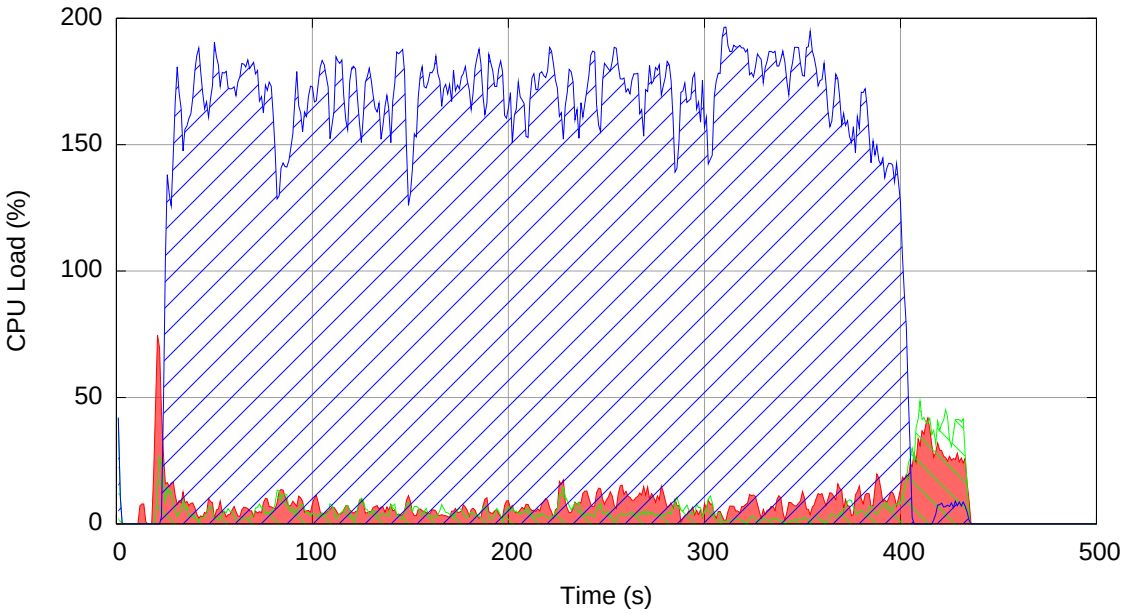
Del estudio de las trazas de los dos experimentos (128 y 256 trabajos) hemos extraído información acerca del consumo de CPU producido durante la ejecución de ambas cargas. Esta información se clasifica según el origen del consumo de la CPU: el trabajo (capa de sistema), los *ServiceThreads* (que corresponden a las operaciones sobre sockets y el procesado del mensaje *SOAP*) y las *RunQueues* (que hacen que el trabajo pase por los distintos estados del mismo).

Es importantes destacar lo que puede significar que se inhabilite alguna de las capas enumeradas: si dedicamos poco esfuerzo a los trabajo, estos tardarían en ejecutarse; si por el contrario lo hacemos sobre los *ServiceThreads*, no conseguiremos que los trabajos lleguen a entrar (o salir); finalmente, dedicar pocos esfuerzos a las *RunQueues* crearía un tapón en el sistema haciendo imposible el avance de los trabajos.

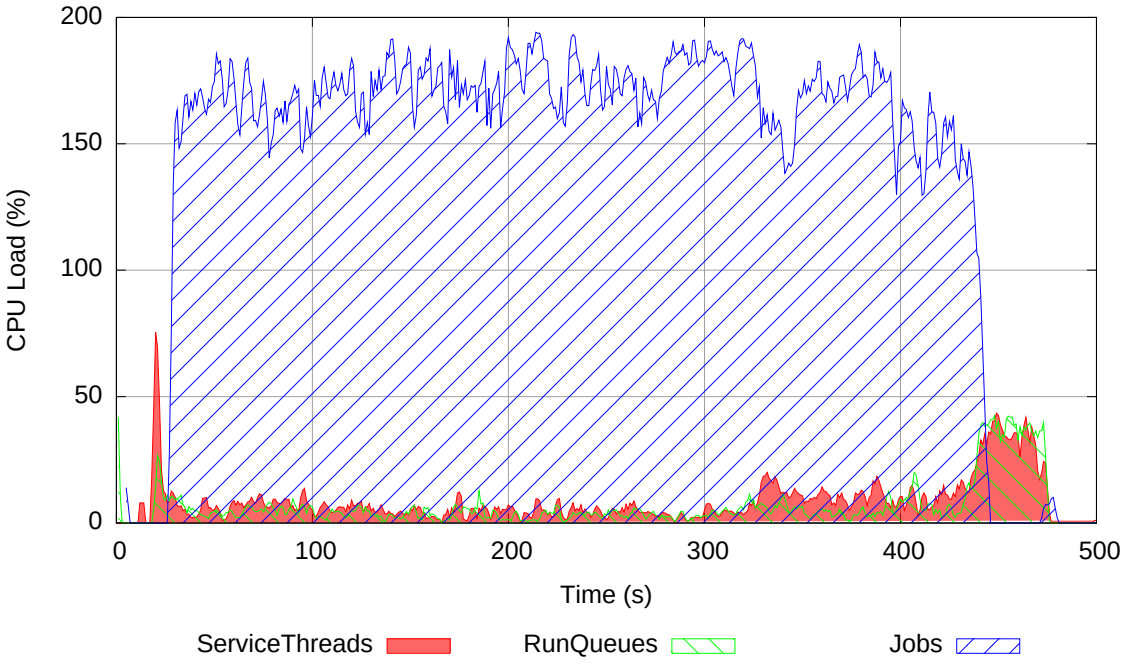
Las figuras 3.12(a) y 3.12(b) muestran los resultados obtenidos después del análisis de las trazas. Estas figuras muestran la distribución del consumo de CPU a través de las distintas capas que forman la pila de ejecución del *middleware* cuando está funcionando en condiciones normales y cuando está sobrecargado.

De estas gráficas podemos observar como el consumo de CPU de los *ServiceThreads* es muy similar en los dos workloads. Un análisis estadístico usando Paraver nos muestra que el problema está en la manera que interfiere el nivel sistema con la manera estándar de trabajar los threads del *middleware* (*ServiceThreads* y *RunQueues*). Este comportamiento genera una pérdida de trabajos dentro del sistema, reduciendo el número de trabajos finalizados sobre los 150. Teniendo esto presente, un análisis más en detalle de las dos figuras anteriores (3.12(a) y 3.12(b)) muestra que el consumo de CPU de las capas de *GT4* no aumenta (cosa que no sería lógica, ya que tenemos el doble de carga), ya que el consumo de CPU por el nivel sistema lo bloquea. Ejecutar los trabajos tan pronto como llegan no sería una política inteligente en muchos casos: consumimos CPU de los otros niveles, además de tener un número relativo de hilos de ejecución superior respecto a los del *middleware*. Este comportamiento deriva en un *middleware* poco eficiente. En otras palabras, los nuevos trabajos y sus respectivas conexiones no se procesan porque no tienen CPU disponible.

Concretando más, la causa de esta alteración en el funcionamiento normal está provocada por la falta de potencia de CPU requerida por el servidor para establecer una nueva conexión segura entre el servidor y el cliente que envía el trabajo. Este problema se ha identificado en otros entornos en [Guitart et al., 2005a].



(a) 128 trabajos.



(b) 256 trabajos.

Figura 3.12: Consumo de CPU en el *middleware* original. 2 procesadores.

Funcionamiento del gestor en dos fases

Nuestro objetivo en este punto es ajustar el comportamiento del sistema en general, modificando algunos de sus parámetros, como por ejemplo la manera en que se crean los threads en el contenedor, el número de *RunQueues* o el planificador local. Así, un nuevo gestor de recursos derivado de los resultados obtenidos en el capítulo anterior se introduce para disminuir los síntomas de sobrecarga.

La estrategia que seguimos en nuestro nuevo gestor de recursos está basada en una política de dos fases. Para poder balancear los procesos o tareas en ejecución en cada capa del servidor (ejecutar trabajos, *RunQueues* y el control de las conexiones) necesitamos dividir la capacidad de proceso en dos fases:

- Fase 1: Recibir nuevos trabajos. Mientras recibimos trabajos en la capa de los *ServiceThreads*, intentamos reducir el consumo de CPU del nivel de sistema (en nuestro caso la ejecución del trabajo). Para obtener este comportamiento reducimos el número de trabajos que se están ejecutando, forzando la ejecución en paralelo de un número específico de trabajos a la vez (limitándonos a 1 ó 2 ya que estamos en una máquina dual-core). Esto significa que la entrada de estos nuevos trabajos al sistema estará garantizada con el coste de reducir el throughput medio del servidor durante este periodo. Esto asegura que menos trabajos (o ninguno si es posible) serán cancelados o perdidos antes de alcanzar el contenedor de GT4.

- Fase 2: No se reciben más trabajos. En este momento cambiamos a la política original de *GT4* (ejecutar tan pronto como sea posible) y toda la CPU está disponible para ejecutar los trabajos. En este capítulo hemos utilizado una política fija para mostrar cómo podemos obtener beneficios con una estrategia más general, comentada en la subsección 3.5.5.

Evaluación

Nuestra propuesta quiere incrementar el número de trabajos que finalizan satisfactoriamente en el sistema, jugando con el tiempo de respuesta y el throughput que obtenemos del contenedor *GT4* modificado con la propuesta en dos fases comentada anteriormente. Aplicando esta estrategia con una carga de 128 trabajos concurrentes produce la salida obtenida en la figura 3.13. La gráfica muestra una vista de la traza del sistema obtenida con Paraver (sombreado tenemos a qué corresponde cada thread), y en la parte inferior la gráfica de consumo de CPU para el sistema modificado.

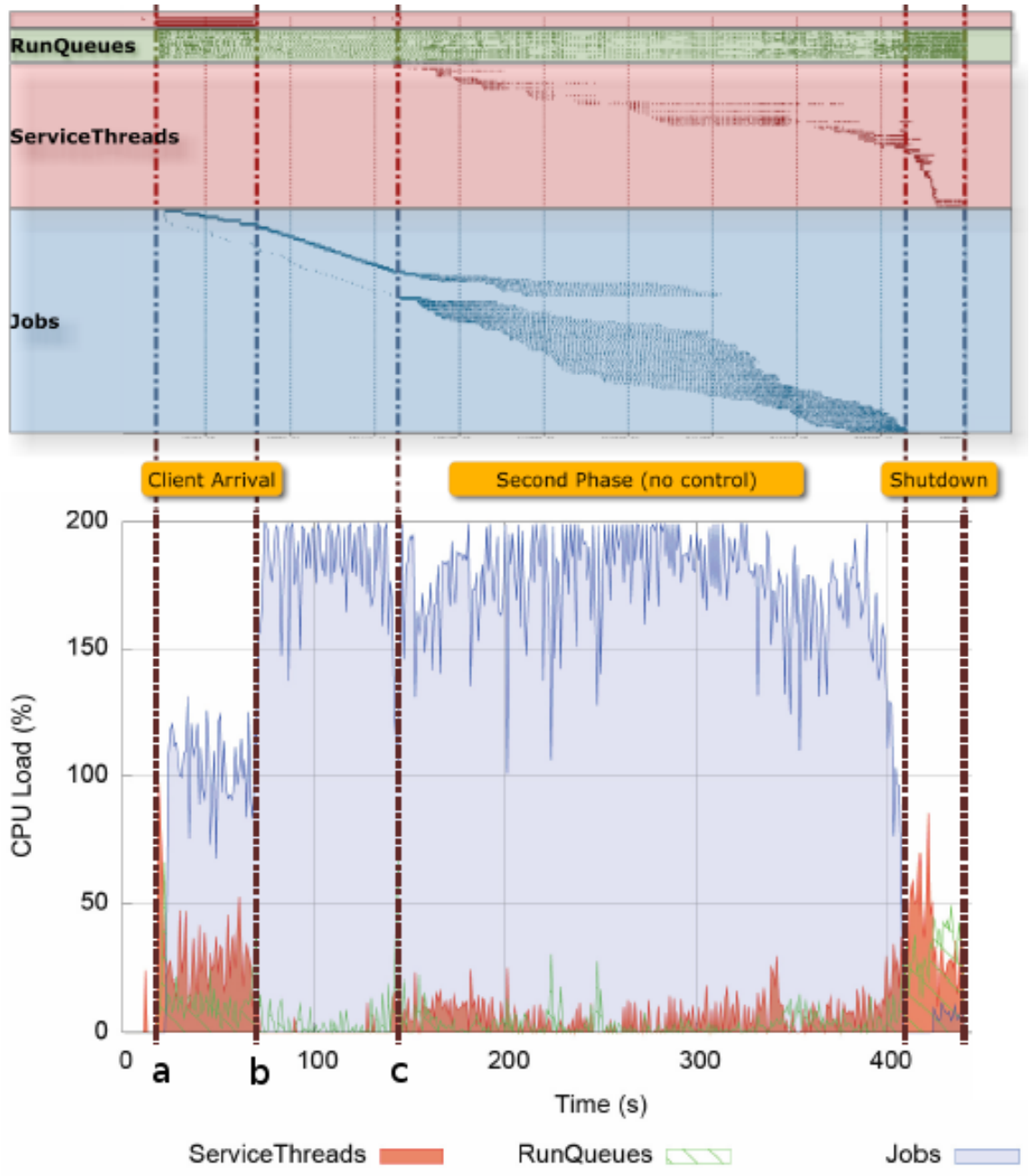


Figura 3.13: Traza Paraver y gráfica mostrando el consumo de CPU de 128 trabajos usando un *middleware* modificado (450 segundos de ejecución).

Podemos ver el cambio de fase en la gráfica mediante una linea vertical (**c**). Además, podemos observar (entre **a** y **b**) como los *ServiceThreads* realizan un trabajo intensivo para conseguir hacer entrar los trabajos en el sistema de *RunQueues*.

Cuando el *middleware* está en la primera fase sólo ejecutamos 2 trabajos en el sistema para dar más CPU a los *ServiceThreads* (o más concretamente, no quitársela). En esta fase el cliente envía todos los trabajos (128) simultáneamente, por eso podemos ver una gran densidad en la vista *Paraver* en la parte de los *ServiceThreads* y un consumo medio en la parte de la gráfica de utilización de CPU.

Cuando no hay más trabajos llegando al sistema, no necesitamos reservar más recursos para otras tareas; podemos desactivar el gestor de recursos de manera que los trabajos se ejecuten lo antes posible (segunda fase). En este momento las colas de ejecución (*RunQueues*) empiezan a trabajar, pero no es un momento crítico.

Comparando la traza y la vista de la gráfica con la figura 3.12(a) y la figura 3.12(b) podemos concluir que la diferencia entre el consumo de CPU está en el inicio, como podemos ver en la figura 3.15(a). En esta gráfica hemos pintado la diferencia entre el consumo de CPU original y el consumo del *middleware*. Una diferencia positiva significa que el consumo de CPU en el sistema original es superior. Podemos ver este cambio al principio de la prueba (cuando los trabajos entran al sistema).

Repitiendo el test con 256 trabajos obtenemos la gráfica y la traza que aparecen en la figura 3.14, equivalente a la figura 3.13 pero con 256 trabajos. Podemos ver la diferencia del consumo de CPU (Original-Modificado) en la gráfica 3.15(b). Destacamos que sólo mostramos el principio del test, ya que con el *middleware* modificado podemos ejecutar todos los trabajos y por lo tanto la prueba dura más. Aun así podemos ver como ahora el comportamiento al inicio es similar en los dos niveles de carga.

Si comparamos la primera fase, en la que el cliente envía todos los trabajos, con la figura del *middleware* original 3.12(b), podemos ver que el consumo de CPU de los *ServiceThreads* aumenta con nuestro gestor de recursos: necesitamos más potencia para servir correctamente los trabajos entrantes. Cuando entramos en la segunda fase el consumo de CPU es similar al anterior.

Con nuestra propuesta hemos sido capaces de ejecutar todos los 256 trabajos en unos 900 segundos y los 128 trabajos en 450 segundos. En el *middleware* original se ejecutan 128 trabajos en cerca de 450 segundos, incluyendo el arranque y la destrucción del servidor, de modo que no introduce retardos en el sistema.

Resumiendo, hemos podido incrementar el número de trabajos finalizados en el servidor sobrecargado cuando éste recibe una gran cantidad de trabajos simultáneos. Esto es importante ya que la pérdida de trabajos en un *middleware Grid* no debería ocurrir nunca.

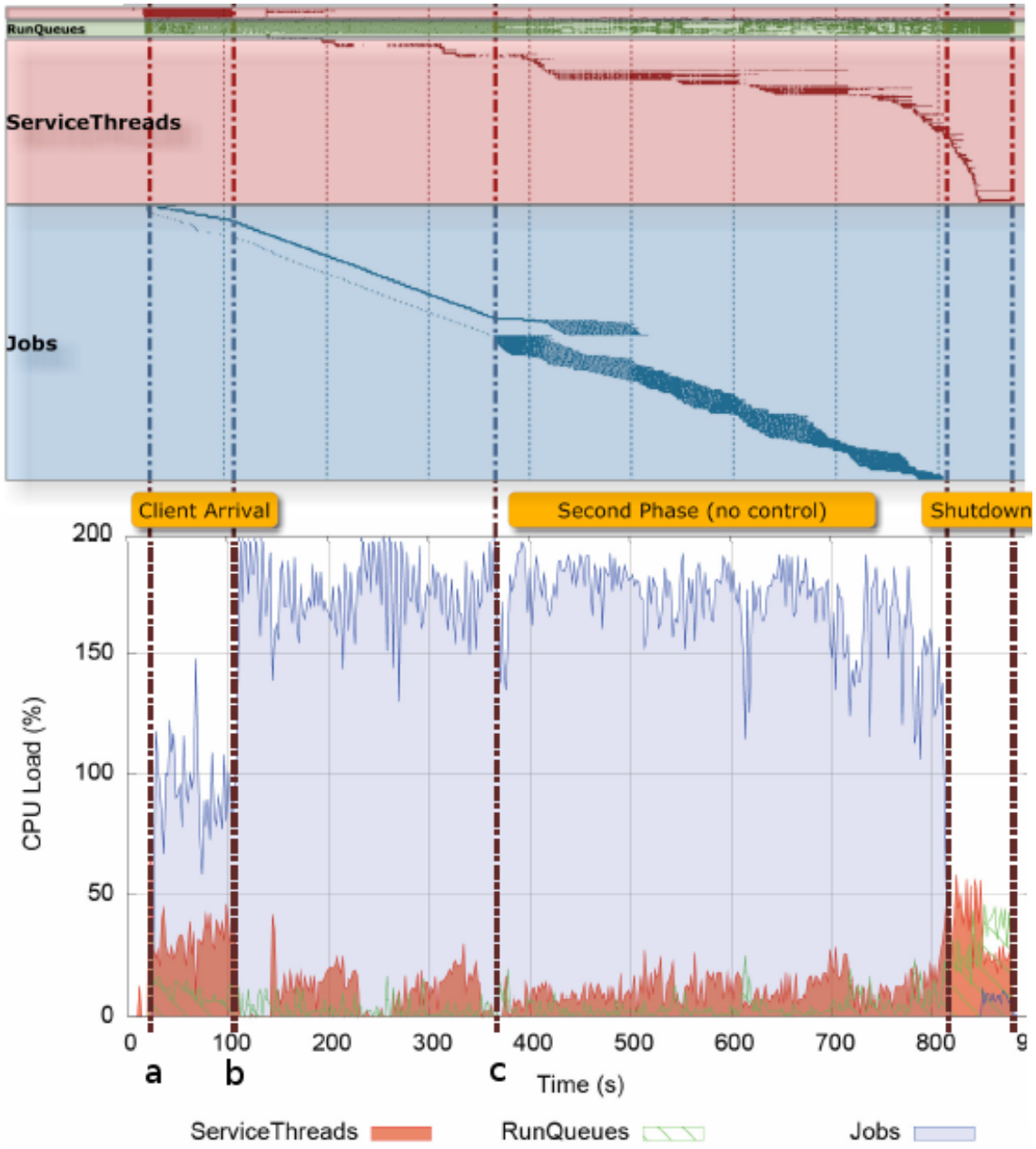
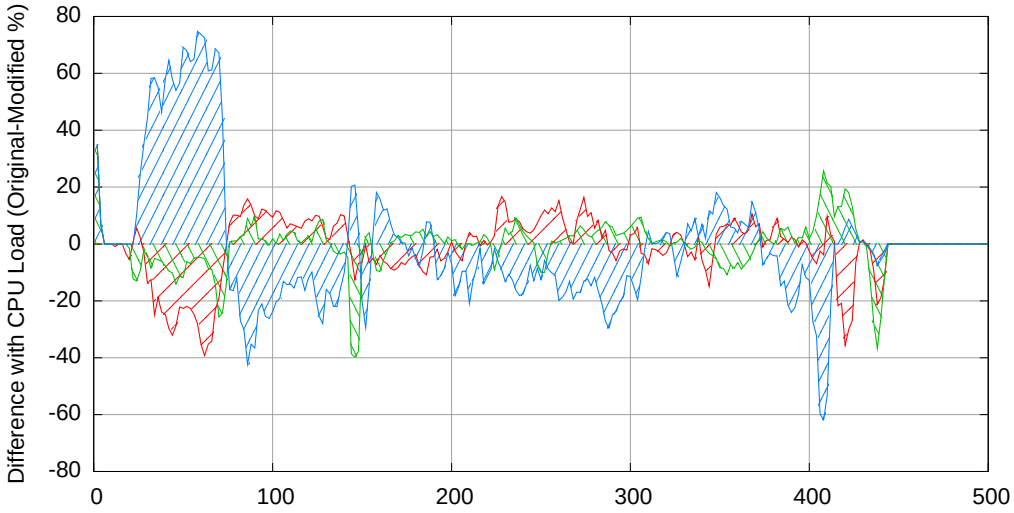
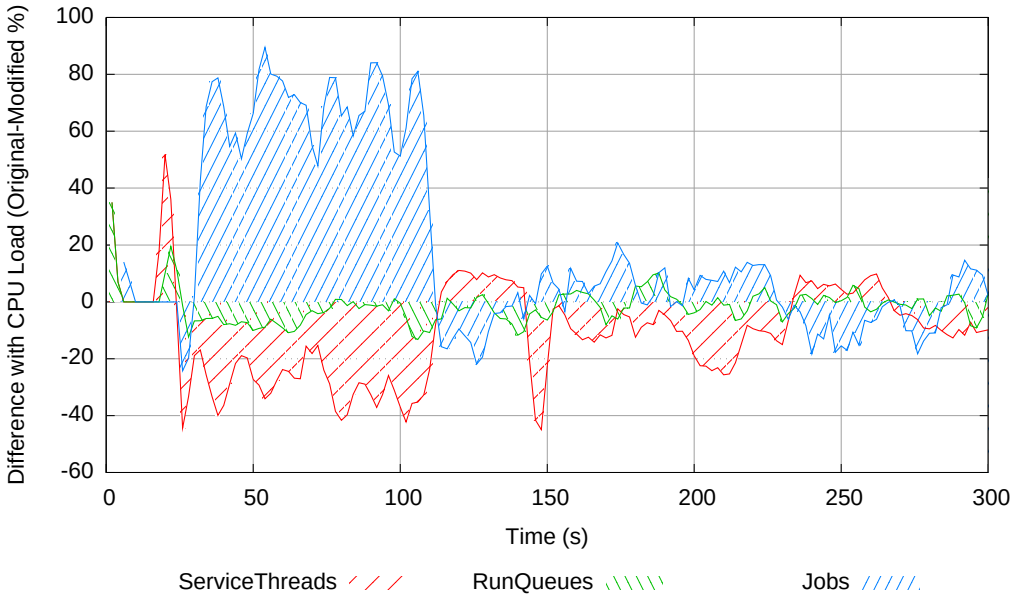


Figura 3.14: Traza Paraver y gráfica mostrando el consumo de CPU de 256 trabajos usando un *middleware* modificado (900 segundos de ejecución).



(a) 128 trabajos.



(b) 256 trabajos (recortado, no finalizan todos).

Figura 3.15: Diferencia del consumo de CPU entre las dos propuestas (original-modificado).

Introducir una capa de auto-gestión es necesaria para mejorar la calidad de servicio de *Grid*. En la siguiente sección mostraremos una generalización, ya autónoma, del gestor.

3.5.5. Generalización de la propuesta

Para poder ofrecer una propuesta más general deberemos encontrar un mecanismo que aprenda del sistema. Podemos usar los datos preliminares y la información obtenida de las trazas (tiempo de servicios, sobrecarga de la CPU...) para cargar o entrenar inicialmente el sistema. *BSC-MF* nos ofrece una manera de ver, analizar y validar las métricas que podemos usar en el área de *Autonomic Computing*. Gracias al estudio sabemos que debemos conseguir el consumo de CPU de los *ServiceThreads* para poder ajustarnos y obtener un *middleware* de calidad (hablando en términos de trabajos finalizados). Usando un sistema auto-gestionable como éste, podríamos obtener mejor rendimiento que el obtenido con las propuestas anteriores. En este punto introduciremos un prototipo capaz de auto-adaptarse al entorno (que puede ser dinámico, como observaremos en el siguiente capítulo).

3.6. Introducción a los sistemas autónomos

La tecnología *Grid* [Sotomayor and Childers, 2005, Romberg, 2002] es compleja, ya que tiene un elevado número de parámetros: desde la red (topología, velocidad y políticas de seguridad) y los recursos (número de máquinas, número de procesadores, velocidad y uso compartido de la máquina) hasta el planificador (*fork* o más complejos). El caso que nos ocupa es el de un nodo de entrada donde enviamos nuestros trabajos. En un *Grid* normal, o de baja carga, este nodo puede ser un nodo donde se procesen trabajos, pero al incrementar la complejidad del *Grid* y aumentar el número de trabajos que nos llegan (*arrival rate* o λ), necesitaremos un servidor (dedicado) como punto de entrada para poder procesar y distribuir tantos trabajos como sea posible.

Como hemos visto en las secciones anteriores, un nodo que se encargue de aceptar y ejecutar trabajos puede llegar a un estado donde se pierdan más trabajos que los que finalmente se ejecutan (como muestra [Nou et al., 2006c]). Perder trabajos es un importante problema en la vida real [Iosup et al., 2006], y es más importante cuando estamos compartiendo recursos.

No hemos encontrado trabajos relacionados usando un nodo *Grid*, pero hay un número importante de artículos donde los sistemas auto-gestionables usan otras técnicas (modelos analíticos [Bennani and Menascé, 2005, Almeida et al., 2006, Menascé and Bennani,

2003]) o se dirigen hacia servidores web [Menascé et al., 2005]. Con estas técnicas, tenemos las limitaciones de un modelo matemático: necesitamos tiempo para obtener un resultado, son inflexibles y no permiten modelar algunas características del software, como hemos visto en el capítulo 2.

Los sistemas autónomos y los entornos auto-gestionables permiten que el sistema reaccione y prevenga estados específicos donde empieza a ser inestable o mantiene un comportamiento intolerable. Estos sistemas pueden también comunicarse con un entorno virtualizado [Menascé, 2005] para ofrecer un conjunto variable de recursos (como veremos en el capítulo siguiente) y aumentar la calidad del sistema [Bennani and Menascé, 2005, Buyya et al., 2005].

3.7. Capa de autogestión autónoma para Globus

Un paso importante para mejorar la *QoS* de un nodo *Grid* [Carrera et al., 2005], de manera que el entorno funcione correctamente, es mejorar el servidor de aplicaciones que utiliza el *middleware Grid*. Conseguir las variables óptimas no es tarea fácil, ya que los trabajos que se ejecutan en el sistema son altamente dinámicos. La solución más apropiada sería ofrecer la habilidad de auto-gestionarse a los componentes del *middleware*; podemos hacerlo con varias propuestas que vienen del área de *Autonomic Computing*, que obtiene conocimiento de un gran numero de areas y campos [Parashar and Hariri, 2005, Kephart, 2005]. Utilizando la monitorización podemos obtener conocimiento de estos sistemas complejos para ayudarnos en la creación del entorno.

En las siguientes secciones mostraremos la necesidad de obtener un *middleware Grid* autónomo para conseguir un rendimiento acorde al tipo de aplicación. El trabajo sobre *middleware* autónomo era en su momento poco común [Chess et al., 2005]; la falta de trabajo en esta área se acentúa cuando trabajamos sobre *middleware Grid*. En [Agarwal et al., 2003, Hau et al., 2003] tratan algunos problemas descritos en este capítulo, pero no realizan auto-gestión en el nivel sistema.

3.7.1. Análisis del problema

El nodo de entrada de una plataforma *Grid* es un componente importante en el conjunto de la arquitectura y es un punto de contención; en este capítulo demostraremos que la utilización de una capa auto-gestionada en el nodo de entrada es importante. Utilizando nuestro sistema auto-gestionado, nos permitirá aceptar y finalizar más trabajos correctamente. Perder trabajos es un comportamiento inaceptable en un *middleware Grid*, por

lo tanto debemos priorizar la finalización y la aceptación de trabajos sobre el tiempo de respuesta o el throughput.

Descomposición de los trabajos y del consumo de CPU

Analizando matemáticamente el comportamiento, podemos considerar que en un determinado momento tenemos $\tau\,\%$ de la CPU disponible. Sabemos el consumo de CPU de la petición, *p*, y el consumo de CPU al procesar un trabajo, *j*. De este modo, si procesamos *n* peticiones y *n'* trabajos, tenemos un consumo de CPU de Γ donde podemos considerar 3 casos:

$$\Gamma = np + n'j \qquad\qquad \Gamma = \begin{cases} < \tau & \surd \\ = \tau & \surd \\ > \tau & \text{X} \end{cases}$$

En los dos primeros casos nuestro consumo es menor o igual a la CPU disponible, de modo que no tenemos ningún problema. Pero, ¿qué sucede cuando necesitamos más CPU? En el caso estándar, la CPU se distribuye hacia todos los threads, de modo que tenemos $\tau/2$ (mitad de la CPU) para cada parte.

$$np + n'j > \tau \qquad\qquad \begin{cases} np > \frac{\tau}{2} & \text{perdemos trabajos} \\ \\ n'j > \frac{\tau}{2} & \text{retardar trabajos} \end{cases}$$

Vamos a centrarnos en la fase en la que se procesa la petición; en este caso el sistema consume demasiado tiempo, de modo que tendremos *timeouts* por el *SSL*. Por otro lado, si necesitamos más CPU para procesar el trabajo, tendremos un retardo en su ejecución. Nuestro prototipo de auto-gestión propone una planificación distinta de la CPU disponible, similar a la siguiente expresión:

$$\begin{cases} np \leq \frac{3\tau}{4} & \text{no se pierden trabajos} \\ \\ n'j \gg \frac{\tau}{4} & \text{retardar trabajos} \end{cases}$$

De este modo, la parte de la petición consume menos CPU que la que está disponible y los trabajos no se pierden.

3.7.2. Prototipo de sistema auto-gestionable

En esta sección explicaremos nuestro diseño. Para construir y probar nuestro prototipo estamos usando dos máquinas; un cliente, que tiene nuestro generador de carga enviando trabajos al servidor, donde se ejecuta un nodo de entrada usando Globus Toolkit 4 (GT4) [Sotomayor and Childers, 2005].

Generador de carga

Un trabajo es un trozo de código que puede ejecutarse en un *Grid*. En este experimento concreto, como estamos probando el proceso de envío de trabajos en un nodo de acceso, el trabajo no hace nada, por lo que sólo usamos la parte de Globus que queremos (la de gestión).

Para intentar sobrecargar el servidor, generamos trabajos usando un ratio de λ trabajos por segundo (donde λ es tan grande como sea posible) y contamos los que se finalizan correctamente (β). En este escenario podemos reducir λ a α, donde α contará el número de trabajos enviados al sistema en el mismo instante. El escenario ideal producirá un resultado donde $\alpha = \beta$. Un escenario sobrecargado dará un resultado tal que $\alpha > \beta$.

En esta prueba necesitamos utilizar otro componente: el tiempo (t). Si vamos a limitar la ejecución para reducir el número de trabajos perdidos, también necesitaremos más tiempo para ejecutarlos (T). Para poder conseguir medidas vamos a contar β para cierto t, donde ($0 < t <= T$). Toda esta información nos dará una gráfica 3D, que analizaremos más tarde. En los experimentos consideramos que con un t de 100 segundos tenemos suficiente para ejecutar todos los trabajos (teniendo en cuenta que los trabajos no consumen CPU por sí mismos).

Carga generada

La carga puede dividirse en varias partes. Primero intentamos enviar simultáneamente entre 10 y 100 trabajos (esto se hace en varios tests consecutivos, incrementando el número de envios con un paso de 10 trabajos). Podemos representar el número de trabajos finalizados como $F^{10}...F^{100}$. Como hemos dicho anteriormente, limitaremos el tiempo de la prueba, cogiendo períodos comprendidos entre 10 y 100 segundos (con pasos de 10 segundos). Esto se representará como F^{10}_{20} (número de trabajos finalizados cuando se envían 10 trabajos y tienen 20 segundos para ejecutarse). Para conseguir todos los datos enviamos todas las configuraciones descritas y ampliamos el detalle en algunos puntos.

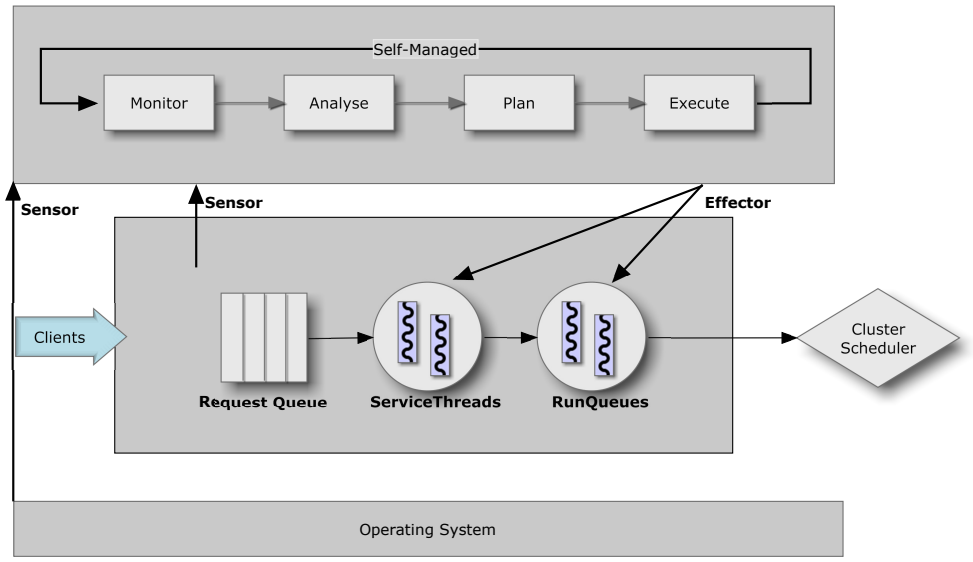


Figura 3.16: Diagrama del prototipo autónomo.

Con esta carga podemos conseguir valores muy útiles. En primer lugar tenemos el total de trabajos enviados (S) en todas las pruebas y los trabajos finalizados ($F = \sum_{x,y=10}^{100} F_y^x$). En nuestro caso S es igual a 12.605 ($100 * 10 + 90 * 10 + 80 * 10... +$ puntos adicionales) y F depende del resultado de las pruebas.

Prototipo

Nuestro prototipo es un nodo auto-gestionable para GT 4.0.1, basado en las ideas mostradas en [IBM-Corporation, 2004]. Aunque el prototipo es simple, es suficiente para demostrar el impacto y la necesidad e importancia en las plataformas *Grid*.

Cuando hablamos de un sistema *Grid* auto-gestionable (o más concretamente de un nodo *Grid* auto-gestionable), necesitamos considerar algunos de los problemas que encontramos estudiando el comportamiento de dichos nodos en entornos sobrecargados [Nou et al., 2006c], o como hemos visto en capítulos anteriores. Concluímos que necesitamos capacidades de auto-gestión para prevenir los comportamientos *destructivos* del nodo en dichas situaciones. Además, desarrollamos un sistema simple capaz de detectar esas situaciones y adaptarse a ellas antes de caer. El nodo auto-gestionable ofrece simplemente una buena gestión de los recursos del sistema.

La diferencia entre el nuevo nodo y viejo es lo que llamamos el *Autonomic Manager*

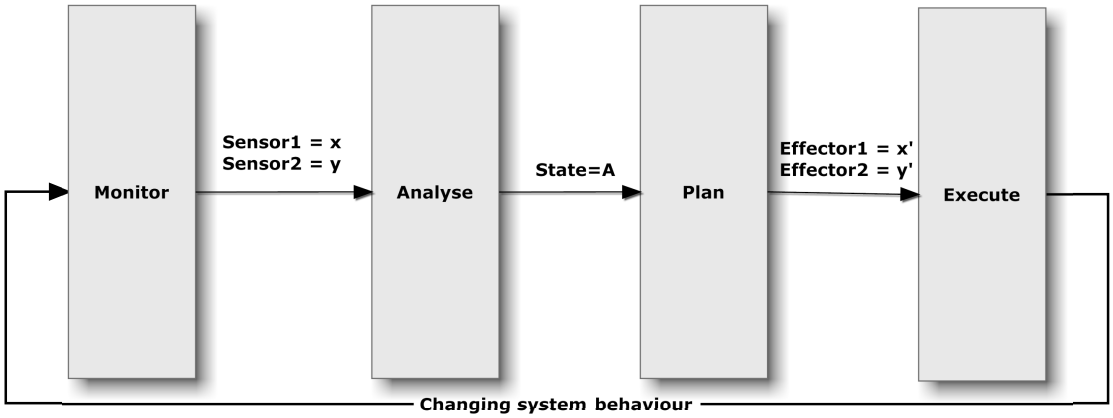


Figura 3.17: Estructura de un sistema autónomo.

(AM), que se encarga de controlar el comportamiento del nodo usando sensores y actuadores (explicados en detalle en la siguiente sección). Con el AM tenemos un nodo que puede gestionarse él mismo para tirar menos recursos. Podemos observar como funciona el prototipo en la figura 3.16.

Arquitectura del sistema

La arquitectura del nuevo nodo es básicamente la misma que antes, pero con una capa adicional que actúa como gestor de alto nivel. Podemos dividir el software en 4 partes: *General Manager* (GM), *Autonomic Manager* (AM), *Touchpoints* y, finalmente, los recursos gestionados (*Managed Resources*, MR).

General Manager Decide qué tipo de política ha de aplicarse para construir el plan (para el AM). Por el momento sólo tenemos una política: perder el mínimo numero de trabajos.

Autonomic Manager Es el corazón de un sistema autónomo (figura 3.17). Esta capa controla la monitorización, el análisis, la planificación y la ejecución de las tareas. La fase de monitorización se hace a través de *touchpoints* que unen el AM con los recursos (MR). La fase de análisis determina en qué estado está el sistema, y cuando lo sabe planifica. Esto lo hace siguiendo la política que proviene del GM, llevando al sistema hacia el estado deseado. La última fase ejecuta el plan usando los *touchpoints*.

Estados del sistema Comparando los valores de los sensores podemos identificar en qué estado está el sistema. En la implementación actual estos sensores son el número de *ServiceThreads* y la carga de CPU. Como los estados se cargan de un *XML*, podemos cambiarlos de forma sencilla.

En la fase de análisis, el AM compara los valores obtenidos en la fase de monitorización con los posibles estados para conseguir el estado actual. En este instante el sistema puede pasar a la siguiente fase (plan).

Planificar Una vez sabemos dónde estamos, necesitamos saber dónde queremos ir. La manera de hacerlo depende de la política determinada por el GM. Básicamente, el GM y el AM tienen el mismo ciclo de vida, pero a distintos niveles. El plan básicamente indica *qué hacer* según el estado en el que estamos.

Touchpoints *Touchpoints* son las interficies que nos permiten consultar y modificar los sensores y los actuadores (modifican valores del sistema). De este modo conseguimos un nivel de abstracción extra.

Managed Resources Los recursos gestionados (o MR), son aquellos recursos que controla el sistema. El AM accede a ellos usando los *Touchpoints*. En nuestro caso controlamos dos recursos, uno para el servidor y otro para el sistema operativo. En cada uno se definen los sensores y los actuadores: los sensores, en nuestro caso, son el número de *ServiceThreads* que están aceptando peticiones del cliente y la carga del sistema; los actuadores son el número de *ServiceThreads* y el número de trabajos que Globus puede ejecutar a la vez en el sistema operativo (se realiza controlando el planificador de Globus).

Con los sensores podemos determinar el estado del sistema; con los actuadores podemos modificar su comportamiento.

Ciclo de vida de un sistema auto-gestionable

Cuando se inicia Globus, el AM carga el conocimiento (los distintos estados del sistema basados en los valores de los sensores y la política para construir el plan). Una vez se ha completado el arranque, cada x segundos (5 por defecto) el gestor coge los valores de los sensores y los analiza. El resultado del análisis es el estado actual del sistema. Entonces tenemos que hacer una planificación según la política actual y, una vez hecho, el gestor lo ejecuta. Este ciclo, una instancia en forma de diagrama de secuencia, se realiza hasta que el sistema se apaga (figura 3.18).

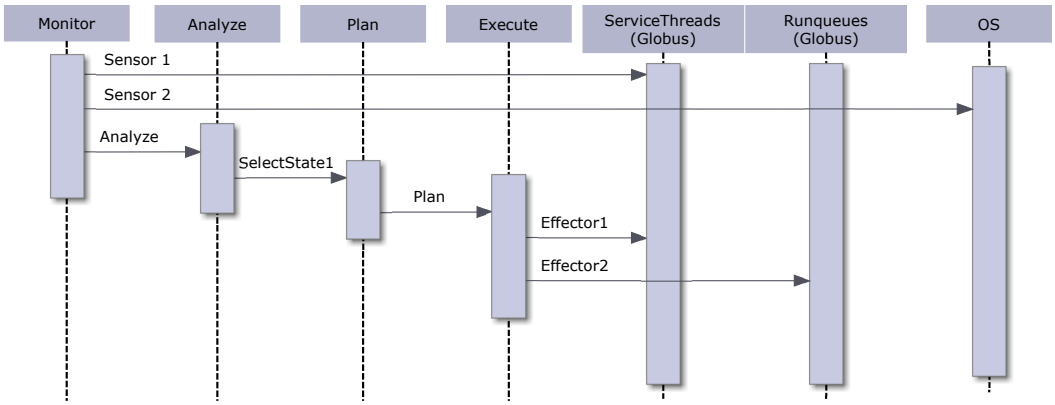


Figura 3.18: Diagrama de secuencia del prototipo.

Cuadro 3.2: Número de trabajos acabados y el % respecto al prototipo y el sistema original.

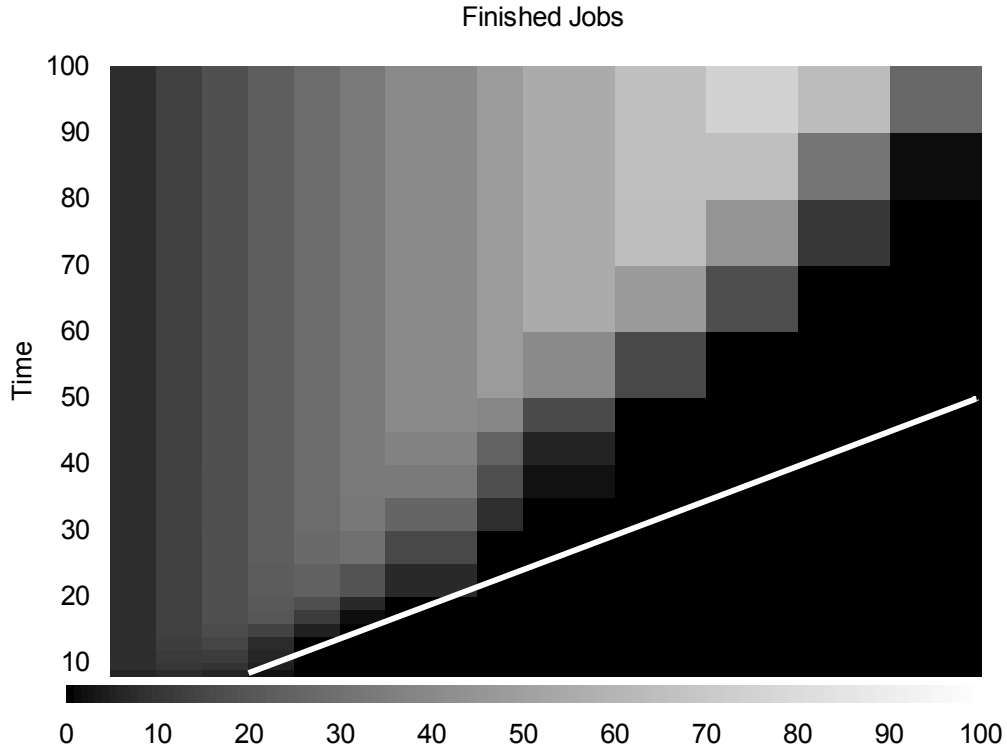
# CPU	**(F) Trabajos acabados**		**% Trabajos acabados** $\frac{F}{S}$	
	Original/Prototipo		**Original/Prototipo**	
1	1.192	1.692	9,45	13,43
2	2.376	3.410	18,85	27,05
3	3.220	4.429	25,56	35,13
4	3.504	5.004	27,79	39,69

3.7.3. Evaluación

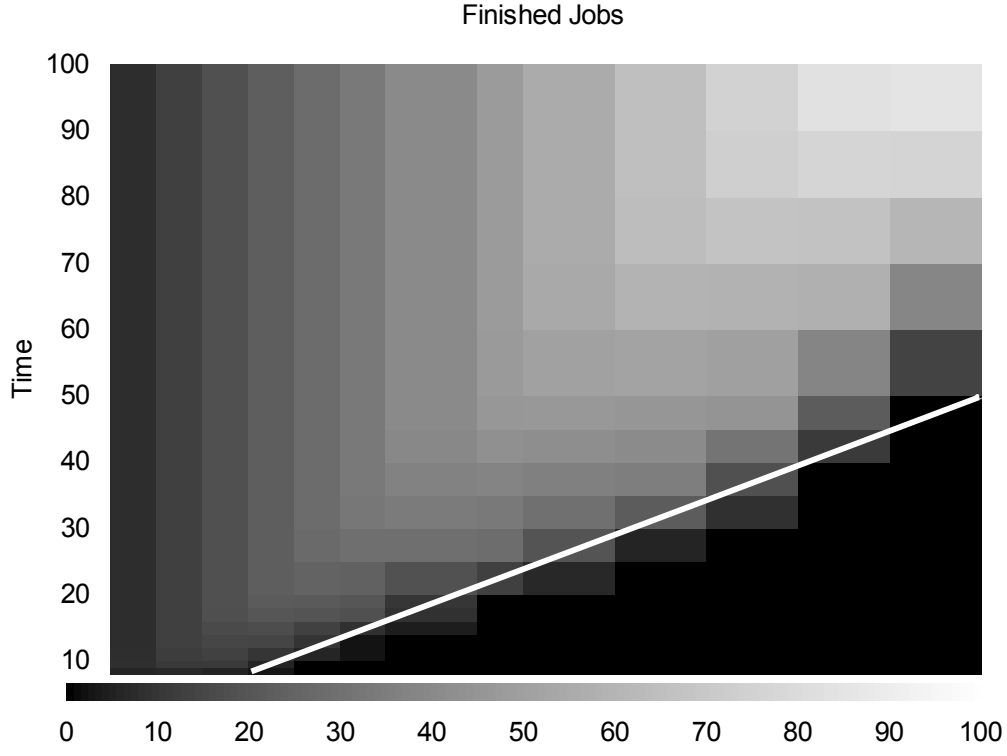
Análisis del middleware original

Cuando ejecutamos nuestro generador de carga en nuestra máquina obtenemos los resultados mostrados en la figura 3.19 (la gráfica de la subfigura (a) es del *middleware* original).

Estos gráficos muestran el número de trabajos finalizados (escala de grises para una escala de 0 a 100 siendo 100 blanco), los ejes corresponden al número de trabajos enviados (eje X, empezando en 5), y finalmente, el eje Y nos muestra el tiempo (en segundos) de la

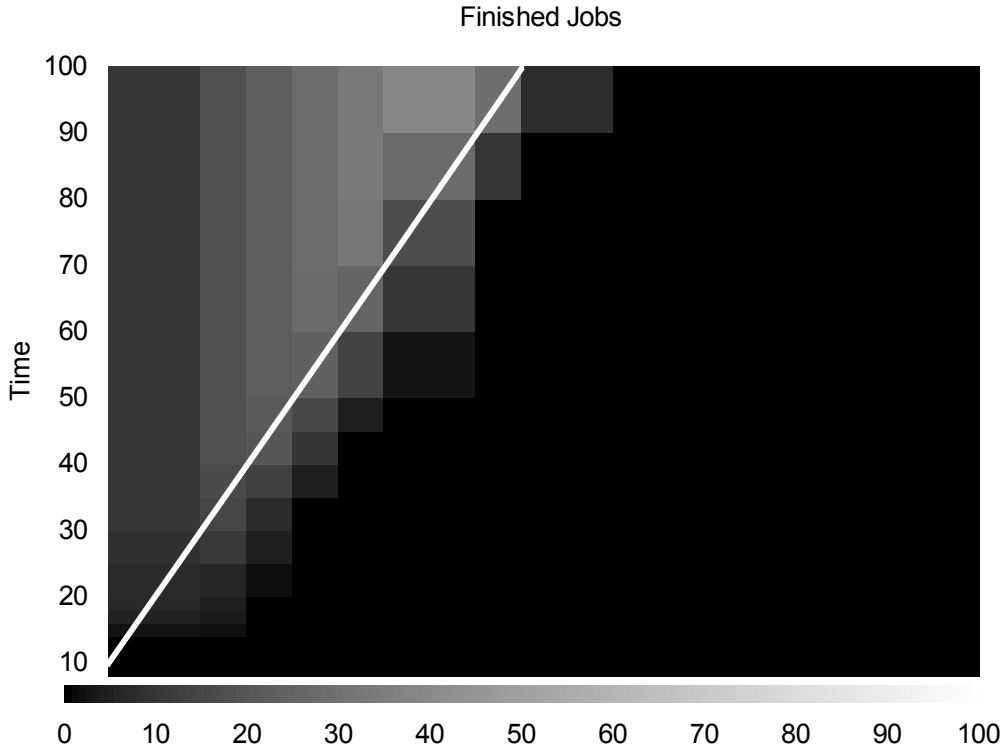


(a) Standard

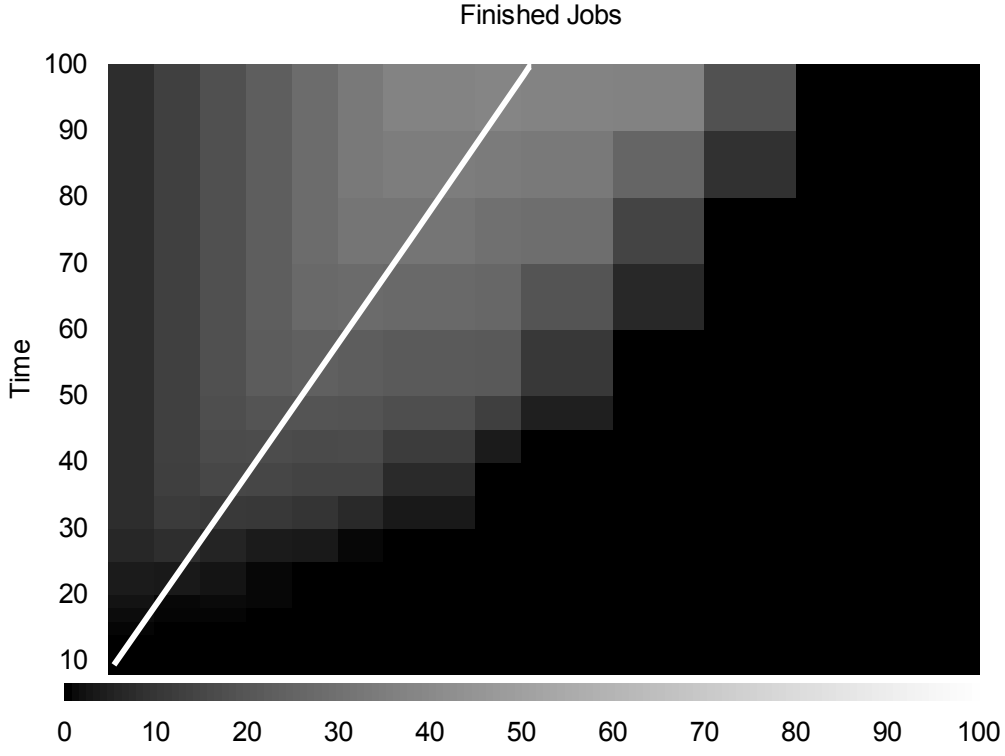


(b) Prototype

Figura 3.19: Trabajos finalizados usando 4 procesadores en el *middleware* original y nuestro prototipo. Con una linea blanca frontera teórica para la ejecución de todos los trabajos.



(a) Standard



(b) Prototype

Figura 3.20: Trabajos finalizados usando 1 procesador en el *middleware* original y nuestro prototipo. Con una línea blanca frontera teórica para la ejecución de todos los trabajos.

prueba. Lo menos deseado de estas gráficas sería el color negro. El color negro significa que no hemos sido capaces de finalizar ningún trabajo.

Como nuestra máquina tiene 4 CPUs, podemos repetir el test usando 1 (figura 3.20), 2 ó 3 CPUs (cuadro 3.2) para ver como funciona el sistema cuando estamos reduciendo los recursos. La tabla 3.2 tiene información del total de trabajos finalizados en todas las implementaciones y su % respecto el total de trabajos enviados (12.605, como hemos especificado al inicio).

Reducir los recursos es importante cuando estamos usando el *middleware* en un entorno virtualizado o en entornos con cargas heterogéneas. Los recursos que modificamos son el número de procesadores. Obligamos al *middleware* y todos sus procesos a usar uno, dos, tres o cuatro procesadores.

Como podemos ver la limitación de recursos produce grandes cambios en el número de trabajos que se pueden finalizar. Nuestro objetivo es reducir las zonas negras (zonas donde no acaba ningún trabajos). El cuadro 3.2 muestra como terminamos 3.504 trabajos (27,79 %) con 4 CPUs y 1.192 (9,45 %) con una CPU. Un vistazo a la figura 3.20 (a) muestra que enviando más de 50 trabajos llegamos a un estado donde no se acepta ningún trabajo más (ni se finaliza).

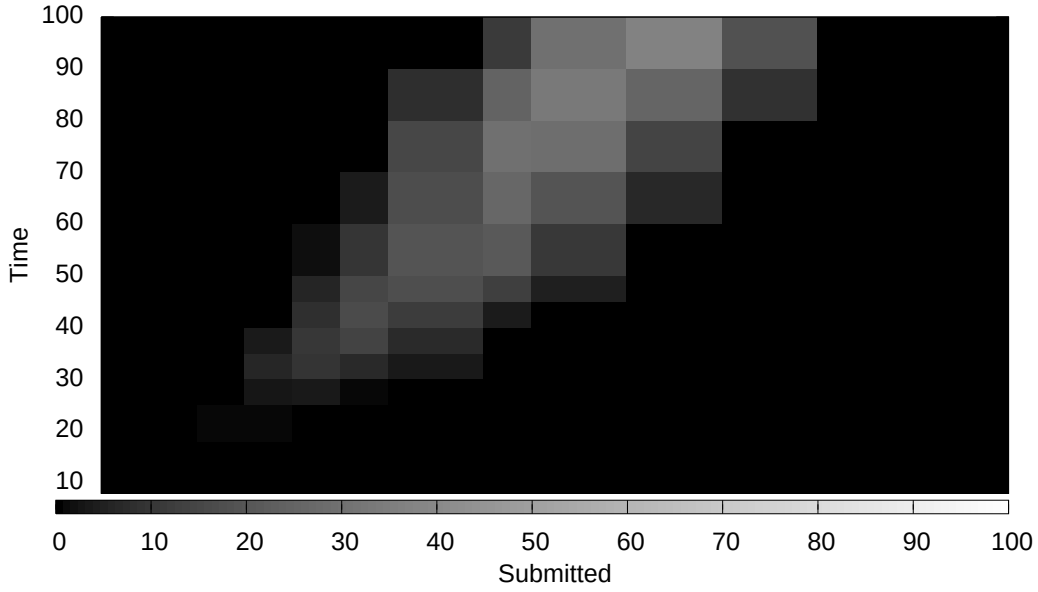
Analizando el prototipo

Hemos repetido el anterior test usando nuestro prototipo y reduciendo, de igual manera que en la anterior prueba, los recursos que damos al *middleware*. Comparando las dos gráficas (superior e inferior) de la figura 3.19 vemos que hemos sido capaces de incrementar el número de trabajos aceptados usando los mismos recursos. Dando más tiempo (más que el que muestran las gráficas) y usando 1, 2 ó 3 procesadores con el prototipo conseguimos la finalización de todos los trabajos. Los resultados del prototipo corresponden a la gráfica (b) de las figuras 3.20 y 3.19.

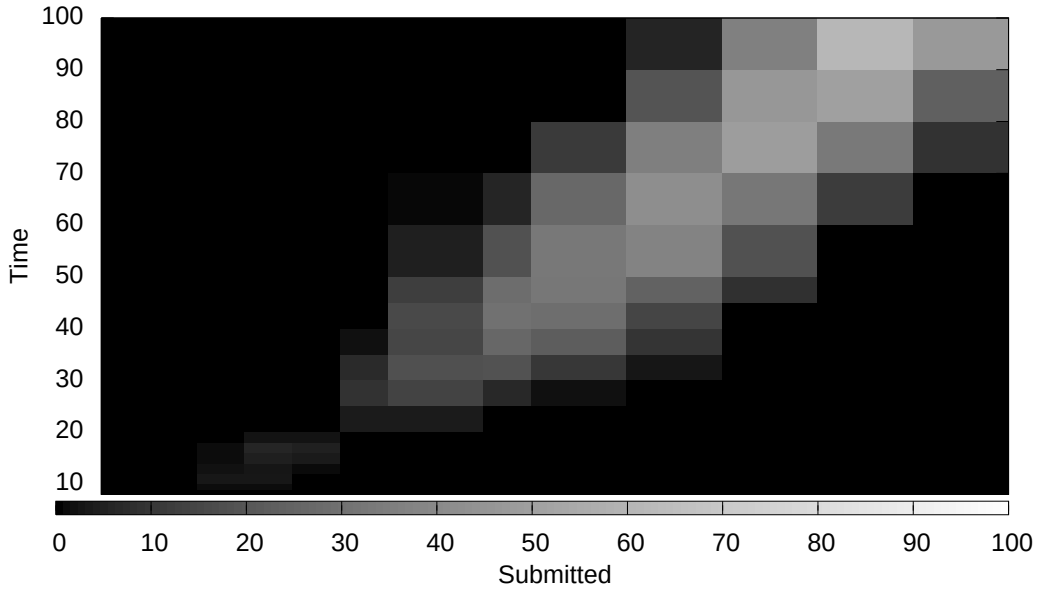
Con el prototipo hemos incrementado en un porcentaje de 39,69 % los trabajos finalizados usando 4 procesadores. Analizando la gráfica de la figura 3.20 observamos que con el prototipo podemos procesar hasta 80 trabajos (el original sólo 50). Finalmente, en la figura 3.19 vemos cumplido el objetivo de conseguir menos zonas negras.

Comparación de los resultados

Comparando los resultados del cuadro 3.2, podemos ver como hay un gran número de trabajos que siguen sin ejecutarse; en el original, el número es mayor, dado que tenemos

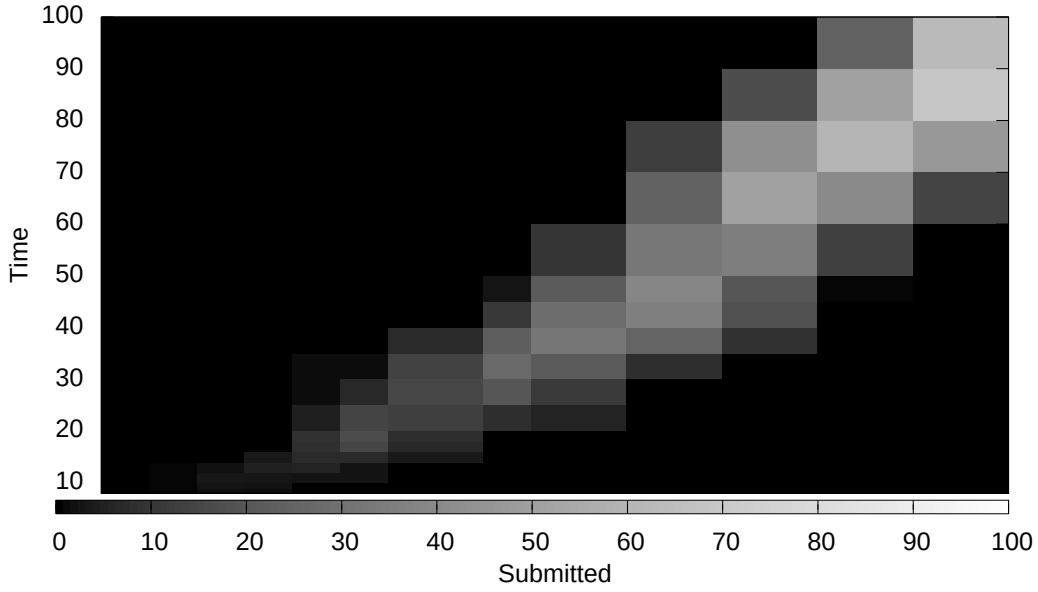


(a) 1 CPU

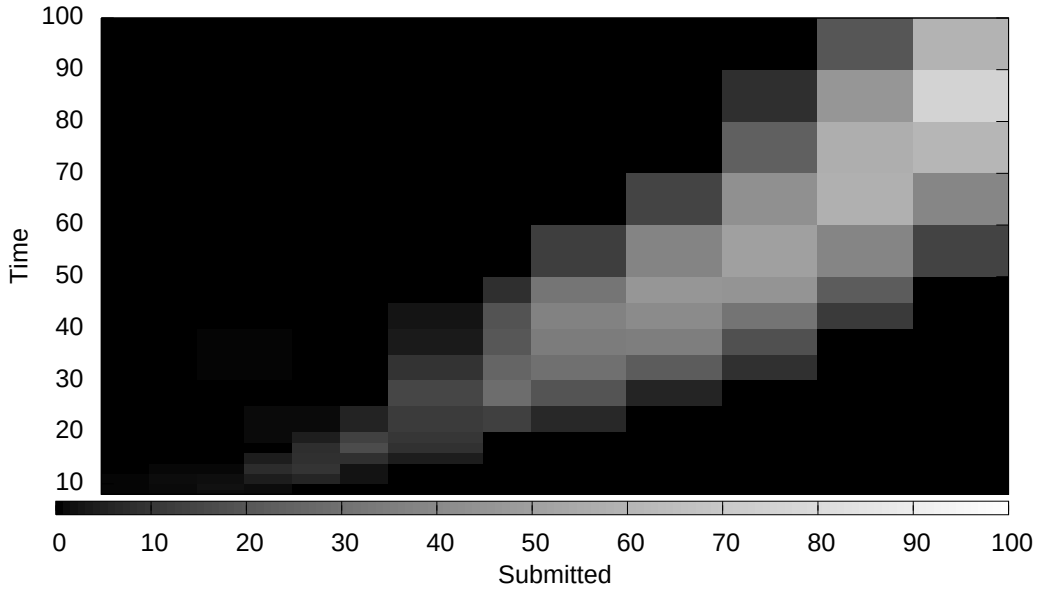


(b) 2 CPU

Figura 3.21: Trabajos finalizados en el prototipo - Trabajos finalizados en el *middleware* original (1 y 2 procesadores).



(a) 3 CPU



(b) 4 CPU

Figura 3.22: Trabajos finalizados en el prototipo - Trabajos finalizados en el *middleware* original (3 y 4 procesadores).

Cuadro 3.3: Comparación del original, el prototipo y la escalabilidad (última columna).

# CPU	% $\frac{F}{max(F_{prototype})}$ Standard/Prototipo		% $\frac{F}{max_{local}(F)}$ Standard/Prototipo	
1	23,82	33,81	34,00	33,81
2	47,48	68,15	67,00	68,15
3	64,34	88,51	91,90	88,51
4	70,00	100,00	100,00	100,00

el *timeout* del cliente afectado por la política de *PS* mostrada en la sección 3.7.1.

En el cuadro 3.3 tenemos el % de trabajos finalizados comparado con el máximo número de trabajos finalizados (5.004, usando 4 procesadores en el prototipo). Finalmente, en la última columna tenemos la escalabilidad de cada implementación usando el máximo número de trabajos finalizados en cada implementación en particular.

Como podemos ver en el cuadro 3.3, el 70 % de los trabajos están aceptados con el *middleware* original usando 4 procesadores, pero cuando usamos 3 procesadores este valor cae casi 6 puntos (64,34 %). Esto puede ser causa de un problema de escalabilidad, ya que ocurre en las dos implementaciones, y podría resolverse cambiando el código de la aceptación de clientes.

Finalmente, las figuras 3.21 y 3.22 muestran visualmente la diferencia en el número de trabajos finalizados ($prototipo - original$). Con el número mínimo de recursos (1 procesador) el prototipo consigue finalizar más trabajos, mientras que el original no puede conseguir ninguno (se pierden). En el caso de 3 y 4 procesadores las diferencias también son notables con zonas donde la diferencia supera los 70 trabajos.

3.8. Conclusiones

En este capítulo hemos analizado el rendimiento de un *middleware Grid* (Globus) utilizando monitorización mientras introducíamos mejoras en el mismo. Finalmente, hemos introducido una capa de auto-gestión para conseguir mejorar el sistema. Una cosa importante en este tipo de trabajo es comprobar si el prototipo está haciendo algo útil, adecuar las pruebas a lo que queremos probar: si nuestro objetivo es disminuir el número de trabajos perdidos (o incrementar el número de trabajos finalizados), necesitamos ser consistentes y, ya que normalmente tenemos más trabajos para ejecutar, dar más tiempo

al prototipo para que puedan finalizar. No podemos probar la implementación original durante 300 segundos y entonces probar el prototipo para la misma cantidad de tiempo. Necesitaremos darle más tiempo al prototipo, al menos hasta que los trabajos que siguen vivos finalicen.

Para resolver esta situación introducimos unas gráficas 3.19, 3.20, 3.21 y 3.22 que mapean datos en 2 dimensiones con gradiente (trabajos enviados, trabajos finalizados y tiempo); con estos datos podemos obtener información útil. En primer lugar nos ayuda a ajustar los datos preliminares del prototipo usando el comportamiento del original, y finalmente nos permite probar que el comportamiento del sistema es correcto. Hemos señalado con una linea blanca la zona donde el sistema debería ser capaz de ejecutar los trabajos (izquierda) y donde deberíamos tener problemas (derecha).

Introducir el prototipo dentro del *middleware Grid* nos ha permitido obtener importantes mejoras sobre el original: perdemos menos trabajos con los mismos recursos.

Aunque el prototipo funciona bien para cargas basadas en ráfagas (*bursty*) y escenarios similares, hemos de vigilar con el tamaño de las colas: se necesita un control de admisión que permita decidir qué clientes o trabajos dejamos entrar al sistema. El *Autonomic Manager* implementado aquí, junto con sus sensores y estados, permite integrar el trabajo en entornos compartidos, como mostraremos en el siguiente capítulo.

Para finalizar, es importante introducir un mecanismo más inteligente en este prototipo: en primer lugar, hemos de ser capaces de predecir los siguientes pasos basándonos en el análisis de los trabajos o la carga que recibimos ([Iosup et al., 2006]). Finalmente, necesitamos eliminar los estados predefinidos y generarlos usando simulación o técnicas de aprendizaje (algoritmos genéticos). Estos vacios los llenaremos con la simulación, que hemos demostrado que se puede utilizar para simular entornos similares en el capítulo (2) y como mostraremos, en un entorno *Grid*, en el capítulo 5.

Este capítulo está basado en las siguientes publicaciones:

- Nou, R., Julià, F., Carrera, D., Hogan, K., Caubet, J., Labarta, J., and Torres, J. (2006c). **Monitoring and analysing a Grid middleware node**. Proc. 7th IEEE/ACM International Conference on Grid Computing 2006, GRID 2006, Barcelona, Spain.

- Nou, R., Julià, F., Carrera, D., Hogan, K., Caubet, J., Labarta, J., and Torres, J. (2007a). **Monitoring and analysis framework for Grid middleware**. In PDP, pages 129–133. IEEE Computer Society.

- Caubet, J., Hogan, K., Nou, R., Labarta, J., and Torres, J. (2006). **Supporting the Introduction of Autonomic Computing in Middleware**. Engineering Conference

VII, IBM Academy of Technology Conference IBM Hursley , England , October 2006.

- Nou, R., Julià, F., and Torres, J. (2007c). **Should the Grid middleware look to self-managing capabilities?** The 8th International Symposium on Autonomous Decentralized Systems (ISADS 2007) Sedona, Arizona.
- Nou, R., Julià, F., and Torres, J. (2007d). **The need for self-managed access nodes in grid environments** The 4th IEEE Workshop on Engineering of Autonomic and Autonomous Systems (EASe 2007), Tucson, Arizona.
- *Submitted*, Un artículo en un número especial del journal MAGS.

Capítulo 4

Entornos compartidos autogestionables

En este capítulo introduciremos un nivel más en la abstracción de sistemas autónomos. Actualmente los entornos virtualizados plantean problemas sobre la distribución de los recursos adecuadamente (y de manera dinámica) en aplicaciones que tienen carga dinámica y no siguen un comportamiento lineal. Con el tiempo los entornos compartidos integrarán más aplicaciones de diferente naturaleza, tanto en comportamiento como en la carga, y la gestión de recursos se hará más díficil. Son entornos heterogéneos que pueden incluir aplicaciones transaccionales, *Grid*, intensivas en cálculo, batch, orientadas a la *E/S* (*streaming*)... . Nosotros nos centraremos en las aplicaciones *Grid* y las transaccionales.

Vamos a utilizar el prototipo anterior en un servidor donde hemos añadido un elemento de autogestión a nivel de sistema operativo. Este nuevo componente se encargará de distribuir los recursos entre dos *middlewares*, un Tomcat y un Globus, ambos con sistemas de autogestión (comentados en capítulos anteriores). Este componente, al igual que los anteriores que hemos presentado hasta ahora, puede utilizar la simulación para predecir. En nuestro caso podría usar la simulación de Tomcat que hemos realizado en el capítulo 2.3 junto con la simulación de Globus que comentaremos en los siguientes capítulos; de esto modo se plantea crear este tipo de entornos auto-gestionables y la utilización de pequeños kernels o plugins de simulación que reproduzcan el comportamiento de las aplicaciones o sistemas virtualizados. Veremos un prototipo de este entorno en capítulos posteriores.

En este capítulo no utilizaremos un entorno virtualizado (en el momento que realizamos los experimentos, la virtualización aún no era estable para poder realizar estudios de este tipo), con *XEN* o *VMWare* por ejemplo, sino que compartiremos el mismo servidor con dos aplicaciones heterogéneas que compiten por los recursos. En el capítulo 6 ya utilizaremos las ventajas que nos ofrece la virtualización y incluiremos la simulación en el gestor de recursos.

4.1. Introducción

La consolidación de la computación distribuida y la *Grid* se ha acompañado con la aparición de nuevos modelos de computación orientados a estos entornos. El más actual es el modelo de *cloud computing*, donde se nos ofrecen servicios sin necesidad de conocer la tecnología o dónde se ejecutan. Otro de ellos es el *utility computing model*, donde las aplicaciones se ejecutan en plataformas que alquilan sus recursos. Los dueños de las aplicaciones pagan por sus recursos, obteniéndolos con garantías de calidad de servicio (*QoS*) que pueden expresarse utilizando un contrato de servicio (*Service Level Agreement, SLA*). La plataforma es responsable de ofrecer los recursos necesarios a cada aplicación para cumplir con su workload, o al menos satisfacer su *SLA*. Estas plataformas de hosting deben ofrecer recursos a aplicaciones heterogéneas que pueden ir desde aplicaciones web (un servidor de aplicaciones con una carga transaccional) hasta aplicaciones científicas, usando una plataforma *Grid*.

La propuesta más utilizada en las plataformas de hosting es obtener un subconjunto separado de los nodos de un clúster para cada aplicación (modelo dedicado [Appleby et al., 2001]). En este modelo la gestión de los recursos se hace con una granularidad de cluster y la técnica de reparto de recursos debe determinar cuántos nodos ha de asignar para cada aplicación; razones económicas (espacio, consumo eléctrico, refrigeración y coste) hacen más apetecible el modelo compartido [Chandra et al., 2003a]: los recursos de un nodo pueden compartirse por múltiples aplicaciones; en este caso la repartición de recursos necesita determinar cómo dividir los recursos entre todas las aplicaciones (que compiten). Nosotros mostraremos una nueva propuesta capaz de ejecutar aplicaciones heterogéneas en un entorno compartido de manera dinámica; repartiremos los recursos de la plataforma (nos centramos en la CPU en este prototipo) mientras mantenemos un buen rendimiento.

Este capítulo extiende el trabajo realizado en [Guitart et al., 2006]. La anterior propuesta propone una estrategia global que impide la sobrecarga de aplicaciones web y que utiliza los recursos de manera eficiente en un entorno compartido, ejecutando aplicaciones homogéneas. Esta propuesta explota la capacidad de variar los recursos dinámicamente segun la carga de cada aplicación, cuyos beneficios se han descrito en estudios recientes [Appleby et al., 2001, Chandra et al., 2003a, Chandra et al., 2003b]. El objetivo es conseguir los recursos necesarios de la aplicación bajo demanda y adaptarse a sus cambios; para ello se requiere una colaboración entre las aplicaciones y el gestor (comunicación bidireccional). En este capitulo presentamos la evaluación y ampliación de la anterior propuesta utilizando aplicaciones heterogéneas.

Este gestor de recursos puede utilizarse en un entorno virtualizado ganando flexibili-

dad.

4.2. Estrategia para la repartición de recursos

Nuestra propuesta utiliza el concepto de comunicación bidireccional. En muchas situaciones de escasez de recursos (ya sea en un entorno informático o en casos como el consumo de agua o el consumo eléctrico) la comunicación con las aplicaciones (o usuarios) de los recursos disponibles, para que éstas puedan adaptarse al entorno, junto con la comunicación por parte de las aplicaciones de los recursos necesarios, para poder repartirlos de manera inteligente, ofrecen muchas mejoras, como veremos en los resultados obtenidos. Nuestra propuesta está basada en un gestor de procesadores global, llamado *eDragon CPU Manager* (*ECM*), responsable de distribuir periódicamente los procesadores disponibles entre las distintas aplicaciones que se ejecutan en la plataforma destino. Se pueden encontrar más detalles en [Guitart et al., 2006].

En los escenarios que estamos estudiando, el procesador es el recurso más limitado. *ECM* coopera con las aplicaciones para gestionarlas de manera eficiente y prevenir que las aplicaciones lleguen a sobrecargarse usando, una comunicación bidireccional.

Las aplicaciones, piden periódicamente a *ECM* el número de procesadores que necesitan para cumplir con la carga que les llega (sin perder QoS), y definimos el número de procesadores pedidos por una aplicación i como R_i. Por otro lado, también pueden preguntar en cualquier momento cuántos procesadores tienen asignados; definimos el número de procesadores asignados a una aplicación i como A_i.

Con esta información las aplicaciones pueden adaptar su comportamiento a los recursos obtenidos (concretamente la capa autogestionada, presentada en el capítulo anterior), evitando la degradación de su servicio. La figura 4.1 muestra un diagrama que describe su funcionamiento.

4.2.1. eDragon CPU Manager

El *eDragon CPU Manager* (*ECM*) es el responsable de la distribución de los procesadores entre las aplicaciones de la plataforma. *ECM* está implementado como un proceso a nivel usuario que despierta periódicamente con un *quantum* de k_{ECM}, examina las peticiones de las aplicaciones y distribuye los procesadores según la política de planificación definida. Con esta configuración no necesitamos modificar el *kernel* para demostrar la utilidad del entorno propuesto.

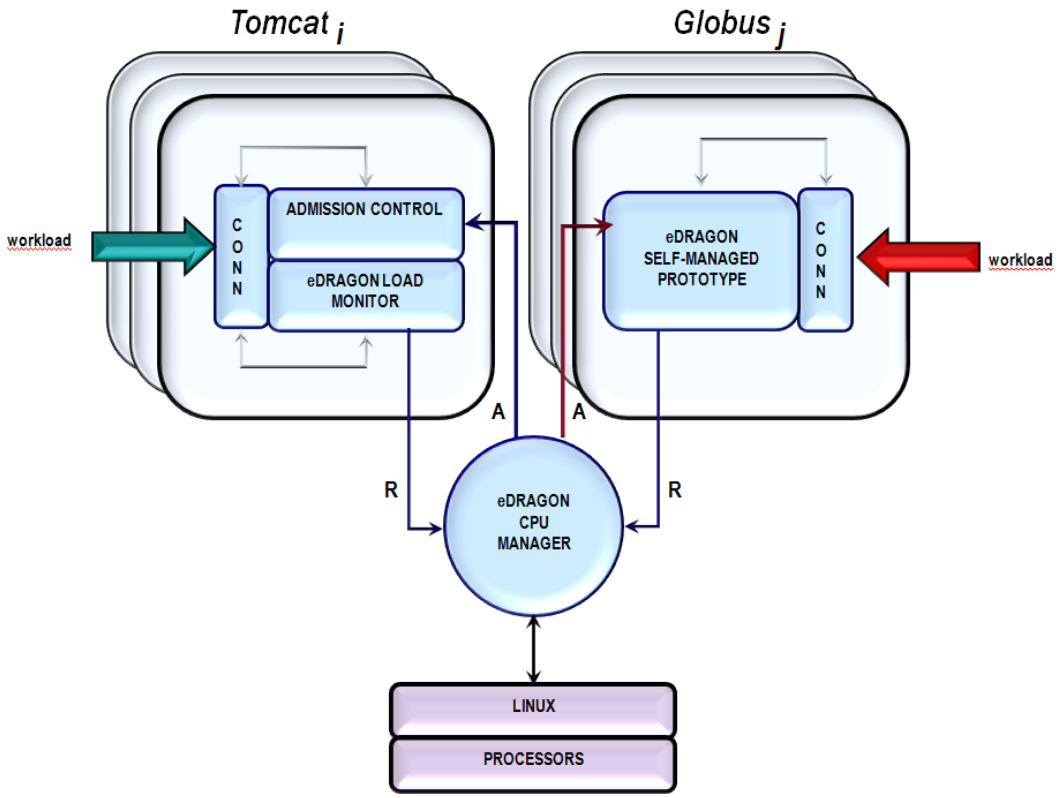


Figura 4.1: Estructura del prototipo con ECM y dos aplicaciones.

Normalmente las políticas de asignación de recursos utilizan métricas como el tiempo de respuesta, el throughput o la disponibilidad. A pesar de que estas métricas son importantes, en aplicaciones de e-commerce también lo son las ganancias y los beneficios, que deberían incorporarse al diseño de políticas [Cherkasova and Phaal, 2002]. Por esta razón, *ECM* puede implementar políticas considerando todos estos indicadores. También podemos tener en cuenta políticas como las definidas en la sección 4.5. Nuestra política incluye prioridades por clases; la prioridad P_i indica la prioridad del dominio de un cliente en relación a otros clientes.

En nuestra política, en cada k_{ECM}, cada aplicación i recibe el número de procesadores ($A_i(k_{ECM})$) proporcionales a su petición ($R_i(k_{ECM})$), ponderada según la prioridad de la clase de la aplicación (P_i), el número de procesadores en la plataforma ($NCpus$), e inversamente proporcional a la carga total del sistema ($\sum P_j * R_j(k_{ECM})$), expresada como la suma de todas las peticiones de las aplicaciones. Podemos encontrarla más detallada en el algoritmo 4.2.1.

Algorithm 4.2.1: RESOURCE PROVISIONING()

comment: Executed at every k_{ECM}

$resourcesUsed \leftarrow 0$
for each $A_i, R_i \in Application$
do $\begin{cases} A_i \leftarrow R_i \\ resourcesUsed \leftarrow R_i + resourcesUsed \end{cases}$

if $resourcesUsed > resourcesAvailable$
then $A_i \leftarrow Round \left[\frac{P_i * R_i * NCpus}{\sum_{j=1}^{N} P_j * R_j} \right]$
EXECUTEASSIGNATION(k_{ECM})

La política de planificación debe permitirnos conseguir el nivel más alto de utilización de los recursos en la plataforma. Nuestra propuesta cumple con esto, ya que *ECM* se basa en compartir procesadores entre las aplicaciones de forma más inteligente: *ECM* no sólo decide cuántos procesadores asignar a cada aplicación sino, que además decide qué procesadores asignar. Para cumplir esto, *ECM* configura la máscara de afinidad de *CPU* de cada aplicación (usando la función de *Linux sched_setaffinity*) de modo que la asignación de procesadores hacia las distintas aplicaciones no se solapan (excepto si 1 procesador está compartido), minimizando las interferencias entre aplicaciones.

4.2.2. Comunicación entre las aplicaciones y ECM

La comunicación entre las dos partes se implementa usando una zona de memoria compartida. La información que se comparte incluye el número de procesadores que cada aplicación quiere en cada momento (R_i y R_j) y el número de procesadores que están asignados para cada aplicación por *ECM* (A_i y A_j). Usando *JNI* [Chen and Mohapatra, 2003] podemos comunicarnos con *ECM* para pedir o consultar el número de procesadores asignados.

4.3. Entorno experimental

Tenemos a Tomcat v5.0.19 [Amza et al., 2002] y un Globus GT 4.0.1 [Sotomayor and Childers, 2005] en el mismo nodo.

La carga del cliente se genera usando un generador de carga y la herramienta *Httperf* [Crovella et al., 1999], usando *RUBiS* (Rice University Bidding System) [Coarfa et al., 2002] como aplicación. La instancia de Tomcat tiene una carga variable que se muestra en la primera subfigura de 4.3, que muestra el número de nuevos clientes por segundo que atacan el servidor como función del tiempo. La distribución de la carga representa diferentes niveles de consumo de procesador cuando se ejecuta junto la carga *Grid*.

El servidor de Globus no tiene ninguna modificación de sus parámetros por defecto (número de *ServiceThreads* o el número de *RunQueues*) en las pruebas normales (con *ECM* estos valores se pueden modificar dinámicamente).

El generador de carga de Globus envía trabajos que sobrecargan o estresan el código encargado de procesarlos. Asumimos que el trabajo se ejecutará en otro nodo o clúster. Desde el generador de carga generamos y enviamos trabajos con una carga incremental, como la utilizada en el capítulo 3, de manera que se consiguen distintos perfiles de consumo de CPU y de sobrecarga. Medimos el throughput obtenido (trabajos finalizados) y lo mostramos en la figura 4.3. Podemos ver como las combinaciones entre las dos cargas son lo suficientemente variadas para mostrar los beneficios de nuestra propuesta. La plataforma donde se ejecuta es un Intel Xeon 1.4 GHz con 4 procesadores y 2 GB de RAM.

Para este prototipo, la modificación de las aplicaciones es necesaria para conseguir que se comuniquen con *ECM* (y obtener todas las ventajas de la comunicación bidireccional); aun así, podríamos utilizar otros mecanismos para evitar estas modificaciones (por ejemplo, usar un proxy). Más detalles de la arquitectura y de las modificaciones realizadas en Tomcat y Globus pueden encontrarse en [Guitart et al., 2006] para Tomcat y en [Nou et al., 2007d, Nou et al., 2007c, IBM-Corporation, 2004] para Globus (así como en los capítulos anteriores). Concretamente, en el caso de Globus, podemos ver el diagrama de secuencia correspondiente a la comunicación con *ECM* en la figura 4.2.

4.4. Evaluación

En este apartado mostraremos los beneficios de nuestra propuesta para gestionar los recursos de manera eficiente y prevenir que el servidor se sobrecargue. Los requerimientos

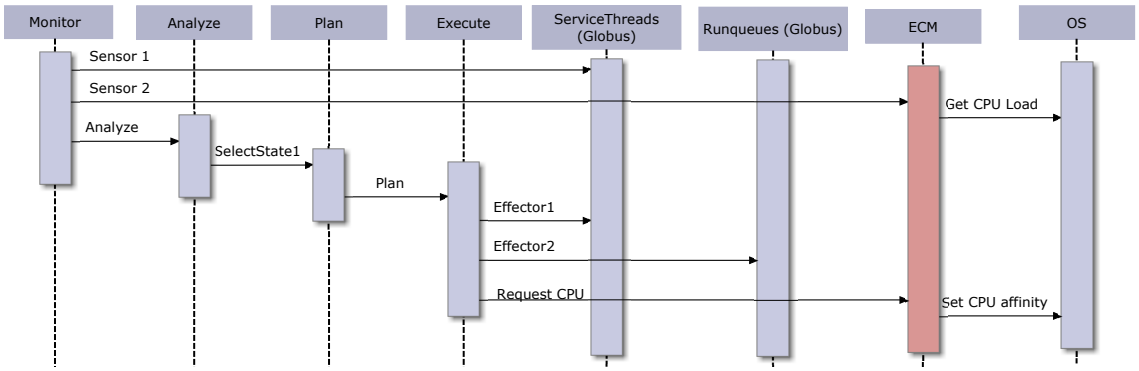


Figura 4.2: Ampliación del prototipo de autogestión de Globus para comunicarse con ECM.

Cuadro 4.1: Resumen de los datos obtenidos; podemos encontrar el numero total de peticiones finalizadas (en Tomcat) y el *throughput* total obtenido en Globus cuando estamos usando *ECM* y cuando usamos el entorno original (sin *ECM*). Finalmente consideremos un escenario sin prioridades y otro con prioridades.

		Orig	**ECM**	**% mejora**
Sin prioridades	Globus	397 tr. finalizados	1.162 tr. finalizados	292 %
	Tomcat	129.886,3 resp.	153.710,4 resp.	118 %
Con prioridades	Globus	397 tr. finalizados	1.043 tr. finalizados	262 %
	Tomcat	129.886,3 resp.	181.230,4 resp.	139 %

para preparar un trabajo de Globus son 10 veces superiores a los requerimientos para procesar una peticion de Tomcat inicial (*SSL Handshake*) y 100 veces superiores a una petición sin *Handshake*.

4.4.1. Aplicaciones sin modificar

En la figura 4.3, podemos ver con un tramado como evoluciona el sistema (los dos gráficos inferiores) cuando enviamos distintas cargas (gráfica superior) a Tomcat y Globus. Hemos dividido el test de manera que podamos probar distintas peticiones de CPU en las dos aplicaciones.

Como podemos ver en la gráfica superior, podemos dividir el test en 2 partes: primero

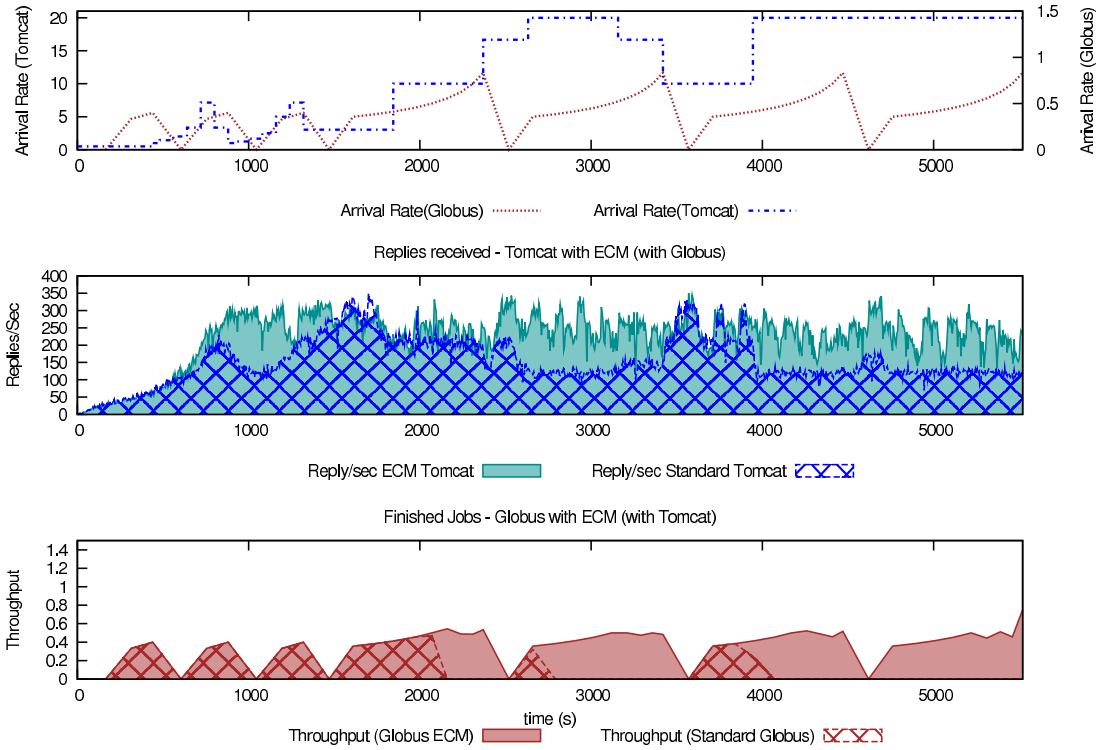


Figura 4.3: De arriba a abajo: Carga de Tomcat y de Globus, respuestas por segundo de un Tomcat sin autogestión a otro con *ECM* y *throughput* de Globus estandard comparado con el *throughput* obtenido con *ECM*. Las dos aplicaciones están ejecutándose a la vez usando ECM.

una carga ligera y posteriormente una carga más importante. Si miramos más detalladamente a Tomcat podemos ver que cuando el servidor está sobrecargado, el *throughput* (reply/sec) cae (por ejemplo, cerca de los 4.000 segundos en gráfica con tramado). Podemos observar otra zona en Tomcat con un comportamiento similar: cerca de los 1.500 segundos el *throughput* de Tomcat es muy elevado, pero si miramos cerca de los 3.000 segundos podemos ver que el throughput es inferior con una carga mayor. Si cambiamos a Globus podemos ver como hay muchas zonas en las que ningún trabajo ha finalizado (2.800-3.600 o 4.000-5.000). Globus tiene una carga demasiado elevada, aun así veremos como con *ECM* podemos mejorar el rendimiento.

Podemos encontrar un resumen de los datos (en términos de respuestas por segundo para tomcat y de *throughput* para Globus) obtenidos en algunos experimentos en la tabla 4.1.

El servidor, como hemos comentado anteriormente, necesita compartir los recursos entre las dos aplicaciones, pero dichas aplicaciones no saben nada acerca de qué entorno y con qué clase de recursos cuentan. Las aplicaciones compiten por una serie de recursos limitados, obteniendo finalmente peor rendimiento que si dividiéramos las dos aplicaciones entre dos máquinas con la mitad de recursos. Dar y recibir información acerca de los recursos que consume la aplicación y los que tiene disponibles el sistema es necesario para evitar esta situación; además podríamos utilizar una alternativa basada en predicción.

4.4.2. Middleware adaptativo con ECM

Si repetimos el último test con *ECM* (sin prioridades) obtenemos los resultados (con color sólido y sin tramado) de la figura 4.3. Podemos ver que el rendimiento general del sistema es mejor. Además observamos un punto importante: no tenemos los niveles bajos de *throughput* de Tomcat y aumentamos el número de trabajos finalizados obtenidos en la anterior prueba.

En algunas zonas podemos ver como tomcat trabaja mejor que antes; tan pronto como *ECM* recibe la petición de Tomcat acerca del número de procesadores que necesita, *ECM* responde a Tomcat cuántas CPUs le ha asignado. Tomcat puede controlar estas situaciones gracias al control de admisión presentado en el capítulo 2 para estabilizarse al nivel deseado. Es el caso de las zonas cerca de 1.000 segundos y las zonas comprendidas entre 2.200-3.500 y 4.000-5.500, donde el sistema sin *ECM* obtiene menos rendimiento que con *ECM*. En estas zonas, al conocer los recursos reales que tenemos podemos adaptar nuestro comportamiento y políticas.

En las zonas enumeradas antes podemos ver cómo trabaja Globus: en la zona cerca de los 1.000 segundos Globus trabaja de igual manera que antes; en la zona de los 2.500 pasa lo contrario, Globus aumenta su rendimiento. Cuando las dos aplicaciones están sobrecargadas, la comunicación con el sistema operativo para conocer los recursos disponibles causa un incremento del rendimiento de ambas aplicaciones, como podemos ver después de los 4.000 segundos. Por otro lado podemos modificar el reparto de procesadores usando prioridades; en esta prueba ambas aplicaciones tienen la misma prioridad, de este modo podríamos dar más prioridad a Tomcat si eso mejora nuestros ingresos, por ejemplo.

4.5. Trabajo relacionado

Estudos recientes [Andrzejak et al., 2002, Chandra et al., 2003b, Chandra and Shenoy, 2003] muestran los beneficios de ajustar dinámicamente la asignación de recursos para manejar cargas variables. Esta premisa ha motivado las propuestas de diferentes técnicas para gestionar los recursos dinámicamente para aplicaciones que utilizan una plataforma de recursos bajo demanda.

Según los mecanismos utilizados para decidir la asignación de recursos, estas propuestas pueden clasificarse en: propuestas de *control theory* usando *feedback* [Abdelzaher et al., 2002], propuestas *open-loop* utilizando modelos de colas [Chandra et al., 2003a, Doyle et al., 2003, Liu et al., 2001] y propuestas basadas en la observación, que usan medidas obtenidas en tiempo real para calcular la relación entre los recursos y el objetivo de calidad de servicio [Pradhan et al., 2002].

Las soluciones basadas en *control theory* necesitan entrenar al sistema en distintos puntos para determinar los parámetros de control para una determinada carga. Los modelos de colas son útiles para el análisis de estados estables, pero no sirven para predecir el tipo de entorno donde estamos. Las propuestas basadas en observación son adecuadas para cargas variables y comportamientos no lineales.

La gestión de recursos en una máquina simple se cubre en [Banga et al., 1999], donde los autores proponen el uso de contenedores de recursos como una abstracción del sistema. En [Liu et al., 2005] los autores proponen el diseño de algoritmos de control basados en el *feedback* (*online*) para ajustar dinámicamente los valores de un contenedor de recursos en un servidor compartido por múltiples aplicaciones. El problema de la asignación de recursos en clústeres está estudiado en [Appleby et al., 2001, Ranjan et al., 2002], asignando máquinas completas (modelo dedicado), y en [Chandra et al., 2003a, Pradhan et al., 2002, Uragonkar and Shenoy, 2004] compartiendo los recursos del nodo con múltiples aplicaciones (modelo compartido).

Cataclysm [Uragonkar and Shenoy, 2004] realiza control de sobrecarga utilizando control de admisión, degradación del servicio adaptativo y asignación de recursos dinámicamente, demostrando que el modo más efectivo de manejar la sobrecarga es considerar una combinación de estas técnicas. En este aspecto su trabajo es similar a nuestra propuesta. Hay otras propuestas [Menascé, 2005] que usan entornos virtuales y utilizan métodos analíticos para ajustar los recursos que se le dan a cada entorno virtual. *R-Opus* [Cherkasova and Rolia, 2006] trabaja en una capa distinta, y en una escala de tiempo diferente. En nuestra propuesta nos centramos en un servidor *SMP* que comparte aplicaciones y tiene una escala de tiempo más pequeña. Además, dar más CPU a una aplicación, como Tomcat, no produce directamente mejor rendimiento. La aplicación debe saber cuántos

recursos tiene disponibles.

4.6. Conclusiones

Hemos podido ver en este capítulo que es necesario gestionar los recursos de una forma inteligente. Gracias a ello hemos podido aumentar el rendimiento de dos aplicaciones, compitiendo por los mismos recursos, en un 300 %. Este entorno puede alimentarse de datos generados gracias a la predicción para conseguir mejorar el comportamiento del sistema.

Por ejemplo, podemos utilizar la simulación/predicción de un sistema complejo, como un servidor de aplicaciones. Además, podríamos utilizar la predicción en un entorno autogestionable, como el que hemos presentado bajo *Globus Toolkit 4*, en el que podríamos realizar un control de admisión utilizando la simulación como herramienta de decisión. Finalmente, y utilizando el trabajo introducido en este capítulo, podemos utilizar la simulación de Tomcat y la simulación que presentaremos posteriormente de *Globus Toolkit* para gestionar de manera inteligente los recursos de las aplicaciones a nivel de servidor o *dominio 0* en un entorno virtualizado; componentes ligeros de predicción pueden instalarse en distintas zonas para conseguir un entorno más inteligente. Veremos este uso en el capítulo 6.

En el siguiente capítulo mostraremos la construcción de un gestor de recursos que sea capaz de controlar cuestiones de calidad de servicio (*QoS*) utilizando la predicción en un entorno *Grid*.

Este capítulo está basado en el siguiente artículo:

- Nou, R., Julià, F., Guitart, J., and Torres, J. (2007b). **Dynamic resource provisioning for self-adaptive heterogeneous workloads in SMP hosting platforms**. International Conference on E-business (2nd) ICE-B 2007, Barcelona, Spain.

Capítulo 5

Gestor de recursos con QoS utilizando predicción online

En este capítulo presentamos cómo se pueden generar dinámicamente modelos para usarlos en la predicción del rendimiento en tiempo real. Empezaremos con un entorno no virtualizado y acabaremos realizando los últimos experimentos (reconfiguración dinámica) con virtualización. La virtualización nos facilita la experimentación reduciendo los recursos necesarios y facilitando el mantenimiento del entorno.

5.1. Introducción

Al aumentar el uso de entornos *Grid* y entrar en el ámbito comercial, la calidad de servicio (*QoS*) y los problemas de rendimiento se convierten en preocupaciones mayores. Para garantizar que la *QoS* esté continuamente satisfecha, el *middleware* debe ser capaz de predecir el rendimiento de la aplicación mientras se está ejecutando, para decidir cómo distribuir la carga con los recursos disponibles. Una manera de conseguir esto es utilizar modelos de predicción en tiempo real, que se generan y se analizan en tiempo real.

Para conseguir el máximo rendimiento, el *middleware* debe ser lo suficientemente inteligente para planificar los procesos de tal manera que la carga esté balanceada entre todos los recursos y que todos sean utilizados equitativamente. Además, para garantizar que las necesidades de QoS están satisfechas, el *middleware* debe ser capaz de predecir el rendimiento de la aplicación en el momento de decidir cómo distribuir la carga entre los recursos disponibles. Las capacidades de predicción son prerequisitos para implementar un gestor de recursos inteligente que tenga en cuenta la QoS, así como los mecanismos de

control de admisión. La predicción del rendimiento en el contexto de sistemas empresariales tradicionales se realiza, normalmente, usando modelos de rendimiento que capturan los detalles más importantes del comportamiento del sistema [Menascé et al., 2004a]. Hay numerosas técnicas para predecir y planificar sistemas distribuidos convencionales, la mayoría basadas en sistemas análiticos o de simulación, que han sido desarrollados y usados en la industria. Estas técnicas asumen de manera general que el sistema es estático y que hay una serie de recursos dedicados usados, particularidades que como hemos visto en los capítulos anteriores no se dan de manera común.

Para solucionar la necesidad de predecir el rendimiento en los entornos *Grid*, necesitamos nuevas técnicas que usen modelos de predicción generados en tiempo real y que muestren los cambios en el entorno. El término *online performance models* fue utilizado para este tipo de modelos [Menascé et al., 2005]. El uso *online* de estos modelos difiere del uso tradicional en la planificación en que las configuraciones y la carga son analizadas de manera que reflejan el sistema real en cortos periodos de tiempo.

Como este análisis se realiza en tiempo real, es esencial que el proceso de generar y analizar estos modelos sea completamente automatizado.

En este capítulo presentamos un estudio realizado con Globus Toolkit [Foster, 2005], uno de los proyectos más importantes para generar infrastructuras *Grid*. Hemos ampliado el *middleware* con un componente de predicción online que puede llamarse en cualquier momento para predecir el comportamiento del *Grid*, dado una determinada asignación de recursos y una estrategia de balanceo de carga. El estudio muestra cómo podemos generar los modelos dinámicamente para ofrecer capacidades de predicción en línea. Utilizamos modelos jerárquicos de Petri Nets con colas (*QPN*) generados dinámicamente para reflejar la configuración del sistema y la carga. Las *QPN* hacen posible la modelización del sistema (cuando no está saturado), así como del comportamiento de la gestión de recursos y de nuestro mecanismo de equilibrado de carga, combinando aspectos del software y del hardware del sistema.

Además, las *QPN* han mostrado su utilidad en modelar sistemas de componentes distribuidos en [Kounev, 2006], que se usan comúnmente para construir bloques de *Grid* [Foster and Kesselman, 2003].

Finalmente, evaluamos la calidad de nuestro mecanismo de predicción y presentamos algunos resultados que demuestran la efectividad y su uso práctico. La estructura de gestor presentada en este capítulo puede usarse como base para implementar mecanismos inteligentes para la asignación de recursos, teniendo en cuenta la calidad de servicio, el balanceo de carga y el control de admisión. Además, aunque nuestra propuesta está orientada a los entornos *Grid*, no está limitada en ningún sentido y puede aplicarse en el contexto de arquitecturas orientadas al servicio (*SOA*) más generales.

5.2. Simulación de una plataforma Grid

Como hemos mostrado en el capítulo 3, aun teniendo una gran popularidad e importancia, la implementación actual de Globus demuestra un rendimiento pobre y poca disponibilidad cuando el *middleware* está sobrecargado. En el trabajo anterior [Nou et al., 2007a, Nou et al., 2007c] hemos estudiado el comportamiento de Globus bajo una carga elevada y hemos propuesto una ampliación del *middleware* utilizando una capa de autogestión para mejorar su rendimiento y estabilidad [Nou et al., 2007d]. En este capítulo mostraremos como el *middleware* puede mejorarse utilizando predicción online, necesaría para implementar mecanismos de calidad de servicio inteligentes.

5.2.1. Modelado del sistema

Formalmente un entorno *Grid* puede representarse con una tupla $G = (S, V, F, C)$ donde:

S $= \{s_1, s_2, ..., s_m\}$ es el conjunto de servidores *Grid*,

V $= \{v_1, v_2, ..., v_n\}$ es el conjunto de servicios ofrecidos por los servidores,

F $\in [S \longrightarrow 2^V]$[1] es una función que asigna un conjunto de servicios a cada servidor *Grid*. Como los *Grids* son típicamente heterogéneos, asumimos que, dependiendo en qué plataforma se ejecuten, los servidores *Grid* podrían ofrecer distintos subconjuntos de los servicios,

C $= \{c_1, c_2, ..., c_l\}$ es el conjunto de las sesiones de clientes activas. Cada sesión $c \in C$ es una tupla (v, λ) donde $v \in V$ es el servicio que se pide y λ es el perfil de carga de las peticiones al servicio del cliente (peticiones/segundo).

Mecanismos de planificación

Hemos implementado un distribuidor de peticiones de servicio configurable (*service request dispatcher (SRD)*) que ofrece una planificación flexible de un mecanismo de balanceo de carga para las peticiones de servicio. Se asume que por cada sesión del cliente, hay un número de threads (desde 0 a ilimitado) asignados a cada servidor *Grid* que ofrece el servicio. Las peticiones son balanceadas entre todos los servidores según la

[1] 2^V es el conjunto de todos los subconjuntos posibles de V.

disponibilidad de threads. Esta clase de configuración de los threads limita la petición concurrente de servicios en cada servidor, de manera que podemos disponer de distintas estrategias de planificación. Una estrategia de planificación puede representarse por una función $T \in [C \times S \longrightarrow \mathbb{N}_0 \cup \{\infty\}]$ que se llamará *función de asignación de threads*.

El SRD encola las peticiones de servicios (como parte de una sesión de un cliente) y las planifica para ser servidas por los servidores *Grid* cuando haya threads disponibles. Hay que destacar que los threads son usados como una entidad *lógica* para garantizar el nivel de concurrencia deseado en cada servidor. La gestión de los threads se realiza por el *SRD*, por lo que los servidores *Grid* no han de saber nada acerca de las sesiones de los clientes y cuántos threads tienen asignados. Mientras el *SRD* puede usar una semántica diferente, hablando de threads físicos y threads lógicos de una sesión, no se requiere en la arquitectura, y hay muchas maneras de no tener que hacerlo para mejorar el rendimiento. Para una escalibilidad mayor se pueden instanciar múltiples *SRD* y distribuirlos en varias máquinas si se necesita.

SRD separa los clientes *Grid* de los servidores *Grid*, cosa que ofrece importantes ventajas que son relevantes en entornos comerciales. Primero, la separación nos introduce un balanceo de carga *de grano fino* en el nivel de servicio y no en el nivel de sesión. Segundo, *SRDs* hacen posible el balanceo de carga de peticiones entre recursos de servidores heterogéneos sin basarse en ninguna planificación específica o mecanismos de balanceo de carga. Finalmente, como los clientes no interactuan con los servidores directamente, es posible ajustar la asignación de recursos y la estrategia de balanceo de carga dinámicamente.

Predicción Online

Para mejorar el *middleware Grid* con capacidades de predicción online, hemos desarrollado un *componente de predicción online* que puede llamarse en cualquier momento durante la ejecución del sistema para predecir el rendimiento de la *Grid*, dada una estrategia de planificación representada por la función de asignación de threads.

La predicción se realiza mediante un modelo de rendimiento (online) generado y analizado en tiempo real. Este componente puede usarse para encontrar una estrategia de planificación óptima que cumpla los SLA del cliente bajo una serie de limitaciones de recursos. Por ejemplo, cuando un cliente envia una petición para iniciar una nueva sesión, el planificador puede desestimar la petición si no es capaz de encontrar una estrategia de planificación que satisfaga el SLA del cliente. El diseño de mecanismos inteligentes para el control de la calidad de servicio está fuera del alcance de este capítulo. En las siguientes secciones nos centraremos en el componente de predicción y la evaluación de

su efectividad, utilizándolo en un entorno real.

El componente de predicción está formado por dos subcomponentes: *generador del modelo* y *solucionador del modelo*. El generador del modelo construye de forma automática el modelo de rendimiento basado en las sesiones activas de los clientes y en los servidores *Grid* disponibles. El modelo se soluciona de manera analítica o a través de simulación. Podemos definir distintos tipos de modelos para implementar el componente. En esta capítulo estamos usando *QPNs* (*Queueing Petri Nets*), que ofrecen una gran potencia de modelado y expresividad que no tienen otros métodos tradicionales, como pueden ser redes de colas, redes de colas extendidas y Petri Nets estocásticas [Bause, 1993, Bause et al., 1997, Kounev and Buchmann, 2003].

En [Kounev, 2006], se muestra como los modelos *QPN* manejan con soltura componentes distribuidos y ofrecen un número importante de beneficios, como la precisión. La expresividad de los modelos hacen posible la modelización de los threads lógicos usados en nuestro mecanismo de planificación de manera precisa. Dependiendo del tamaño de los modelos se pueden usar distintos métodos para su análisis, desde soluciones analíticas en forma de producto [Bause and Buchholz, 1998] hasta técnicas optimizadas de simulación [Kounev and Buchmann, 2006].

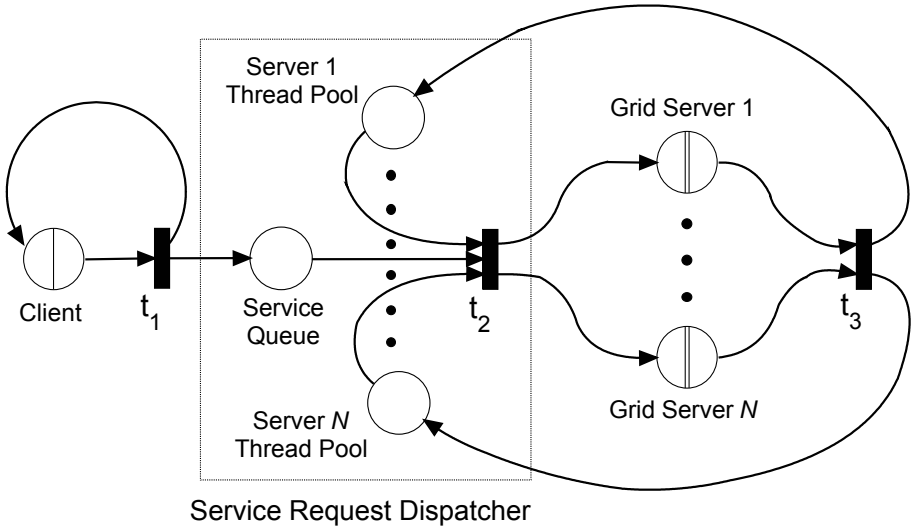


Figura 5.1: Modelo de alto nivel de *QPN* del entorno *Grid*.

La figura 5.1 muestra una representación jerárquica del modelo *QPN* de un conjunto de servidores *Grid* cuyo acceso se realiza a través de nuestro SRD. Los servidores *Grid* se modelan usando una representación de redes enlazadas *QPN* que representan subredes. El *Client* es una lugar que contiene una cola $G/G/\infty/IS$ que modela las llegadas de las

peticiones de servicio enviadas por los clients. Las peticiones de servicio están modeladas usando fichas de distinto color, donde cada color representa una sesión de un cliente. Para cada sesión activa siempre hay una ficha en el emplazamiento del *Client*.

Cuando la ficha abandona la cola *Client*, la transición t_1 se dispara, moviendo la ficha hacia el lugar *Service Queue* (representando la llegada de una petición de servicio) y depositando una nueva copia en la cola *Client*. Esta ficha nueva representa la siguiente petición de servicio, que se espera en la cola el tiempo entre llegadas especificado. Se puede usar una distribución de llegadas arbitraria. Para cada servidor *Grid*, el *SRD* tiene un emplazamiento *Server Thread Pool* que contiene las fichas, representando los threads lógicos en este servidor que estan asignados a las distintas sesiones (usando colores para distinguirlas entre ellas). Un servicio que llega se encola en la *Service Queue* y espera hasta que el thread de su sesión está disponible.

Cuando esto sucede, la petición se envía a la subred que representa el servidor *Grid* correspondiente. Después de que la petición sea procesada, el thread lógico se devuelve a la pool de donde se sacó. Encapsulando los detalles del servidor *Grid* en una red separada, podemos modelar diferentes servidores con distintos niveles de detalle según la complejidad de los servicios que ofrecen.

En cierto momento, el componente de predicción tiene en su poder las sesiones activas y los servidores disponibles. Se asume que cuando se añaden nuevos servidores al sistema, el modelo correspondiente a este nuevo servidor se añade al sistema. De esta manera, cuando se invoca al componente de predicción, éste es capaz de construir dinámicamente un modelo que represente el estado actual del sistema. El modelo se construye integrando los modelos de los servidores *Grid* en el modelo de alto nivel *QPN*. Esta construcción está automatizada y ocurre en tiempo real.

5.2.2. Estudio de la aplicación

En esta sección presentamos nuestro estudio de una instalación de *GT 4*, el cual ha estado ampliado para poder usar funcionalidades de predicción, como hemos descrito anteriormente. Hemos evaluado la calidad de nuestro mecanismo de predicción y hemos presentado algunos resultados experimentales para demostrar su efectividad. Nuestro entorno experimental es el representado en la figura 5.2, y consiste en dos servidores *Grid* heterogéneos: el primero es un Pentium Xeon a 2.4 GHz con 2 GB de memoria y 2 procesadores, y el segundo es un Pentium Xeon a 1.4 GHz con 4 GB de memoria y 4 procesadores. Los dos ejecutan un Globus Toolkit 4.0.3 (con los últimos parches) en una *JVM* de Sun 1.5.0_06. Los clientes *Grid* están emulados en una máquina separada con el mismo hardware que el primer servidor *Grid*. Las máquinas se comunican con una red Gigabit.

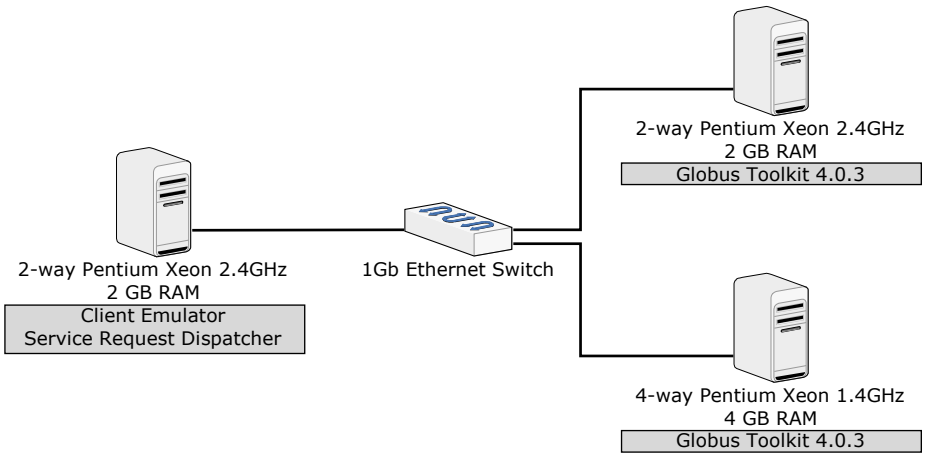


Figura 5.2: Diagrama de nuestro entorno experimental.

Caracterización de la carga

Para nuestros experimentos, usamos diferentes tipos de trabajo que ejecutan alguna función que requiere una cierta cantidad de tiempo de CPU. Algunos de los trabajos incluyen llamadas a proveedores de servicio externos que no son parte del entorno *Grid*.

Para construir los modelos de los dos servidores *Grid* hemos de caracterizar la carga en términos de *service demand* de los trabajos que ejecuta.

Hay muchas técnicas para determinar las necesidades de CPU de los servicios. La más exacta sería utilizar un profiler de Globus para medir los tiempos de CPU directamente. Podemos usar, por ejemplo, el BSC Monitoring Framework (*BSC-MF*) [Nou et al., 2007a], junto con la herramienta de análisis Paraver [Jost et al., 2003]. Otra propuesta que no requiere hacer profiling de Globus es estimar las necesidades de CPU midiendo la utilización de la CPU y los datos de *throughput* de los trabajos. Este método es muy general y no requiere ninguna herramienta externa. Para cada trabajo se ejecuta un experimento inyectando trabajos de cada tipo. Con la ley de la utilización [Denning and Buzen, 1978] podemos computar el tiempo de servicio medio D como la división de la utilización de la CPU (U) entre el *throughput* de trabajos (X). En algunos casos, ciertas técnicas pueden usarse para ayudar a estimar el tiempo de servicio sin necesidad de hacer ninguna medida en el sistema [Menascé and Gomaa, 2000]. Estas técnicas estan basadas en analizar la lógica de negocio que se ejecuta mirando el código del servicio.

El cuadro 5.1 muestra los tiempos de servicio de los trabajos analizados. Para cada tipo de trabajo, mostramos el tiempo de procesado interno, el tiempo esperando el proveedor

Cuadro 5.1: Tiempo de servicio estimado y overhead de procesado de Globus (seg).

Job	**A**	**B**	**C**	**D**	**E**	**F**	**G**
Tiempo de procesado interno	20,00	10,00	6,00	5,00	4,00	1,00	0,50
Tiempo de procesado externo	5,00	0,00	2,00	0,00	3,00	0,20	0,00
Tiempo de servicio del trabajo	21,00	11,00	6,89	5,84	4,79	1,93	1,54
Overhead de Globus	1,00	1,00	0,89	0,84	0,79	0,93	1,04

Cuadro 5.2: Tiempo de servicio de los servicios escogidos (seg).

Servicio	**Servicio 1 (C)**	**Servicio 2 (E)**	**Servicio 3 (D)**
Tiempo de servicio en el servidor de 2 CPU	6,89	4,79	5,84
Tiempo de servicio en el servidor de 4 CPU	7,72	5,68	6,49
Tiempo de servicio externo	2,00	3,00	0,00

externo y el total (medido) del tiempo de servicio y del overhead que introduce Globus. El overhead que introduce Globus es constante (1 segundo) en los 7 tipos de trabajo.

En el resto de los experimentos nos centramos en los trabajos centrales (C, D y E), que analizaremos con más detalle. Asumimos que estos trabajos se ofrecen como servicios separados por los servidores *Grid*. El cuadro 5.2 muestra los tiempos de servicio medidos de estos trabajos para los dos servidores que estamos testeando.

Modelos de los servidores Grid

Asumimos que cuando los servidores entran en la *Grid*, se registran en el componente de predicción. Cada servidor ofrece un modelo de su rendimiento en forma de *QPN*, que captura su comportamiento cuando procesa peticiones de servicio. Cuando el componente de predicción se invoca, se construye el modelo del entorno *Grid* dinámicamente, integrando las subredes en el modelo de alto nivel presentado en la sección 5.2.1 (ver Figura 5.1).

Los dos servidores usados en nuestro estudio están modelados usando una red *QPN*, como se muestra en la figura 5.3. Las peticiones de servicio que llegan al servidor *Grid* cir-

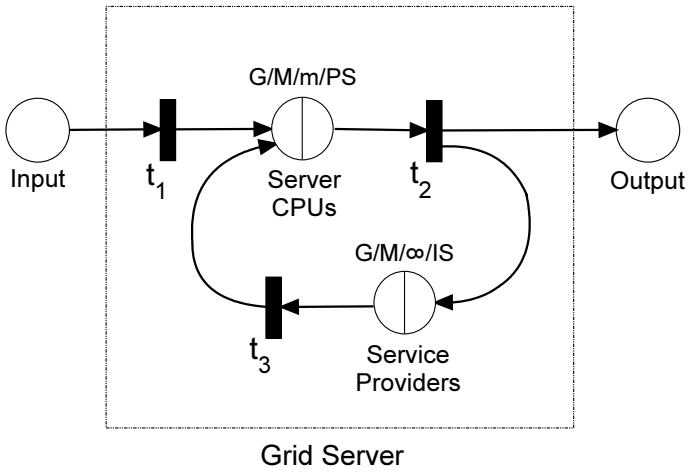


Figura 5.3: Modelo *QPN* de un servidor *Grid*.

culan entre la cola de *Server CPUs* y la cola de *Service Providers*, que modelan el tiempo usado por los procesadores del servidor y el tiempo esperando la respuesta del servicio externo, respectivamente. El emplazamiento *Server CPUs* contiene una cola $G/M/m/PS$, donde m es el número de CPUs, mientras que el emplazamiento *Server CPUs* contiene una cola $G/M/\infty/IS$.

Los tiempos de servicio de las peticiones que llegan a estas colas se seleccionan según las medidas mostradas en la tabla 5.2. Para simplificar, suponemos que los tiempos de servicio, las llegadas de las peticiones y el tiempo esperando el servicio externo están distribuidos exponencialmente. En el caso general esto es un requerimiento. Los pesos de las transiciones t_2 están seleccionadas de manera que *Service Providers* se visita una vez para los servicios 1 y 2, y no se visita para el servicio 3. El solucionador del modelo se implementa usando SimQPN, un motor de simulación para *QPN* optimizado [Kounev and Buchmann, 2006].

Antes de usar los modelos de los servidores deben validarse. Esto se realiza, normalmente, comparando las predicciones obtenidas con el modelo, con medidas obtenidas en el sistema real. Nuestros intentos iniciales para validar el modelo mostraron que las predicciones eran precisas en escenarios sin limitación en el número de peticiones concurrentes, y no tan precisas cuando limitamos la concurrencia (ver sección 5.2.1).

El cuadro 5.3 muestra cuatro escenarios que hemos considerado. Dado que para algunos escenarios el error fue superior al 15 %, los modelos no pasaban nuestro primer intento de validación. Para investigar el problema, analizamos el comportamiento interno de Globus cuando se procesaban peticiones de servicio. La figura 5.4 muestra las fases y el uso de CPU mientras se procesa una petición para el servicio 3(Trabajo D).

Servicios	**Nº de threads**	**Tiempo entre peticiones (seg)**	**Tiempo de respuesta (seg)**		**Error (%)**
			medido	**predecido**	
2	ilimitado	4	11,43	10,47±0,033	8,3 %
1—3	ilimitado	8 / 8	13,66 / 12,91	12,21±0,019 / 11,17±0,031	11 % / 13 %
3	5	2,5	10,93	8,14±0,030	**25 %**
1—3	2/2	8 / 8	18,15 / 9,79	15,58±0,23 / 7,8±0,05	14,1 % / **20,3 %**

Cuadro 5.3: Predicciones antes de la calibración.

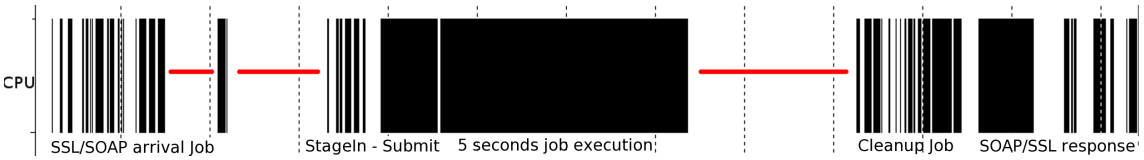


Figura 5.4: Vista paraver de la ejecución de un trabajo dentro de Globus.

Esta vista se obtiene usando una traza generada por *BSC-MF* [Nou et al., 2007a], procesada con Paraver [Jost et al., 2003]. Las zonas negras muestran periodos donde se usa una CPU, y las zonas marcadas con una línea roja horizontal corresponden a periodos donde todas las CPUs están sin hacer nada (en total cerca de 1 segundo). Este tiempo sin hacer nada transcurre entre las transiciones de estado de los trabajos. Hemos de notar que el servicio se ejecutó sin tener otros usuarios en la máquina, por lo que estos periodos no fueron causados por contención software o hardware, ni tampoco por E/S, ya que el uso de disco es muy bajo. Tener en cuenta estás esperas internas nos ayudará a averiguar por qué el modelo es menos preciso en el caso de concurrencia limitada. En este caso, una diferencia de 1 segundo, dado que hay trabajos esperando, provoca más impacto en el tiempo de respuesta que en el caso de concurrencia ilimitada, donde este tiempo se filtra.

Monitorizando Globus cargada con peticiones múltiples, vimos que estos periodos eran constantes y no estaban afectados por la intensidad o la variedad de la carga. Con estos resultados decidimos calibrar nuestro modelo introduciendo este periodo de inactividad durante la ejecución de un servicio. Por simplicidad, lo añadimos al tiempo esperando en *Service Providers*. Después de esta calibración, las diferencias entre las predicciones y las medidas desaparecieron.

Una propuesta alternativa a modelar el entorno a *SimQPN* en utilizar un sistema de simulación general como *OMNeT++* [Vargas, 2001], basado en simulación por paso de mensajes y utilizado en el capítulo 2.3, donde hemos modelado un servidor Tomcat [Nou et al., 2006b] con un control de admisión. La figura 5.5 compara la precisión ofrecida por *SimQPN* y *OMNeT++* cuando simulamos un modelo de nuestro entorno utilizando varios trabajos concurrentes. De igual manera, en la figura 5.6 podemos ver como el pérfil

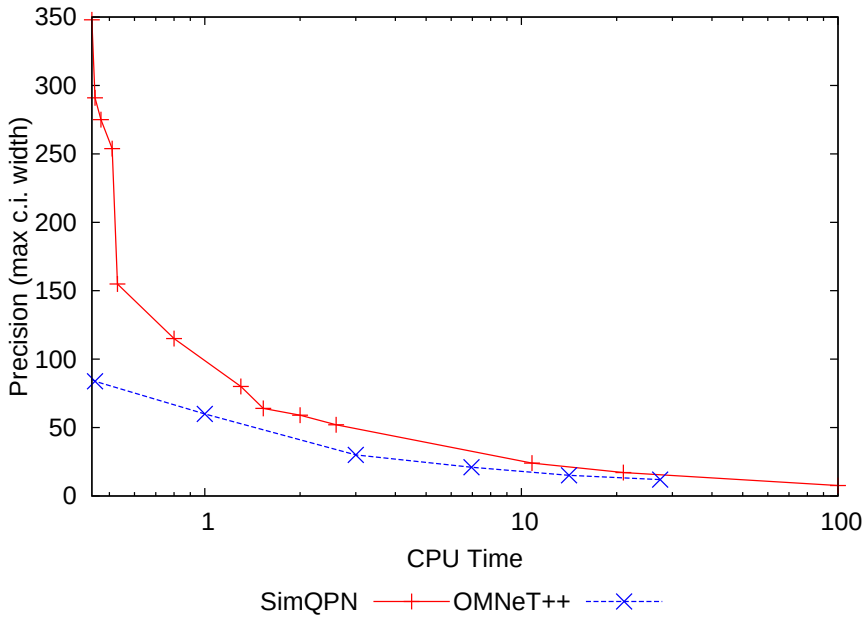


Figura 5.5: Precisión obtenida con SimQPN y con OMNeT++ dado un tiempo de simulación (seg).

de tiempo de respuesta predecido es similar al medido; el eje x representa un incremento en la concurrencia de las peticiones de servicios en el servidor.

La precisión se mide en términos de la amplitud máxima de un intervalo de confianza del 95 % para el tiempo de respuesta de los trabajos. Para tiempos de ejecución inferiores a 1 segundo, *SimQPN* nos ofrece un intervalo de confianza mayor que *OMNeT++*. A pesar de esto, las diferencias se hacen inapreciables con tiempos de ejecución superiores a 1 segundo. Además, mientras que los resultados de *OMNeT++* estaban limitados a los tiempos de respuesta, ya que no hemos programado otras métricas, en *SimQPN* los resultados incluían estimaciones del *throughput*, utilización, tamaños de cola, etc. Los modelos *QPN* tienen la ventaja de ser más fáciles de construir como hemos mostrado en la sección 5.2.1; aun así no conviene descartar *OMNeT++*, ya que permite flexibilidad, al poder simular características específicas o particulares del software que son complicadas de realizar con *QPN* u otros métodos.

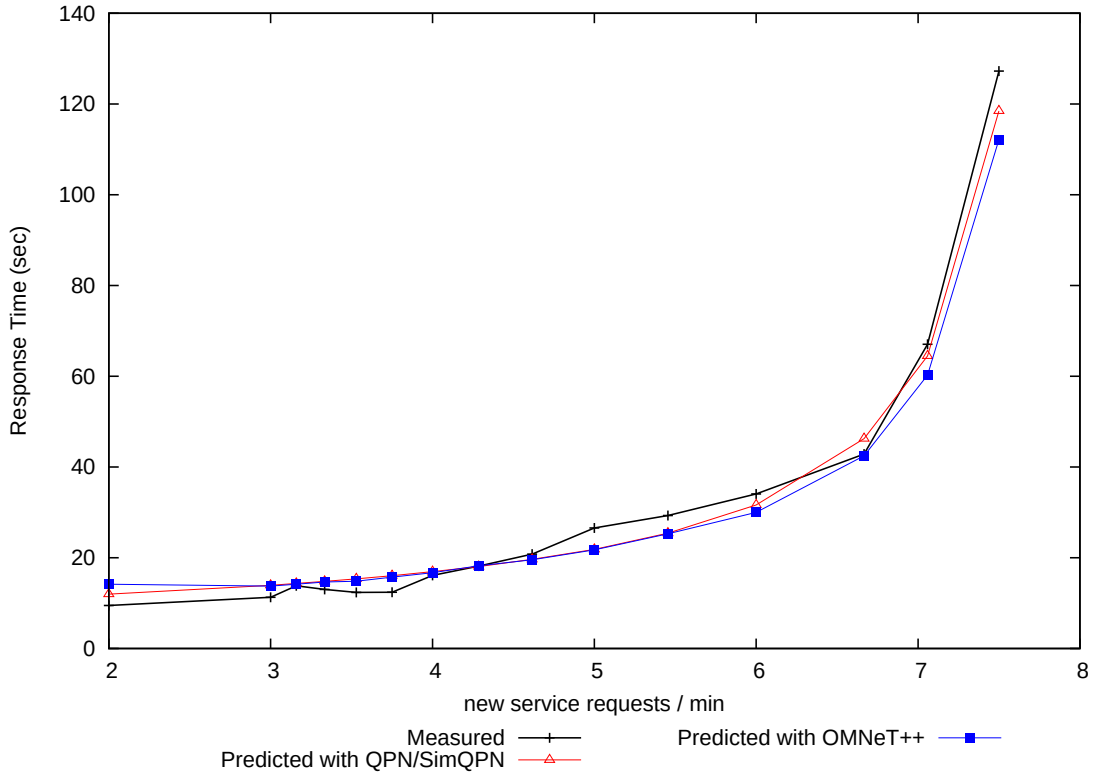


Figura 5.6: Tiempos de respuesta (seg) predecidos y medidos en servidor de 1 CPU.

Resultados experimentales

Ahora evaluaremos la calidad de nuestro mecanismo de predicción en el contexto del escenario descrito anteriormente.

Cuando se invoca el componente de predicción, se analiza utilizando *SimQPN*. *SimQPN* soluciona el modelo usando el método de *non-overlapping batch means*, usando un tamaño de bloque de 300 y simulando hasta que las alturas medias de los intervalos de confianza del tiempo de respuesta (95 %) bajan hasta 0,5 segundos.

Hemos usado el componente de predicción para predecir el rendimiento de la *Grid* bajo un número elevado de distintas cargas y escenarios, variando los tipos de sesiones, la estrategia de planificación y el número de servidores disponibles; el cuadro 5.4 presenta los resultados de un experimento donde la carga evoluciona a través de 5 fases (cada uno con una mezcla de sesiones distinta y con una duración entre 1.200 y 3.600 segundos).

Al inicio de cada fase, la estrategia de planificación se modifica dinámicamente y

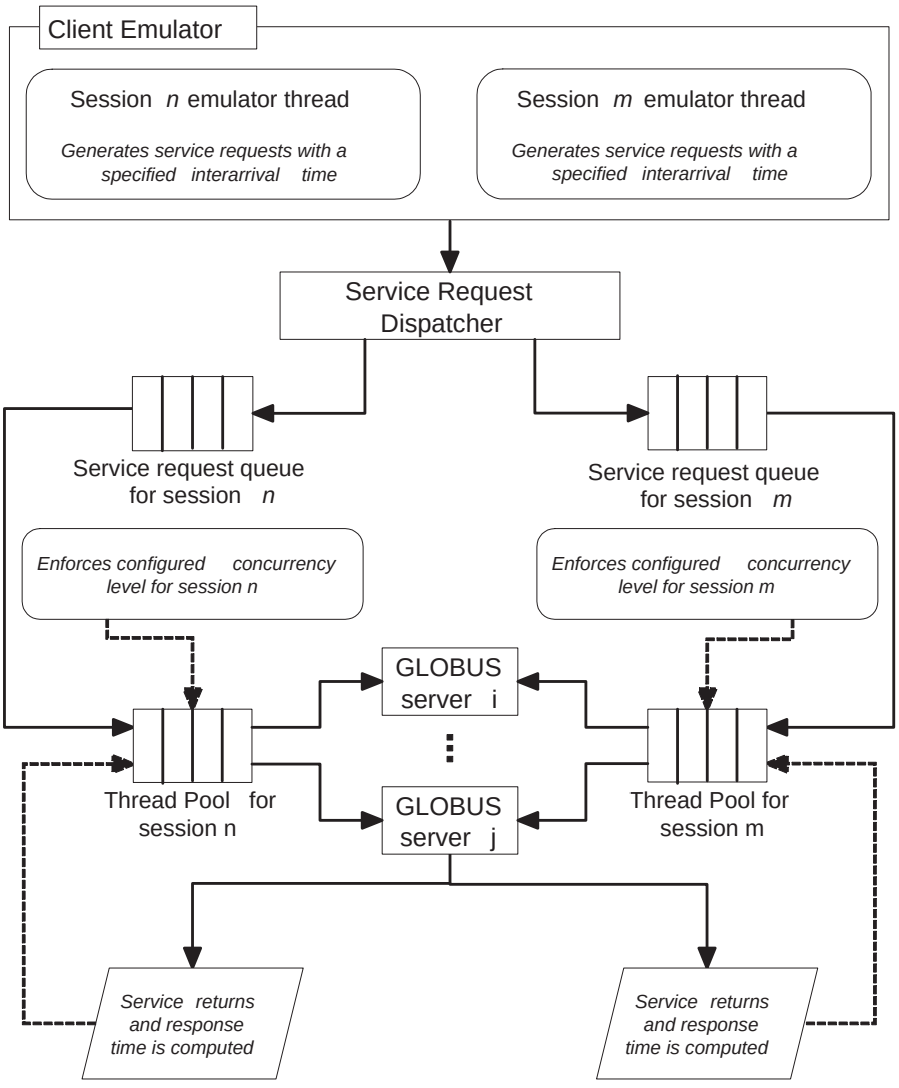


Figura 5.7: Diagrama del procesado de una petición de servicio.

Servicios	Nº de threads		T. entre peticiones	T. de respuesta		Error	Util. CPU	
	S1	S2		M	P		M	P
1	3	2	12,5	13,86	13,23±0,591	+0,63	0,66	0,65
1	5	3	13	14,57	13,41±0,622	+1,16		
2	2	4	11	11,36	10,63±0,432	+0,73		
2	1	2	12	11,07	10,49±0,452	+0,58		
2	5	4	15	12,49	10,99±0,453	+1,5		
3	1	5	13	9,37	8,36±0,331	+1,01		
3	1	2	16	9,41	8,53±0,363	+0,88		
3	4	4	16	10,17	9,79±0,520	+0,38		
1	1	10	12,5	11,61	11,13±0,597	+0,48	0,56	0,58
1	4	7	15	11,95	12,57±0,736	-0,62		
2	8	6	15	12,20	10,45±0,567	+1,75		
2	4	6	9	10,67	10,45±0,466	+0,22		
2	3	8	15,5	10,59	10,39±0,542	+0,2		
3	7	3	16	10,90	8,93±0,577	+1,97		
3	10	2	12	11,67	9,01±0,508	+2,66		
1	5	2	12,5	13,81	13,81±0,669	+0	0,64	0,66
1	2	1	13	13,76	14,19±0,750	-0,43		
2	1	4	11	10,4	10,2±0,532	+0,2		
2	1	5	12	10,3	10,18±0,507	+0,12		
2	4	1	15	13,45	11,80±0,479	+1,65		
3	1	1	13	9,61	9,7±0,504	-0,09		
3	3	2	16	10,9	9,82±0,627	+1,08		
3	2	2	16	9,51	9,57±0,617	-0,06		
1	1	1	14,5	12,36	14,28±0,443	-1,92	0,61	0,63
1	3	2	17	13,40	13,2±0,393	+0,2		
1	5	3	18	15,51	13,2±0,438	+2,31		
2	5	1	15	13,19	11,5±0,336	+1,69		
2	4	3	25	11,88	11,16±0,405	+0,72		
2	1	3	15	10,29	10,13±0,302	+0,16		
2	5	2	25,5	13,3	11,11±0,412	+2,19		
3	4	4	16	9,48	9,66±0,317	-0,18		
3	1	4	16	8,25	8,38±0,234	-0,13		
3	3	3	25	8,65	9,56±0,340	-0,91		
1	4	2	18,5	13,35	14,77±0,623	-1,42	0,64	0,65
1	4	5	15	12,05	14,56±0,618	-2,51		
1	4	4	16	12,58	14,50±0,673	-1,91		
2	1	2	15	10,67	10,78±0,460	-0,11		
2	5	1	17	11,86	12,59±0,534	-0,73		
2	2	2	19	11,09	11,51±0,492	-0,42		
2	4	2	15,5	11,53	11,99±0,528	-0,46		
3	2	5	16	9,42	10,1±0,475	-0,68		
3	1	4	19	8,98	9,14±0,413	-0,16		
3	3	5	23	8,92	10,57±0,535	-1,65		

Cuadro 5.4: Comparación de las predicciones obtenidas con las medidas recogidas del sistema real. Tiempos en segundos. S1,S2 = Servidor, M = Medido, P = Predecido.

el componente de predicción se ejecuta para predecir el rendimiento en tiempo real. El tiempo medio para predecir el rendimiento de sistema (incluyendo el tiempo de generación del modelo y el análisis) fue de 6,12 segundos, con un máximo de 12 segundos. Las predicciones se guardan y se comparan más tarde con las medidas en la ejecución real. El experimento fue repetido múltiples veces y mostro variaciones inapreciables en las medidas. El cuadro 5.4 muestra los resultados de la comparación. Para el tiempo de respuesta se usa un intervalo de confianza del 95 %. El error de la predicción fue sólo de un 7,9 % (con una desviación estándard de 6,08), no superando el 22,5 % durante las pruebas.

Hemos realizado un experimento similar para otras cargas y configuraciones. Los resultados han sido similares a los presentados aquí, lo que demuestra la efectividad de nuestro mecanismo de predicción. El overhead o carga que genera el componente es de menos de 60 segundos para escenarios con 40 servidores *Grid* y 80 sesiones concurrentes.

5.3. Gestor de Recursos con control de calidad utilizando simulación online

En esta sección mostraremos los resultados obtenidos en la utilización de un gestor de recursos inteligente usando simulación online, que tiene en cuenta cuestiones de calidad de servicio (*QoS*) y de *SLA* (*Service Level Agreement*).

Para validar nuestra propuesta, hemos implementado un prototipo de gestor de recursos con control de calidad de servicio y lo hemos instalado en un entorno *Grid* real basado en *Globus Toolkit* [Foster, 2005]. Hemos realizado una evaluación experimental extensiva comparando su comportamiento en dos configuraciones distintas: "con control de *QoS*" y "sin control de *QoS*". Los resultados demuestran la efectividad de nuestra propuesta y su aplicabilidad en la gestión de recursos con calidad de servicio en entornos *Grid*.

5.3.1. Prototipo

Los sistemas de planificación y de asignación usados a nivel global y a nivel local juegan un rol crítico en el rendimiento y disponibilidad de una aplicación *Grid*. Para prevenir la congestión de recursos y la no disponibilidad de los mismos, es esencial que se utilicen controles de admisión en los gestores de recursos locales. Además, para conseguir el rendimiento máximo, el *middleware Grid* debe ser lo suficientemente inteligente para planificar las tareas de tal manera que la carga sea balanceada entre todos los recursos

disponibles y que todos sean igualmente utilizados. Aun así, esto no es suficiente para garantizar que el rendimiento y la calidad de servicio sean satisfechos.

Para prevenir que la calidad de servicio no sea inadecuada, los gestores de curso deben ser capaces de predecir el rendimiento del sistema dada una asignación de recursos y una distribución de la carga. En esta sección presentaremos una metodologia para diseñar este tipo de gestores de recursos que tienen en cuenta la calidad de servicio como parte de un *middleware Grid*. La metodología se aplica a la capa de recursos de una arquitectura general *Grid* como la mostrada en la sección 3.2.

Arquitectura del gestor de recursos

La arquitecura del gestor de recursos que proponemos está compuesta por cuatro componentes principales: "*QoS Broker*", "*QoS Predictor*", el registro de clientes y el registro de servicios. La figura 5.8 muestra una visión de la arquitectura. El gestor de recursos es responsable de gestionar el acceso a un conjunto de servidores *Grid* ofreciendo algunos servicios *Grid*. Los servidores pueden ser heterogéneos tanto en hardware, software o en los servicios que ofrecen. Los servicios se pueden ofrecer en un número indefinido de servidores, dependiendo de la demanda. El gestor de recursos mantiene qué servidores están disponibles, y pone de acuerdo a los clientes y los servidores para asegurarse que los *SLAs* se cumplen. Para que un cliente sea capaz de usar un servicio debe enviar una petición de sesión (*session request*) al gestor de recursos. La petición de sesión específica el tipo de servicio y la frecuencia con la que enviaran peticiones de ese servicio (*service request*) como parte de la sesión. Además, se necesita conocer el tiempo medio de respuesta (*SLA*) que pedimos. El gestor de recursos intenta encontrar una distribución de las peticiones que le llegan entre los servidores disponibles que pueden ofrecer la calidad de servicio necesaria. Si no fuera posible, la petición de sesión se cancela o se envía una oferta con un *throughput* menor o un mayor tiempo de respuesta. La figura 5.9 muestra de manera más detallada la vista de la arquitectura del gestor de recursos y sus componentes internos. A continuación analizaremos cada componente por separado.

El registro de servicios tiene constancia de los servidores *Grid* y los servicios que ofrecen. Antes de que pueda usarse un servidor *Grid*, debe registrarse con un gestor de recursos, ofreciendo información acerca de los servicios que ofrece, sus necesidades de recursos y la capacidad del servidor que se ofrecerá al *Grid*. Para una máxima interoperatividad, se espera que los mecanismos estándars de servicios Web, como *WSDL* [Chinnici et al., 2006], sean utilizados para describir los servicios. Adicionalmente, el servidor *Grid* debe ofrecer el modelo que capture el comportamiento del servicio y el uso de recursos. Dependiendo del tipo de servicio, el modelo puede variar en complejidad, pasando de una simple especificación del tiempo de servicio, a un modelo detallado, capturando el flujo

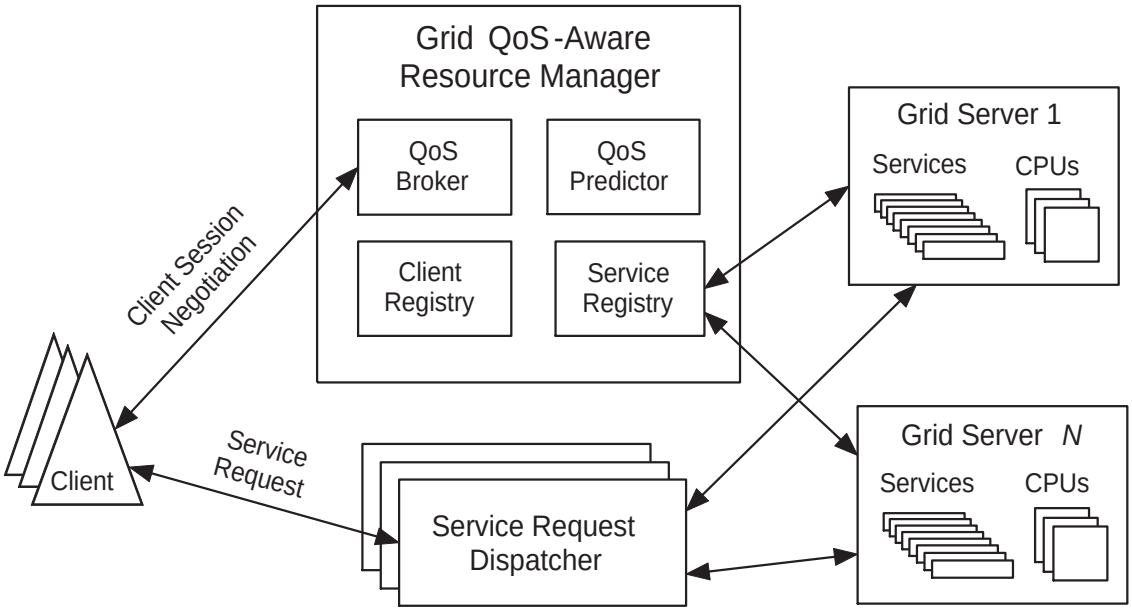


Figura 5.8: Arquitectura del gestor de recursos.

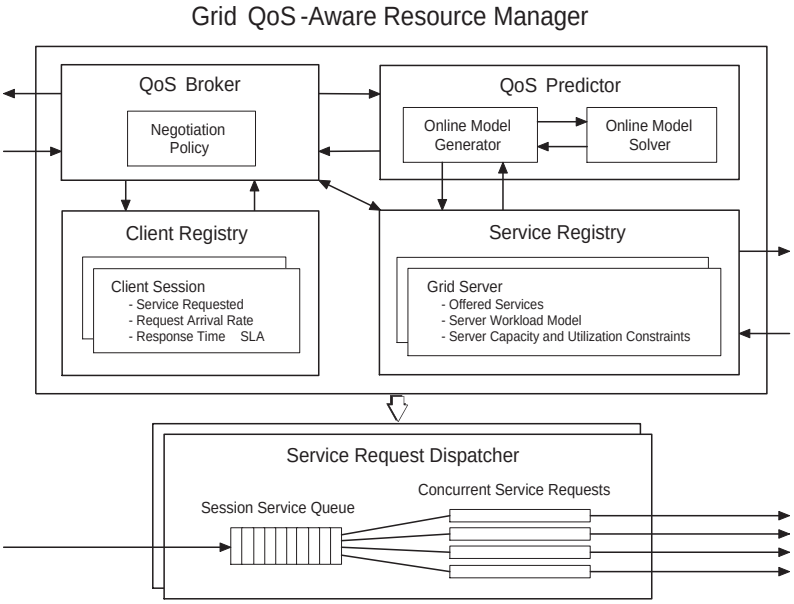


Figura 5.9: Vista en detalle de la arquitectura del gestor de recursos.

de control durante su ejecución. El *Job Submission Description Language* (*JSDL*) [An-

jomshoaa et al., 2005] podría usarse para definir los trabajos de una manera estándar. Finalmente, el administrador del servidor puede imponer una limitación en el uso de CPU del *middleware* para evitar sobrecargas. Algunos servidores *Grid* pueden mostrar un comportamiente inestable cuando su utilización es muy elevada [Nou et al., 2007c]. Para cada servidor, el registro mantiene la utilización máxima permitida.

El broker (*QoS broker*) recibe, las peticiones de sesión de los clientes, asigna los recursos y negocia el *SLA*. Cuando un cliente envia una petición de sesión, el broker intenta encontrar una distribución óptima de la carga entre todos los servidores, que disponibles ofertan el servicio, de manera que satisfagan los requerimientos de calidad de servicio. Se asume que para cada sesión, un número dado de *threads* (0 hasta ilimitado) se asignan en cada servidor, ofreciendo el respectivo servicio. Las peticiones son balanceadas entre los servidores según su disponibilidad. Los threads nos sirven como mecanismo para limitar la concurrencia de las peticiones en cada servidor, de modo que podemos utilizar distintos esquemas de balanceo de carga. El broker intenta distribuir la carga uniformemente entre todos los servidores disponibles para maximizar la eficiencia. Para hacerlo, considera diferentes configuraciones respecto la asignación de threads, y para cada uno de ellos usa el predictor (*QoS predictor*) para predecir el rendimiento del sistema. El objetivo es encontrar la utilización máxima del servidor *Grid*. Si no podemos encontrar una configuración adecuada, el broker puede descartar la petición de sesión o enviar una contraoferta al cliente. En cada momento el *registro de clientes* mantiene las sesiones activas de cada cliente. Para cada sesión tenemos información de la petición de servicio, el tiempo de llegada de las peticiones y el tiempo de respuesta que queremos (*SLA*). Si se acepta una petición de sesión, el gestor de recursos prepara un *SRD* (*Service Request Dispatcher*) para la nueva sesión, que es un componente autónomo responsable de planificar la llegada de las peticiones de servicio (como parte de una sesión) para que sean procesadas por los servidores *Grid*. Se asegura, además, que el número de peticiones concurrentes no superen el número especificado de threads para la sesión en el correspondiente servidor.

Los *SRD* no necesitan ejecutarse en la misma máquina que el gestor de recursos y puede distribuirse si se necesita. Para resumir, los *SRD* nos sirven de balanceadores de carga (ligeros) que garantizan la distribución de la carga seleccionada en fases anteriores.

Los *service request dispatchers* juegan un papel esencial en nuestra plataforma, ya que consiguen separar los clientes *Grid* de los servidores *Grid*. Esta separación nos ofrece importantes ventajas, relevantes en las aplicaciones *Grid* modernas. En primer lugar, nos permite introducir un balanceo de carga fino en el nivel de petición de servicio, y no de sesión. Segundo, hacen posible balancear las peticiones a través de recursos heterogéneos sin basarse en ningún planificador específico o mecanismo de balanceo de carga. Finalmente, como los clientes no interactuan con los servidores directamente, es posible ajustar la asignación de recursos y la estrategia de balanceo dinámicamente, lo que lo hace

especialmente atractivo para conseguir un entorno autogestionable y autoadaptable.

QoS Predictor

Hemos mostrado el predictor anteriormente en la sección 5.2.1. En este prototipo usamos ese mismo modelo. Adicionalmente, ampliaremos el mecanismo de planificación mostrado anteriormente por el siguiente:

Podemos representar el número de procesadores (CPUs) de un servidor $s \in S$ como $P(s)$, y la utilización máxima como $\overline{U}(s)$. Se asume que para cada sesión de un cliente, un número de *threads* (desde 0 hasta ilimitado) se asigna en cada servidor que ofrece el servicio. El objetivo del algoritmo de asignación de recursos es encontrar una configuración, en términos de asignación de *threads*, que cumpla el *SLA* de las sesiones activas y las limitaciones de utilización de los servidores. Una configuración se puede representar por una función $T \in [C \times S \longrightarrow \mathbb{N}_0 \cup \{\infty\}]$ (*función de asignación de threads*). De modo que una función o cantidad elevada a T significará que depende de la función de asignación de *threads*, por ejemplo $X^T(c)$.

Como hemos comentado anteriormente, el *QoS Broker* examina un conjunto de posibles configuraciones usando el predictor para determinar si cumplen los requerimientos. Para cada configuración el predictor coge como entrada la función de asignación de *threads* T y nos ofrece las siguientes métricas (predecidas) como salida:

$X^T(c)$ para $c \in C$ es el número total de peticiones de servicio completadas para la sesión del cliente c por unnidad de tiempo (Throughput total),

$U^T(s)$ para $s \in S$ es la utilización media del servidor s,

$R^T(c)$ para $c \in C$ es el tiempo de respuesta medio de las peticiones de servicio que llegan de la sesión del cliente c.

Definimos los siguientes predicados:

$P_X^T(c)$ para $c \in C$ definido como $(X^T(c) = c[\lambda])$

$P_R^T(c)$ para $c \in C$ definido como $(R^T(c) <= c[\rho])$

$P_U^T(s)$ para $s \in S$ definido como $(U^T(s) <= \overline{U}(s))$

Para que una configuración representada por T sea aceptable, las siguientes condiciones deben cumplirse: $(\forall c \in C : P_X^T(c) \wedge P_R^T(c)) \wedge (\forall s \in S : P_U^T(s))$. Definimos las siguientes funciones:

$A^T(s) \stackrel{def}{=} (\overline{U}(s) - U^T(s))P(s)$ es la cantidad de tiempo de CPU que no se utiliza dada una determinada configuración teniendo en cuenta la utilización máxima permitida del servidor.

$I^T(v, \epsilon) \stackrel{def}{=} \{s \in S : (v \in F(s)) \wedge (A^T(s) \geq \epsilon)\}$ es el conjunto de servidores que ofrecen el servicio v y que tienen un ϵ de tiempo de CPU sin usar. Ahora presentamos un algoritmo heurístico simple de asignación de recursos utilizando un pseudocódigo matemático. Está fuera del alcance de esta tesis presentar un completo análisis de las posibles heurísticas y de su eficiencia. Sea $\tilde{c} = (v, \lambda, \rho)$ una nueva petición de sesión de un cliente. El algoritmo sería el siguiente:

```
1  C := C ∪ {c̃}
2  for each s ∈ I^T(v, ε) do T(c̃, s) := ∞
3  if (∃ c ∈ C : ¬P_X^T(c)) then reject c̃
4  while (∃ ŝ ∈ S : ¬P_U^T(ŝ)) do
5  begin
6     T(c̃, ŝ) := 1
7     while P_U^T(ŝ) do T(c̃, ŝ) := T(c̃, ŝ) + 1
8     T(c̃, ŝ) := T(c̃, ŝ) − 1
9  end
10 if (∃ c ∈ C \ {c̃} : ¬P_X^T(c) ∨ ¬P_R^T(c)) then reject c̃
11 if (¬P_X^T(c̃) ∨ ¬P_R^T(c̃)) then
12    send counter offer o = (v, X^T(c̃), R^T(c̃))
13 else accept c̃
```

El algoritmo añade la nueva sesión a la lista de sesiones activas y asigna un número ilimitado de threads en cada servidor que tiene CPU disponible. Si esto conduce a que el sistema no puede mantener el *throughput* de la sesion activa, podemos descartarla. En otra situación, comprobamos que los servidores no superan la utilización requerida. Para cada uno de estos servidores, el número de threads asignados a esta nueva sesión es igual al máximo número que no resulta en la violación del *SLA* de la utilización. Llegados a este punto, comprobamos si estamos violando algún valor de las sesiones que ya estaban en el sistema, descartando la nueva sesión si fuera el caso. Cuando todo esto se ha cumplido,

sólo nos queda comprobar que el *SLA* de la nueva sesión no se ha violado; si esto sucediera podemos enviar una contraoferta. Si todos los requerimientos se cumplen, la sesión se acepta. En general, cuantos más recursos pida una petición, tendrá más posibilidades de no ser aceptada si el sistema está sobrecargado. Un *workload* largo necesitará más recursos y por lo tanto generará más interferencias con las sesiones que se están ejecutando (rompiendo su *SLA*), provocando que las probabilidades de descarte de la sesión sean más elevadas.

Dado que por cada servidor sobrecargado, los threads se asignan uno por uno hasta que se llega a la máxima utilización, en el caso peor, el número de configuraciones consideradas tienen un límite superior, igual al número de procesadores disponibles. El algoritmo presentado aquí puede mejorarse de muchas formas; por ejemplo, agregar sesiones y ejecutar los mismos servicios, recogiendo los recursos del servidor de abajo a arriba en vez de arriba a abajo, paralelizar la simulación para utilizar procesadores *multi-core*, reutilizar configuraciones antiguas o simular de forma proactiva. Las posibles optimizaciones y el coste del *QoS Broker* se comentan con más detalle en la sección 5.4.

5.3.2. Evaluación

Nuestro entorno es el mismo que el presentado en las secciones anteriores, usando los mismos tres servicios seleccionados.

Análisis de los resultados

Presentamos ahora los resultados de un experimento en el cual se hicieron 16 peticiones de sesión al gestor de recursos. Este experimento se ejecuta hasta que todas las sesiones aceptadas finalizan. La duración de la sesión, en terminos de número de peticiones de servicio enviadas antes de cerrar una sesión, varían entre 20 y 120 peticiones, con una media de 65. El tiempo de respuesta que se requiere (*SLA*) varía entre 16 y 30 segundos. Comparamos el comportamiento del sistema en dos configuraciones, con *QoS Control* y *sin QoS Control*. En el primero, el gestor de recursos aplica control de admisión usando nuestro framework para asegurarse que los *SLA* se cumplen. En el segundo, el gestor simplemente hace balanceo de carga de las peticiones hacia los dos servidores sin considerar las peticiones de calidad de servicio. Para ambos servidores asumimos que hay una utilización máxima del 90 %. El experimento fue repetido 10 veces por cada una de las configuraciones para evaluar la variabilidad de los datos obtenidos.

Las figuras 5.10 y 5.11 muestran la utilización obtenida de los servidores mientras se ejecutaba el experimento en las dos configuraciones. Podemos ver los puntos donde

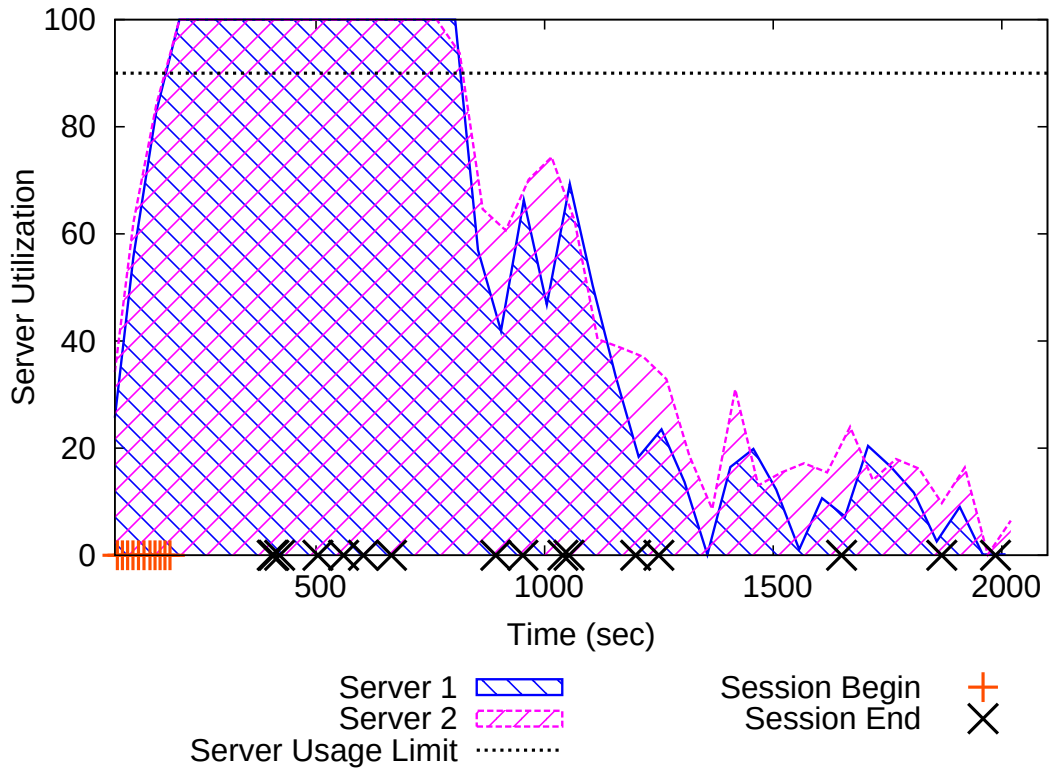


Figura 5.10: Utilización del servidor sin *QoS Control*.

se inician y finalizan las sesiones, marcados en el eje x. Como podemos ver, sin *QoS Control* ambos servidores están sobrecargados durante la primera mitad del experimento, sobrepasando su utilización máxima. Por otro lado, si activamos el *QoS Control*, algunas peticiones de sesión son descartadas y la utilización del servidor no sobrepasa el valor establecido del 90 %. La utilización no llega al 90 % por la naturaleza de las sesiones, ya que no existen sesiones suficientemente pequeñas (tanto de duración en tiempo de sesión como de tiempo de servicio) que puedan llenar el espacio hasta el 90 % en este caso. Además, los recursos del servidor para las sesiones aceptadas utilizan de igual manera los dos servidores *Grid*, de modo que en media estan equilibrados. El cuadro 5.5 compara el *throughput* conseguido de las sesiones y el tiempo de respuesta medio en los casos con y sin *QoS Control*. El *throughput* conseguido por cada sesión en las dos configuraciones puede observarse en la figura 5.12 (con un intervalo de confianza del 95 %)[2]. En el caso de sin *QoS Control*, el *throughput* esperado de la sesión 3 y las sesiones 11 a 15 no se alcanza, ya que los servidores están sobrecargados. Cuando se activa el *QoS Control*, todas

[2]Notar que en el caso de *QoS Control* sólo se muestra para las sesiones que se han aceptado

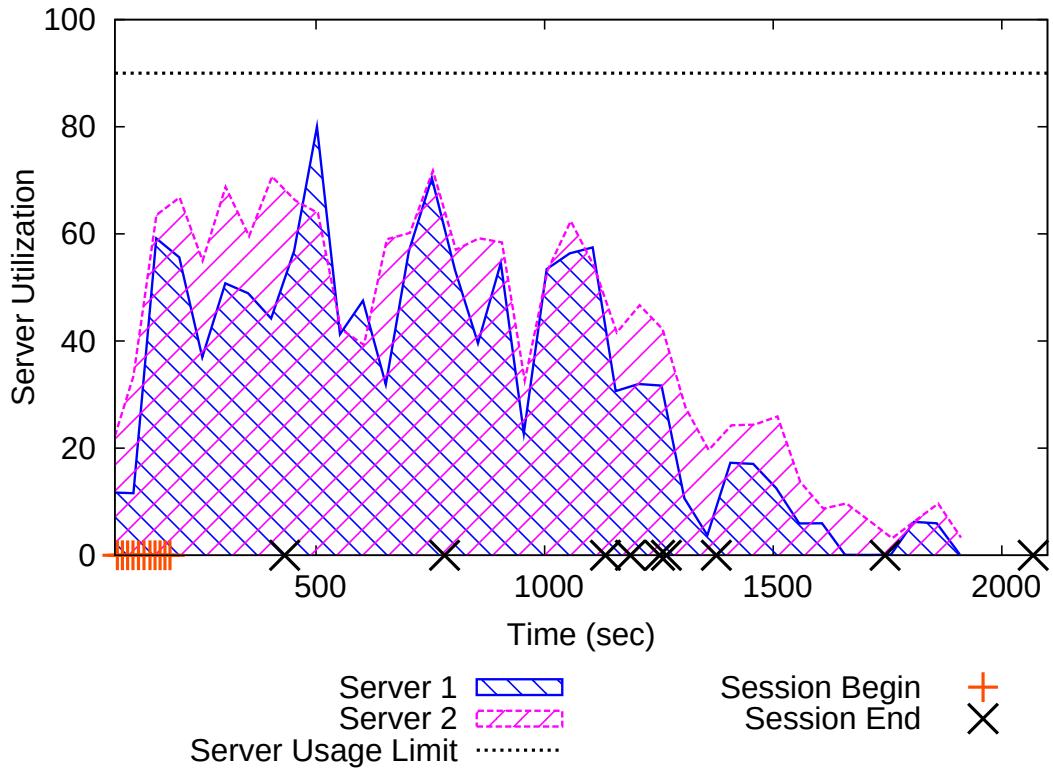


Figura 5.11: Utilización del servidor con QoS Control.

las sesiones aceptadas consiguen su *throughput*. Se ha de ver que el *throughput* esperado es una media sobre un largo periodo de tiempo, y que dada la limitación en la duración de la sesión, el *throughput* es muchas veces inferior o superior que la media esperada.

La figura 5.13 muestra el tiempo de respuesta medio de las sesiones con *QoS Control* y sin *QoS Control* (95 % de intervalo de confianza). Como hemos visto con los resultados, cuando se activa el *QoS Control*, el tiempo de respuesta es muy estable y los *SLAs* se cumplen. El sistema muestra un comportamiento muy estable en cada una de las iteraciones que hemos ejecutado, como podemos ver con los intervalos de confianza. En el otro lado, sin *QoS Control*, dado que los servidores están sobrecargados, el sistema muestra un tiempo de respuesta muy variable y los *SLA* del cliente no se cumplen. Los intervalos de confianza son más amplios en este caso.

Un objetivo importante de nuestro gestor de recursos es minimizar la sobrecarga al evaluar configuraciones alternativas para decidir si aceptar o descartar una nueva petición de sesión. El tiempo requerido para llegar a una decisión se estima entre $10{,}82 \pm 0{,}14$

Cuadro 5.5: Throughput de sesiones y tiempo de respuesta (i.c. del 95 %) predecido y medido con y sin QoS Control. P=Predecido, M= Medido. Tiempos en segundos

Sesion			Throughput			T. de respuesta		
ID	**Nº Servicio**	**Nº de peticiones**	**P**	**M con QoS Control**	**M sin QoS Control**	**SLA**	**M con QoS Control**	**M sin QoS Control**
0	1	120	0,0952	0,0942 ± 0,0062	0,0970 ± 0,0035	18,00	11,5 ± 0,06311	43,08 ± 7,22101
1	2	100	0,0800	0,0796 ± 0,0045	0,0818 ± 0,0050	16,00	10,28 ± 0,08231	76,82 ± 10,8527
2	3	120	0,0606	0,0586 ± 0,0018	0,0621 ± 0,0038	16,00	8,31 ± 0,09357	31,05 ± 5,62380
3	1	30	0,0606	0,0607 ± 0,0054	0,0526 ± 0,0038	16,00	11,64 ± 0,18346	69,58 ± 11,21426
4	2	120	0,1176	Descartado	0,1137 ± 0,0082	17,00	Descartado	36,37 ± 7,06515
5	3	100	0,0541	0,0557 ± 0,0027	0,0547 ± 0,0024	16,00	8,38 ± 0,04617	29,15 ± 4,04623
6	1	70	0,0541	0,0545 ± 0,0034	0,0576 ± 0,0019	18,00	11,51 ± 0,11032	40,54 ± 4,99668
7	2	80	0,0541	0,0597 ± 0,0034	0,0560 ± 0,0024	16,00	10,31 ± 0,08630	27,73 ± 4,23221
8	3	70	0,0541	0,0517 ± 0,0037	0,0573 ± 0,0050	16,00	8,54 ± 0,09493	43,63 ± 8,66235
9	1	60	0,0690	0,0768 ± 0,0010	0,0689 ± 0,0064	18,00	12,28 ± 0,0010	57,86 ± 7,92183
10	2	50	0,0870	Descartado	0,0832 ± 0,0066	15,00	Descartado	55,8 ± 10,46297
11	3	40	0,1176	Descartado	0,0864 ± 0,0082	18,00	Descartado	78,06 ± 11,34140
12	2	20	0,0500	Descartado	0,0437 ± 0,0032	30,00	Descartado	53,09 ± 10,15639
13	3	20	0,0556	Descartado	0,0454 ± 0,0038	30,00	Descartado	51,38 ± 21,02672
14	2	20	0,0526	Descartado	0,0417 ± 0,0041	30,00	Descartado	66,62 ± 13,61111
15	3	20	0,0556	Descartado	0,0484 ± 0,0035	30,00	Descartado	63,15 ± 13,33922

segundos (con un intervalo de confianza de un 95 %). Cuanto más detallado sea el modelo, mayor será la carga del gestor.

El experimento mostrado anteriormente se repite con un número distinto de configuraciones, variando la carga, el tamaño de la sesión, la utilización del servidor, etc. Los resultados fueron de similar calidad a los que se han presentado antes. La figura 5.14 muestra los tiempos de respuesta obtenidos en un experimento más largo, donde 99 sesiones se ejecutan sobre un periodo de 2 horas. La duración media de la sesión es de 18 minutos, en los cuáles se envían 92 peticiones de servicio media. En este experimento, cuando lo ejecutamos sin *QoS Control*, el sistema se configura para descartar las peticiones de sesiones durante los periodos en que los servidores están saturados. Mientras esto

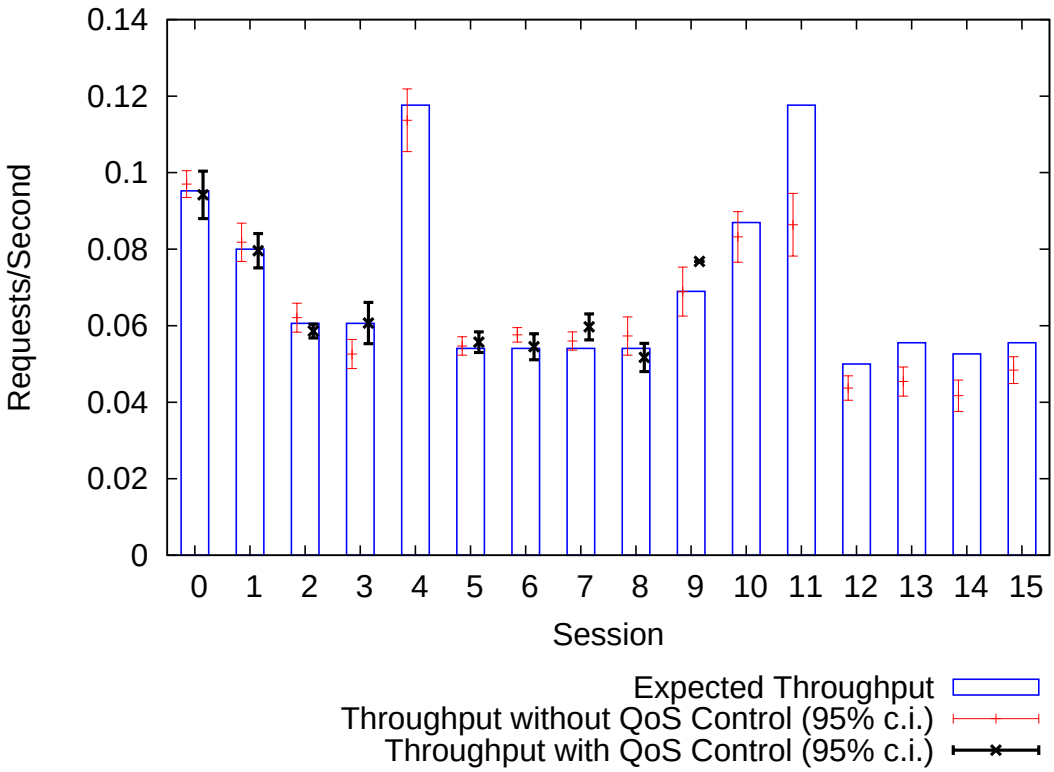


Figura 5.12: Throughput obtenido con QoS Control y sin él.

mejora el tiempo medio de respuesta de las sesiones aceptadas, el tiempo de respuesta es aún demasiado elevado, con lo que el *SLA* sigue siendo violado. Por otro lado, con *QoS Control* los tiempos de respuesta de las sesiones aceptadas fueron menores y los *SLAs* no se rompieron.

5.3.3. Ampliación

En esta sección mostraremos nuevas funciones que podemos realizar con el framework que proponemos.

Una de las características que añadimos es la caracterización de forma autónoma de la carga, de modo que ya no es necesario conocer qué tiempo de servicio tienen los distintos servicios que ofrecemos. Esto reduce la precisión, pero veremos en la evaluación como es perfectamente usable y supone importantes ventajas respecto a la alternativa sin predicción y además de posibilizar la utilización del sistema cuando no tenemos un modelo de

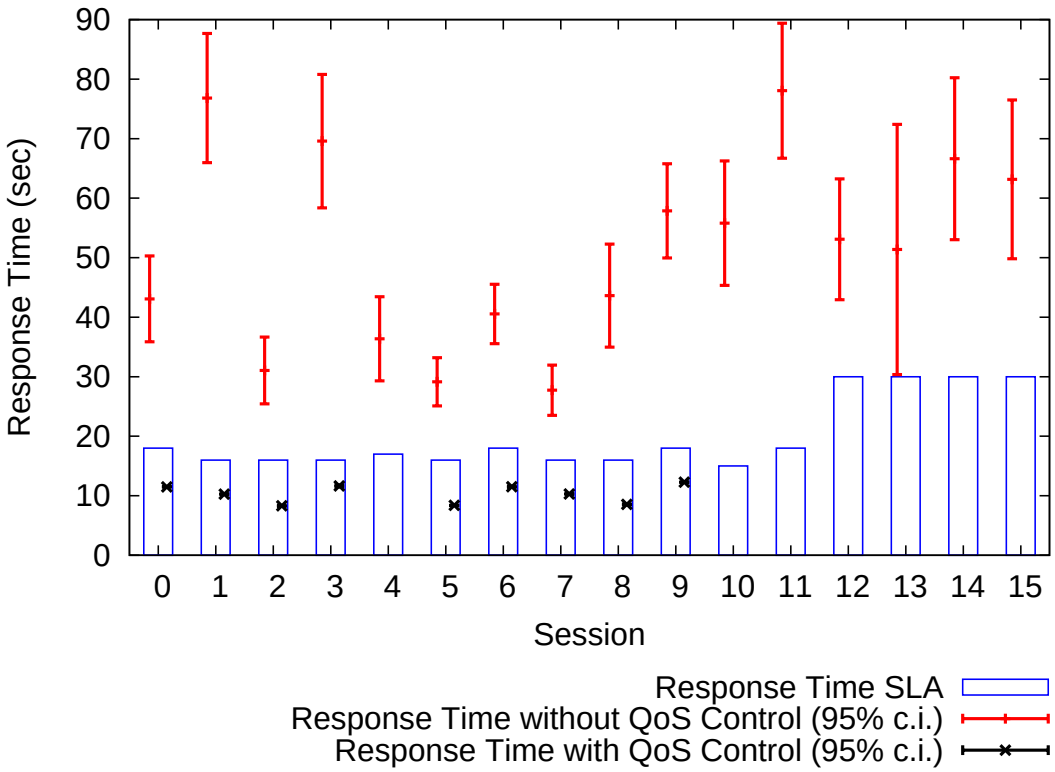


Figura 5.13: Tiempo de respuesta obtenido con QoS Control y sin él.

carga disponible. Esta caracterización de la carga se consigue utilizando monitorización.

También añadimos pruebas con dinamismo en el entorno; en nuestro caso permitimos que se añadan servidores *Grid* bajo demanda, así como la reconfiguración del sistema después de un fallo en el sistema. Para hacerlo montamos nuestro entorno en un entorno virtualizado para incrementar nuestro número de servidores hasta 9; finalmente, cambiamos el motor y el método de simulación para usar *OMNeT++*.

Caracterización de la carga en linea

Cuando un nuevo servidor *Grid* se registra en el gestor de recursos, un modelo de la carga que captura el comportamiento de los servicios y de sus necesidades en cuanto a recursos es añadido al sistema. En esta sección presentamos un método para generar estos modelos en linea utilizando información de monitorización. Asumimos que los servicios usan la *Grid* para ejecutar lógica de negocio que requiere usar un determinado tiempo

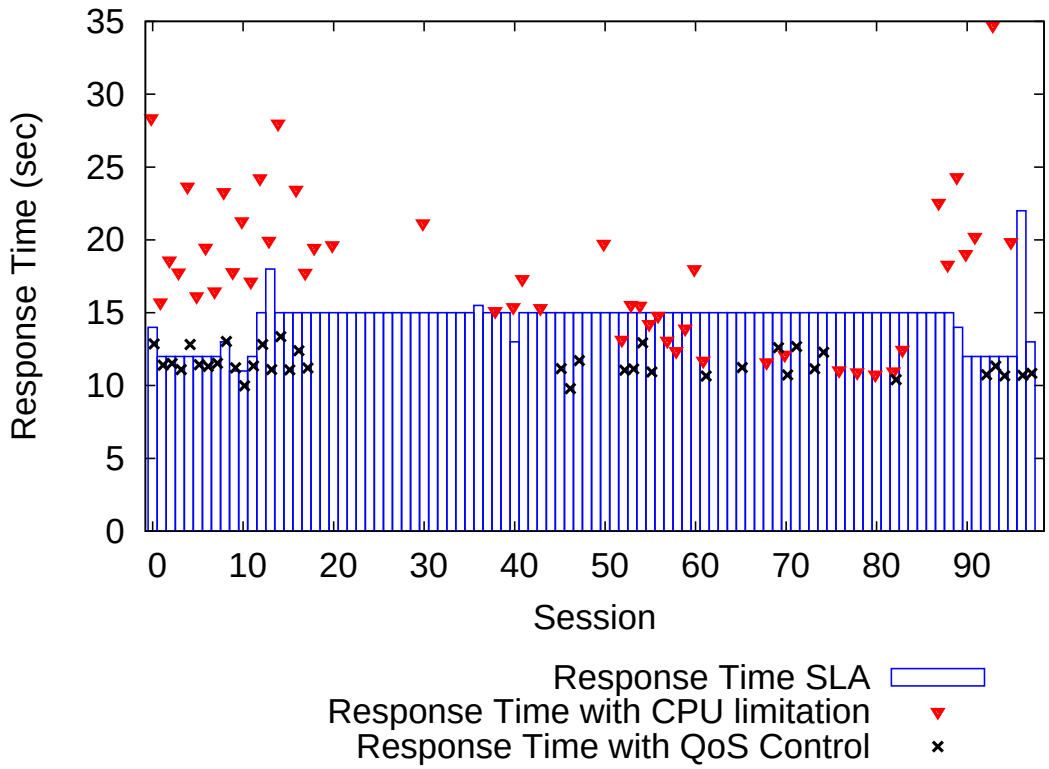


Figura 5.14: Tiempo de respuesta obtenido con QoS Control y sin QoS Control.

de CPU. El método que proponemos no es aplicable a servicios con paralelismo interno. Aun así, la lógica de negocio puede incluir llamadas a proveedores externos. Resumiendo, para generar el modelo de carga, necesitamos estimar el tiempo de servicio (*CPU service demands*) de los servicios (tiempo total que utilizan la CPU del servidor) y el tiempo que han estado esperando al servidor externo.

Hemos desarrollado un algoritmo para estimar este tiempo iterativamente durante la ejecución del sistema. Asumimos que los servicios se añaden y se suprimen dinámicamente y no tenemos información de su comportamiento y de sus necesidades de recursos. Los tiempos de servicio son estimados para cada sesión basándonos en información de monitorización. Al principio de la sesión, el tiempo de servicio de cada servidor es igual al tiempo de respuesta deseado (*SLA*)($\tilde{c}[\rho]$ usando la notación de las anteriores secciones). Esto sirve como un límite superior (conservador) de la demanda real. Cuando esta petición es procesada en un servidor *Grid*, el tiempo de respuesta obtenido se compara con la estimación actual del tiempo de servicio y si es inferior se actualiza. Tal como progresa la sesión, el tiempo de servicio se va actualizando iterativamente al menor tiempo

de respuesta observado en el correspondiente servidor. Se espera que durante periodos de baja intensidad, el tiempo de respuesta observado sea similar al tiempo de servicio. Esta propuesta funciona si el tiempo de espera de las peticiones exteriores es insignificante comparado con el tiempo que está ejecutando lógica de negocio. Para irnos al caso más general, podemos monitorizar el uso de la CPU del servidor durante la ejecución del servicio para separar el tiempo de respuesta medido entre el tiempo de CPU y el tiempo esperando llamadas externas. Esto nos permite estimar el tiempo de espera del servicio externo. Más formalmente, el algoritmo es el siguiente:

```
1 Upon arrival of new session request ĉ do
2  for each s ∈ S : (ĉ[v] ∈ F(s)) do
3  begin
4       Γ[ĉ, s]  :=  ĉ[ρ]
5       Ψ[ĉ, s]  :=  0
6       Θ[ĉ, s]  :=  ĉ[ρ]
7  end
8
9 Upon completion of request x of session ĉ at server ŝ do
10 if (M_R(x) < Θ[ĉ, ŝ]) then
11 begin
12      Θ[ĉ, ŝ]  :=  M_R(x)
13      Γ[ĉ, ŝ]  :=  M_U(ŝ)Θ[ĉ, ŝ]
14      Ψ[ĉ, ŝ]  :=  Θ[ĉ, ŝ]  −  Γ[ĉ, ŝ]
15 end
```

donde

$\Gamma[c, s]$ para $c \in C$ y $s \in S$ muestra el tiempo de servicio (total) de las peticiones de la sesión c al servidor s.

$\Psi[c, s]$ para $c \in C$ y $s \in S$ muestra el tiempo total consumido esperando un servicio externo cuando se sirve una petición de la sesión c en el servidor s.

$\Theta[c, s]$ para $c \in C$ y $s \in S$ muestra el tiempo de respuesta mínimo observado de una petición de sesión c en el servidor s.

$M_U(s)$ para $s \in S$ muestra la utilización de CPU del servidor s durante la ejecución de un servicio.

$M_R(x)$, donde x es una petición de servicio, muestra el tiempo de respuesta medido de una petición excluyendo cualquier tiempo de espera en colas en el *SRD* (service request dispatcher).

Debemos distinguir entre la versión *básica* del algoritmo y la *ampliada*, donde se incluye pasos adicionales para dividir el tiempo de respuesta entre tiempo usando la CPU y tiempo esperando el proveedor externo. Nuestro algoritmo es conservador de la manera en que empieza estimando el tiempo de servicio como el *SLA* y lo refina iterativamente durante la sesión. Como resultado de esto, al principio de la sesión, la asignación de recursos será sobreestimada, descartando algunos clientes que podrían aceptarse sin problmas. El algoritmo presentado puede usarse para sesiones que ejecutan servicios donde no tenemos información previa en el registro de servicios. Una vez este tiempo de servicio se ha estimado, podría registrarse en el registro de servicios como punto de partida para nuevas sesiones, refinándose aún más.

Reconfiguración dinámica

El algoritmo de asignación de recursos presentado anteriormente, en la sección 5.3.1, supone que en cada punto hay una serie de servidores fijos disponibles e intenta distribuir la carga entre ellos, de manera que el *SLA* de los clientes se cumpla. Con el incremento del uso de entornos virtualizados, la ejecución de servidores bajo demanda es más común. En línea con esta moda, hemos extendido nuestro algoritmo para que soporte la adición de servidores *Grid* bajo demanda, así como la reconfiguración dinámica en el caso de que un servidor falle. Cuando no podemos ofrecer a un cliente la correspondiente calidad de servicio que necesita, el algoritmo considera la petición de un nuevo servidor adicional para ejecutar la nueva sesión. Al mismo tiempo, cuando se detecta un fallo en un servidor, el gestor de recursos reconfigura todas las sesiones que tienen threads asignados en el servidor que falla. El algoritmo extendido procede a considerar las sesiones afectadas como nuevos clientes. Las sesiones existentes podrían cancelarse en el caso de que no hubieran suficientes recursos disponibles para ofrecer la calidad de servicio requerida. Factores económicos, como los beneficios obtenidos de las sesiones e los clientes, deberían considerarse cuando decidimos qué sesiones debemos cancelar. Presentamos ahora la versión de nuestro algoritmo que soporta la adición de servidores bajo demanda. Llamamos B el conjunto de servidores apagados que tenemos disponibles que pueden ejecutarse bajo demanda. Sea $\tilde{c} = (v, \lambda, \rho)$ una nueva petición de sesión de cliente. El algoritmo es el siguiente:

```
1  C := C ∪ {c̃}
2  Ī := I^T
3  B̄ := B
4  for each ŝ ∈ S do T(c̃, ŝ) := 0
5  repeat
6   begin
7     for each ŝ ∈ Ī(v, ϵ) : T(c̃, ŝ) = 0 do
8     begin
9        T(c̃, ŝ) := ∞
10       if (∃ c ∈ C \ {c̃} : ¬P_X^T(c) ∨ ¬P_R^T(c)) then
11       begin
12          T(c̃, ŝ) := 0
13          Ī(v, ϵ) := Ī(v, ϵ) \ {ŝ}
14       end
15       else break
16    end
17    if ((∀ s ∈ S : T(c̃, s) = 0) ∨ ¬P_X^T(c̃) ∨ ¬P_R^T(c̃)) then
18    begin
19      if (∃ n ∈ B̄) then
20      begin
21         B̄ := B̄ \ {n}
22         Ī(v, ϵ) = Ī(v, ϵ) ∪ {n}
23      end
24      else break
25    end
26  end
27 until ((P_X^T(c̃) ∧ P_R^T(c̃)) ∨ ¬(∃ ŝ ∈ Ī(v, ϵ) : T(c̃, ŝ) = 0))
28 if (∀ s ∈ S : T(c̃, s) = 0) then reject c̃
29 else if (¬P_X^T(c̃) ∨ ¬P_R^T(c̃)) then
30    send counter offer o = (v, X^T(c̃), R^T(c̃))
31 else accept c̃
```

El algoritmo intenta minimizar el número de servidores en los que son asignados threads para una nueva sesión. En nuestro caso, si no podemos ofrecer la calidad de servicio

necesaria, encedemos o pedimos un nuevo servidor. Finalmente, presentamos nuestro algoritmo para la reconfiguración dinámica después de un fallo del servidor. Usamos $Alg(\hat{c}, I^T, S)$ para definir una llamada de nuestro algoritmo de gestión de recursos original de la sección 5.3.1, con sus respectivos parámetros. Sea $E(s)$ el número de fallos detectados en el servidor s (inicialmente 0).

```
1  Ī := I^T
2  while (∃ ŝ ∈ S : E(ŝ) > 0) do
3  begin
4     D := ∅
5     for each c ∈ C : T(c, ŝ) > 0 do
6     begin
7        D := D ∪ {c}
8        for each s ∈ S do T(c, s) = 0
9     end
10    S := S \ {ŝ}
11    Ī := Ī \ {ŝ}
12    for each ĉ ∈ D do Alg(ĉ, Ī, S)
13 end
```

El algoritmo presentado puede extenderse para tener en cuenta factores adicionales, como pueden ser los costes asociados a la adición de nuevos servidores, el beneficio obtenido de una nueva sesión o los costes asociados a la violación de un *SLA* de cliente. Funciones de utilidad (*Utility Functions*) pueden usarse para cuantificar la influencia de estos factores cuando tomamos decisiones [Walsh et al., 2004].

5.3.4. Entorno experimental con virtualización

El entorno experimental donde ejecutaremos las nuevas pruebas está formado por nueve servidores *Grid* en un entorno virtualizado basado en la máquina virtual de *Xen* [Dragovic et al., 2003]. Usando virtualización nos es más facil usar múltiples servidores [Figueiredo et al., 2003] en el mismo hardware y experimentar diferentes topologías. Los nueve servidores están en dos máquinas de 64-bits, una de 8 procesadores (Pentium Xeon 2.60 GHz con 9 GB de memoria) y una de 4 procesadores (Pentium Xeon a 3.16 GHz con 10 GB de memoria). La primera máquina contiene 7 servidores de 1 procesador, mientras que la segunda contiene un servidor de 1 procesador y otro de 2 procesadores. Una

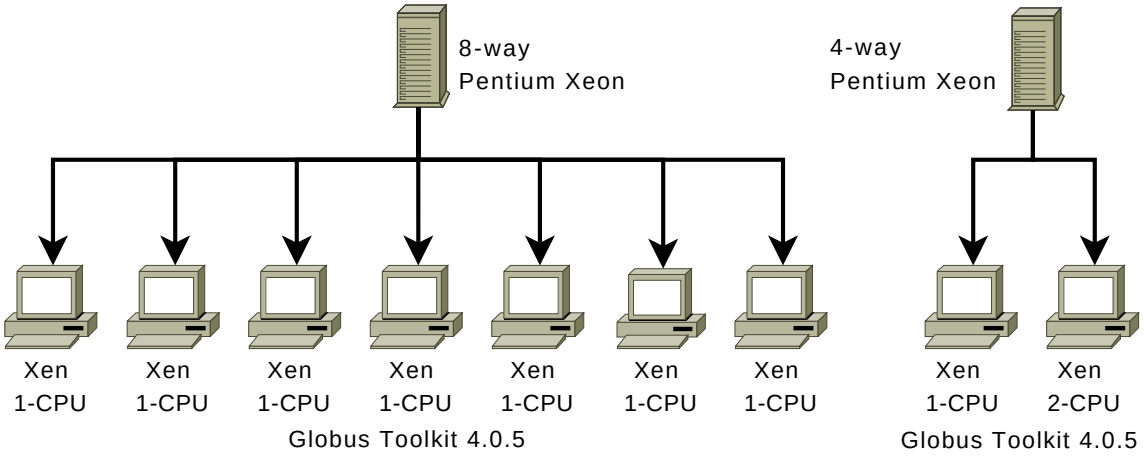


Figura 5.15: Entorno experimental virtualizado.

Cuadro 5.6: Tiempos de servicio de la carga en el entorno virtualizado.

	Servicio 1	**Servicio 2**	**Servicio 3**
Tiempo de servicio en 1 CPU (host 8 CPUs)	7,48	5,28	6,05
Tiempo de servicio en 1 CPU (host 4 CPUs)	7,17	5,19	6,22
Tiempo de servicio en 2 CPUs (host 4 CPUs)	7,04	5,07	6,04
Tiempo de servicio externo (seg)	2,00	3,00	*na*

CPU de cada servidor físico se asigna al *Domain-0* de *Xen* y el resto de los procesadores se asignan exclusivamente a cada uno de los servidores virtuales. Cada servidor ejecuta *Globus Toolkit* en su versión 4.0.5 con los últimos parches en una máquina virtual Sun 1.5.0_12 con 1 GB de memoria. Finalmente, las máquinas físicas están conectadas usando una red *Gigabit*. Un diagrama de este entorno puede observarse en la figura 5.15.

El gestor de recursos está ejecutándose en una máquina de dos procesadores como en los experimentos anteriores. Utilizaremos los mismos servicios que antes. El cuadro 5.6 muestra los tiempos de servicio de los tres servicios anteriores en los nuevos servidores virtualizados.

Caracterización de la carga en línea

En este escenario evaluamos la efectividad de nuestro entorno cuando no tenemos información acerca del comportamiento de los servicios que tenemos y de los recursos que consumen. La caracterización de la carga se realiza en línea (*on-the-fly*), usando nuestra técnica presentada en la sección 5.3.3. El experimento se realizó en el entorno virtualizado con 9 servidores. Un total de 85 sesiones se ejecutaron en un periodo cercano a las 2 horas. El experimento se repitió para diferentes configuraciones:

Configuración 1 Control de sobrecarga: Se descartan las nuevas sesiones cuando el servidor llega a una determinada utilización.

Configuración 2 Control de calidad de servicio con el modelo de la carga conocido (sabemos los tiempos de servicio)

Configuración 3 Control de calidad con la caracterización de la carga realizada en línea usando el algoritmo básico.

Configuración 4 Control de calidad con la caracterización de la carga realizada en línea usando el algoritmo ampliado.

En la primera configuración, los recursos del servidor se asignan usando un algoritmo de round-robin sin control sobre la calidad de servicio, y el gestor de recursos se configura para descartar las nuevas peticiones de sesiones si la utilización de un servidor supera el 70 %. En la segunda configuración, se activa el *QoS Control* y el gestor de recursos se configura para usar los parámetros de la carga obtenidos mediante una caracterización de la carga offline. En la tercera y la cuarta configuración, la caracterización de la carga se realiza en línea usando el algoritmo básico y el ampliado, respectivamente. La figura 5.16 muestra el tiempo de respuesta medio de las sesiones aceptadas en las cuatro configuraciones. Podemos ver los parámetros estimados de la carga (tiempo de servicio y tiempo de servicio del servidor externo) para seis servidores en la tercera y la cuarta configuración en el cuadro 5.7.

El cuadro 5.8 presenta una descomposición de las sesiones de los clientes: en i) sesiones por las que el *SLA* se ha cumplido correctamente, ii) sesiones en las que el *SLA* se ha roto y iii) sesiones descartadas por el gestor de recursos. Sin *QoS Control*, el 96 % de las sesiones se admiten; a pesar de esto, sólo se cumple el *SLA* del cliente en el 22 % de ellas. Sin embargo, en todas las configuraciones con *QoS Control*, el *SLA* se cumplió en un porcentaje cercano al 100 % de las sesiones aceptadas. Sólo 2 sesiones tuvieron una violación del *SLA*, aunque fue por un pequeño margen (un 1 % superior de lo que se requería).

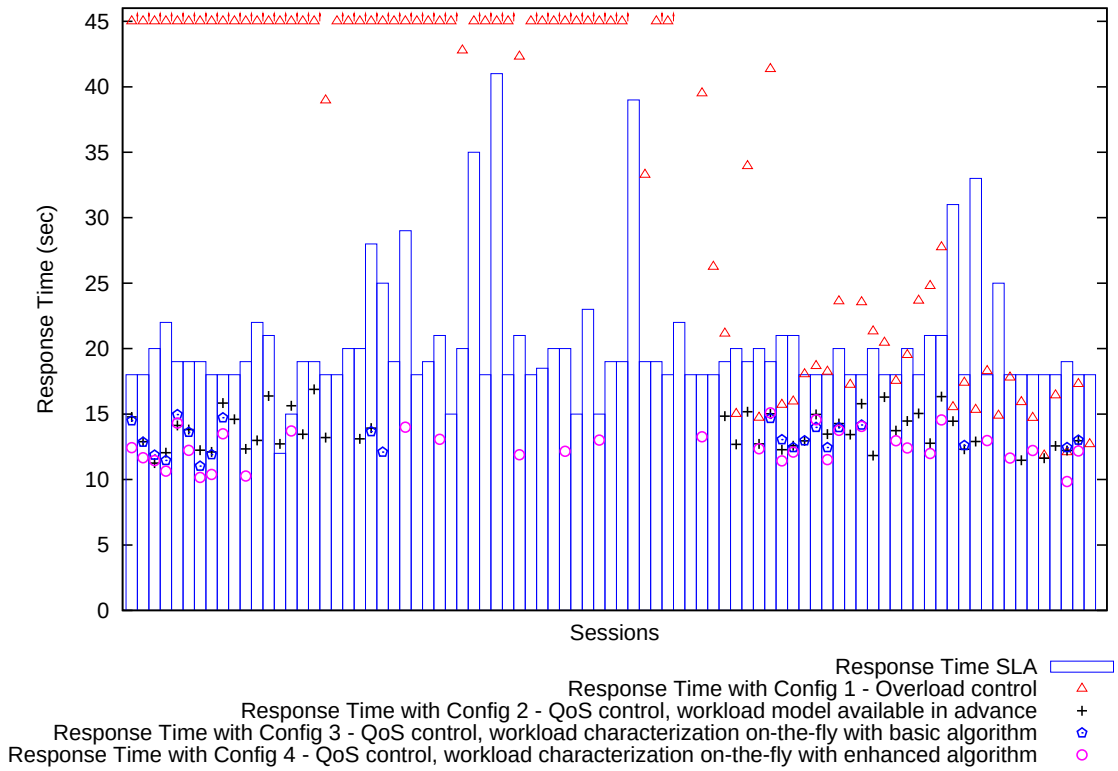


Figura 5.16: Tiempo de respuesta obtenidos con la caracterización en línea de la carga.

Como esperábamos, el precio de realizar una caracterización de la carga en línea provoca que se descarten más sesiones, dado el carácter conservador del algoritmo. En la tercera configuración con el algoritmo básico, sólo se aceptaron 22 sesiones (26 %) comparadas con las 48 (54 %) de la segunda configuración, donde conocemos la caracterización de la carga. En la cuarta configuración, sin embargo, sólo se descartaron 14 sesiones más (16 %) de las descartas por la segunda configuración. Un margen perfectamente aceptable para lo que estamos ganando con esta técnica.

Servidores Grid añadidos bajo demanda

En este apartado, evaluaremos la efectividad de nuestro entorno cuando los servidores se añaden bajo demanda como se ha explicado en la sección 5.3.3. Un total de 85 sesiones se ejecutaron en un periodo de unas 2 horas. El experimento se repitió con las siguientes 4 configuraciones.

Cuadro 5.7: Tiempos de servicio (Γ) y tiempo de espera para obtener servicio externo (Ψ) en milisegundos obtenidos en la 2º, 3ª y 4ª configuración para seis servidores.

Servidor	**Servicio**	**Configuración**					
		2		**3**		**4**	
		Γ	Ψ	Γ	Ψ	Γ	Ψ
1-way	0	7,480	2,000	9,423	0	6.973	2,450
	1	5,280	3,000	8,570	0	4,799	3,771
	2	6,050	0	6,940	0	6,662	278
1-way	0	7,480	2,000	9,513	0	7,040	2,473
	1	5,280	3,000	8,570	0	3,599	4,971
	2	6,050	0	6,720	0	6,384	336
1-way	0	7,480	2,000	9,423	0	4,052	5,371
	1	5,280	3,000	8,570	0	4,199	4,371
	2	6,050	0	6,720	0	5,309	1,411
1-way	0	7,480	2,000	9,423	0	6,502	2,921
	1	5,280	3,000	8,570	0	4,799	3,771
	2	6,050	0	6,720	0	4,701	1,719
1-way	0	7,040	2,000	9,423	0	6,973	2,450
	1	5,070	3,000	8,570	0	3,685	4,885
	2	6,040	0	6,720	0	4,838	1,882
2-way	0	7,170	2,000	9,423	0	6,973	2,450
	1	5,190	3,000	8,570	0	4,799	3,771
	2	6,220	0	6,732	0	6,395	337

Cuadro 5.8: Resumen de las violaciones de SLA en la prueba de caracterización de la carga en línea.

Configuración	**SLA cumplido**	**SLA violado**	**Sesiones descartadas**
1	19	63	3
2	46	2	37
3	22	0	63
4	34	0	51

Cuadro 5.9: Resumen de los SLA cuando tenemos servidores bajo demanda.

Configuración	**SLA cumplidos**	**SLA violados**	**Sesiones descartadas**
1	19	63	3
2	46	2	37
3	15	61	9
4	45	7	33

Configuración 1 Control de sobrecarga con los 9 servidores disponibles desde el inicio.

Configuración 2 *QoS* control con los 9 servidores disponibles desde el inicio.

Configuración 3 Control de sobrecarga con un servidor disponible y 8 servidores bajo demanda.

Configuración 4 *QoS* control con un servidor disponible y 8 servidores bajo demanda.

Las dos primeras configuraciones son las mismas que en las sesiones previas y asumen que los 9 servidores están disponibles desde el inicio del experimento. La tercera y cuarta configuración sólo tienen un servidor disponible al principio y los demás servidores se añaden bajo demanda. En la tercera configuración, un nuevo servidor se añade a la *Grid* cuando la utilización de todos los servidores supera el 70 % y llega una nueva sesión. En la cuarta, un nuevo servidor se añade cuando la calidad de servicio del nuevo cliente no puede satisfacerse con los servidores disponibles. La diferencia principal, en el caso de la adición de servidores bajo demanda, es que los recursos se asignan de forma incremental y por lo tanto el espacio de configuraciones posibles se reduce. De este modo, si tenemos todos los recursos disponibles al inicio podemos realizar un mejor balanceo de carga, ya que el espacio para ello es mayor.

La figura 5.17 muestra el tiempo de respuesta medio de las sesiones aceptadas en las cuatro configuraciones presentadas. La tabla 5.9 muestra un resumen de los *SLA* cumplidos. Los resultados muestran que añadir servidores bajo demanda no tiene un impacto significante en el rendimiento del gestor de recursos, a pesar de ser menos flexible en la distribución de la carga.

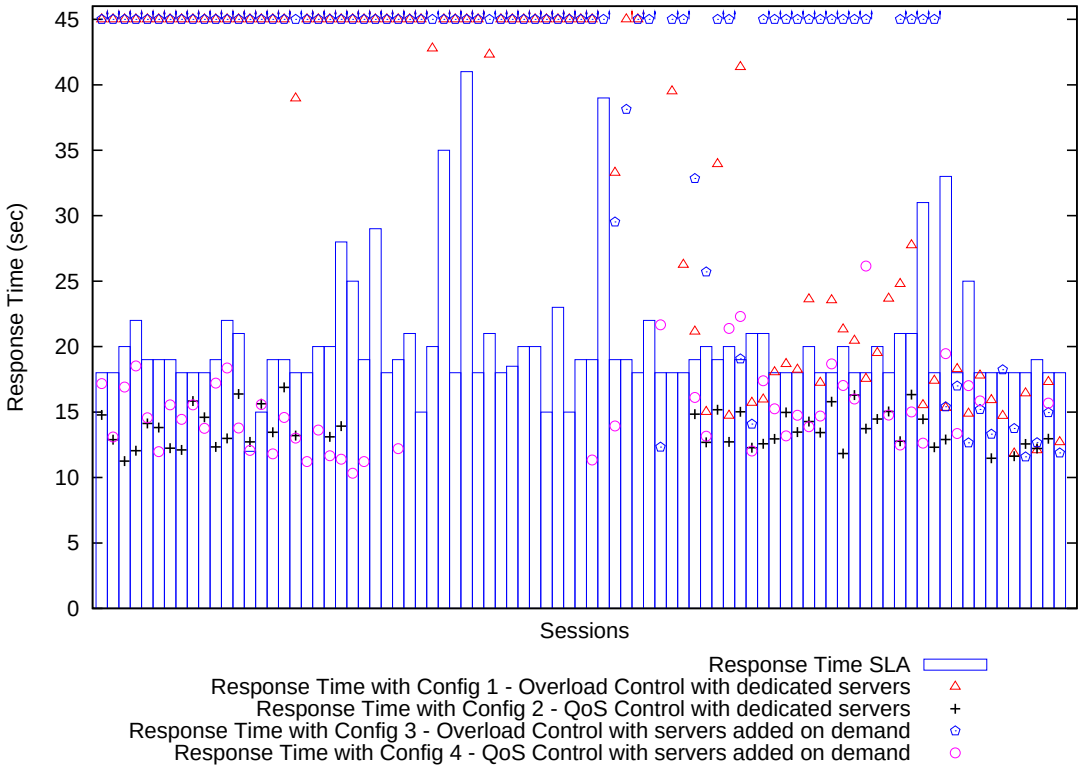


Figura 5.17: Tiempos de respuesta obtenidos añadiendo servidores bajo demanda.

5.3.5. Evaluación: Reconfiguración dinámica después de un fallo de servidor

En este último escenario, consideremos la reconfiguración dinámica del sistema después de un fallo en un servidor *Grid*. El experimento utiliza la misma configuración que las dos pruebas anteriores. Se ha repetido con 5 niveles diferentes de fallos, de 1 a 5 servidores caídos. Los puntos donde los servidores fallan se han escogido aletoriamente durante las 2 horas de la prueba. Cada vez que un servidor falla y se detecta, el gestor de recursos reconfigura todas las sesiones que tenían *threads* asignados en el servidor caido. Esto se realiza usando el algoritmo mostrado anteriormente, considerando las sesiones afectadas como nuevos clientes. Las sesiones existentes podrían tener que cancelarse en el caso de que no hubieran suficientes recursos para ofrecer una adecuada calidad de servicio.

La figura 5.18 muestra el tiempo de respuesta medio de las sesiones aceptadas en las cinco iteraciones del experimento. El cuadro 5.10 presenta un resumen de los *SLA*

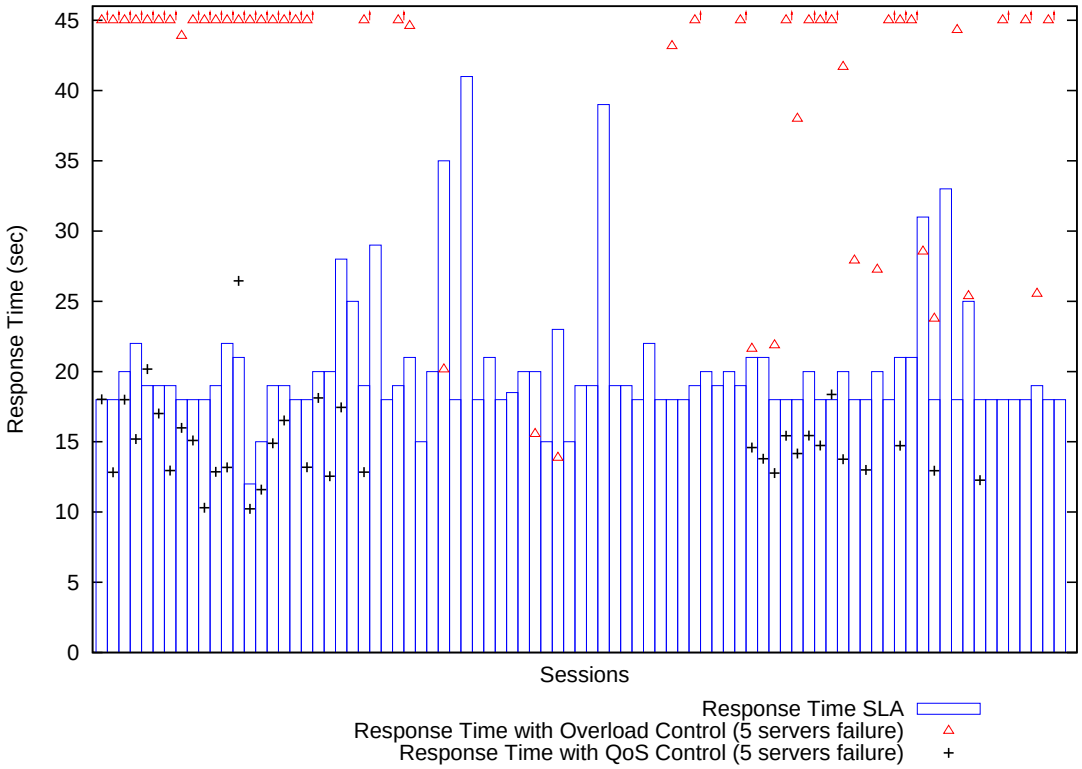


Figura 5.18: Tiempos de respuesta obtenidos en un experimento con 5 servidores caídos.

(cumplidos, violados) ejecutando las pruebas con *QoS Control* o sin *QoS Control*. En el caso sin *QoS Control*, la proporción de sesiones canceladas después de un fallo del sistema se muestra entre paréntesis en la última columna. Como podemos ver, cuando ejecutamos la prueba de una manera reactiva, el gestor de recursos todavía realiza un buen trabajo distribuyendo la carga entre los recursos disponibles y asegurando que el *SLA* del cliente se cumple.

5.4. Coste del componente de predicción

Un objetivo importante de nuestro gestor de recursos es minimizar el coste de evaluación de configuraciones alternativas usando el predictor cuando tomamos decisiones. Cuanto más detallados estén los modelos, más sobrecarga tenemos en el componente. Por eso, debemos decidir qué precisión y qué eficiencia deseamos. Para limitar el tiempo necesario para tener una decisión en los escenarios que hemos considerado, la simulación

Cuadro 5.10: Resumen de los SLA cuando utilizamos reconfiguración dinámica.

Fallos	**sin QoS Control**			**con QoS Control**		
	SLA cumplidos	**SLA violados**	**Sesiones descartadas**	**SLA cumplidos**	**SLA violados**	**Sesiones descartadas**
1	14	62	9	37	1	47 (0)
2	16	57	12	39	3	43 (1)
3	10	58	17	40	3	42 (2)
4	3	56	26	38	1	46 (6)
5	4	45	36	31	4	50 (13)

se configura para ejecutarse por un periodo limitado de tiempo. Aun siendo los resultados aproximados, nuestros experimentos muestran que son lo suficientemente precisos para garantizar una calidad de servicio aceptable para las sesiones admitidas. En los escenarios 3, 4 y 5, el tiempo medio para llegar a una decisión fue de 14,81 segundos, con un máximo de 37,35 segundos.

Hay diferentes propuestas, formando parte del trabajo futuro, para mejorar, en velocidad, el algoritmo de asignación. Por ejemplo, podemos cortar el tiempo de simulación si simplificamos el modelo agregando las sesiones que ejecutan el mismo servicio. Actualmente, en nuestros modelos cada sesión se modela de manera separada. Si dos sesiones piden el mismo servicio y sus peticiones pueden modelarse con un proceso *Poisson*, las dos sesiones pueden combinarse en una siendo sus llegadas la composición de los dos procesos de *Poisson*. Esto reduce el tamaño del modelo a coste de tener menos flexibilidad en el balanceo de carga, ya que las dos sesiones no se pueden distinguir. Usando esta propuesta, podemos simular un escenario con 40 servidores *Grid* y 80 sesiones concurrentes en menos de 60 segundos. Dado que el *QoS Control* sólo se realiza al inicio de la sesión, creemos que es un coste aceptable para aplicaciones largas.

Otra propuesta en la que estamos trabajando es la posibilidad de usar el motor SimQPN de forma que obtenga ventajas del paralelismo de los múltiples procesadores disponibles para mejorar su velocidad. Actualmente, sólo se explota el paralelismo para analizar varias configuraciones simultaneamente; aun teniendo esta posibilidad, no la utilizamos, en nuestros experimentos por simplicidad. Para modelos largos y más complejos, podemos paralelizar la simulación (simular en paralelo las subredes). Otros métodos son la utilización de caché de configuraciones, así como simular configuraciones de manera proactiva.

5.5. Trabajo relacionado

Hay una gran cantidad de trabajo en la gestión de recursos y calidad de servicio en entornos *Grid* y en arquitecturas orientadas a los servicios en general.

En [Menascé and Casalicchio, 2004] se presenta una plataforma para la asignación de recursos en *Grid*. Los autores consideran el caso general en el cual las aplicaciones se descomponen en tareas que muestran relaciones de precedencia. El problema consiste en encontrar la asignación de recursos óptima que minimiza el coste total mientras se preserva los *SLA* de tiempo de ejecución. Finalmente, se desarrolla esta plataforma para construir soluciones heurísticas para este problema (que es *NP*-Completo). En [Bennani and Menascé, 2005] muestran como las redes de colas analíticas combinadas con técnicas de búsqueda combinatoria pueden usarse para desarrollar métodos de asignación de recursos de manera óptima en centros de datos autónomos. Los trabajos mencionados no tratan con el problema de la negociación de la calidad de servicio.

En [Menascé et al., 2004b] se propone un entorno para diseñar componentes de software que tengan en cuenta aspectos de calidad de servicio. Los autores introducen los llamados *Q-Components*, que negocian las necesidades de calidad de servicio con los clientes (por ejemplo, el tiempo de respuesta medio y el *throughput*), y usan un modelo de rendimiento analítico online (más específicamente, una red de colas multiclase cerrada) para asegurar que las peticiones del cliente son aceptadas sólo si la *QoS* pedida puede ofrecerse. La misma propuesta se aplica en [Menascé et al., 2005] para el diseño de arquitecturas orientadas al servicio que tienen en cuenta la calidad de servicio. Mientras estos trabajos ofrecen soporte básico para negociar y reforzar los requerimientos de *QoS* en entornos *SOA* (orientados a los servicios), no separan completamente los usuarios del servicio de los proveedores del servicio y sufren algunos problemas. Por ejemplo, no se ofrece un balanceo de carga a nivel servicio. Además, la estrategia de balanceo y de asignación de recursos de una sesión de un cliente no puede ser reconfigurada dinámicamente. Finalmente, estos métodos basados en redes de colas con forma de producto están limitadas en los aspectos de precisión de modelado y expresividad.

Una propuesta alternativa para la gestión de recursos autónoma en data-centers con diferentes aplicaciones, es la basada en *reinforcement learning* como se propone en [Tesauro et al., 2005]. En vez de usar modelos de rendimiento explícitos, utiliza una metodología de conocimiento y de prueba de error para aprender y estimar los recursos y construir políticas óptimas de decisión. En [Tesauro et al., 2006] los autores extienden su propuesta para soportar aprendizaje offline con los datos obtenidos, mientras una política inicial (basada en un modelo explícito) se utiliza inicialmente. Una propuesta de aprendizaje activo para la asignación de recursos para cargas *batch* se propone en [Shivam et al., 2006]. Esta propuesta utiliza históricos de rendimiento capturados mediante instrumentación para ob-

tener modelos de predicción de las aplicaciones más usadas. Esta propuesta está orientada hacia la ejecución de tareas *batch*, que se ejecutan hasta que finalizan utilizando todos los recursos. Las llegadas y la concurrencia no se contempla. En [He et al., 2002] se propone un algoritmo de planificación con calidad de servicio para *Grid*. El algoritmo utiliza un modelo de predicción a largo plazo y a nivel de aplicación para predecir el tiempo de finalización de las tareas en un entorno no dedicado. Basado en el mismo modelo, se desarrolla un sistema llamado *Grid Harvest Service* [Sun and Wu, 2003]. El objetivo de este trabajo es utilizarlo en la ejecución de aplicaciones largas. En [Pacifici et al., 2005] se presenta un sistema de gestión de rendimiento para *web services* en clústers. El sistema soporta múltiples clases de tráfico de web services y asigna los recursos del servidor de forma dinámica para maximizar una función de utilidad del cluster. Se utilizan modelos de colas simples para predecir el rendimiento. Esta herramienta no soporta, actualmente, la negociación de *QoS* ni el control de admisión. Existen algunas propuestas para la caracterización autonóma de la carga: en [Xu et al., 2007] por ejemplo se basa en lógica difusa (*fuzzy logic*), en [Andrzejak et al., 2006] se basa en un algoritmo genético y, finalmente, en [Gmach et al., 2007] se basa en trazas usadas para generar patrones de comportamiento. La última técnica no es aplicable en nuestro entorno, ya que supone una carga homogénea. Hay más trabajos relacionados en el área de gestión de recursos y *QoS* en *Grid* y entornos orientados a los servicios (*SOA*); podemos encontrarlos en [Kapadia et al., 1999], [Al-Ali et al., 2004], [Adam and Stadler, 2006], [Al-Ali et al., 2005], [Xu et al., 2000], [Othman et al., 2003] y [Diao et al., 2006].

5.6. Conclusiones

En este capítulo hemos propuesto un framework para el control de la calidad de servicio (*QoS*) en entornos *Grid* usando simulación. Nuestra intención es demostrar que la simulación se puede utilizar en un gestor de recursos (prototipo) obteniendo buenos resultados. Primero hemos presentado una metodología novel para diseñar gestores de recursos autónomos que tengan en cuenta la calidad de servicio con capacidad para predecir el rendimiento de los distintos componentes *Grid*, y así conseguir gestionar los recursos de manera que los distintos *SLA* se cumplan. Posteriormente, proponemos una metodología para caracterizar la carga en línea basandonos en la información que monitorizamos. Esto nos posibilita la utilización de nuestra propuesta en servicios donde no conocemos el modelo de carga. Junto con esto, hemos extendido nuestro algoritmo para soportar la adición de servidores *Grid* bajo demanda, así como reconfigurar dinámicamente el sistema después de un fallo en un servidor.

Hemos presentado una evaluación extensiva de nuestro *framework* en dos entornos

experimentales distintos usando *Globus*. Se han realizado experimentos en cinco escenarios que explotan algunos aspectos de nuestro *framework*. En todos los escenarios en los que se activa el *QoS control*, el tiempo de respuesta medido fue estable, y cerca del 100 % de los *SLA* de los clientes fueron cumplidos. Los resultados confirman la efectividad de nuestra arquitectura para asegurar que los requerimientos de calidad de servicio se cumplen. El método propuesto para la caracterización de la carga en línea es efectivo para ofrecer estimaciones de los tiempos de servicio basándonos en la monitorización online. Hemos visto que añadir servidores bajo demanda no tiene un impacto significativo en el rendimiento del gestor de recursos aunque se observa una menor flexibilidad para distribuir la carga. Finalmente, hemos demostrado la efectividad de nuestros algoritmos de planificación cuando reconfiguramos el gestor dinámicamente después del fallo de un servidor.

Se puede extender el trabajo en múltiples direcciones, como permitir requerimientos de *QoS* duros (garantizar el 90 % de los percentiles en las métricas), ampliar el *framework* teniendo en cuenta los costes económicos o mejorar los algoritmos de búsqueda de configuraciones.

Este capítulo está basado en los siguientes artículos:

- Nou, R., Kounev, S., and Torres, J. (2007e). **Building online performance models of grid middleware with fine-grained load-balancing: A globus toolkit case study**. In Wolter, K., editor, European Performance Engineering Workshop (EPEW'07), volume 4748 of Lecture Notes in Computer Science, pages 125–140. Springer.
- Kounev, S., Nou, R., and Torres, J. (2007). **Autonomic QoSaware resource management in grid computing using online performance models**. Second International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS-2007), Nantes, France.
- Nou, R. Kounev, S., Julià, F.,and Torres, J. (2008). **Autonomic QoS control in enterprise Grid environments using online simulation**. Journal of Systems and Software. *Pendiente de publicación, Julio 2008*.

Capítulo 6

Gestor de recursos utilizando predicción online en un entorno heterogéneo

6.1. Introducción

Aumentar la potencia de los servidores aumenta el rendimiento, pero en un gran número de situaciones también incrementa el porcentaje del servidor que no realiza trabajo eficiente. Esto provoca problemas: consumimos energía (tanto para alimentar los servidores como para mantener la temperatura baja), utilizamos espacio físico (racks, conexiones a la red eléctrica) y aumentamos el mantenimiento. La solución propuesta en el capítulo 4 no reducía el mantenimiento. Ahora podemos resolver gran parte de estos problemas utilizando la virtualización, pero seguimos necesitando un gestor de recursos para repartir dinamicamente los recursos a las aplicaciones.

Aunque se están realizando esfuerzos para producir servidores enérgicamente eficientes, nuestros experimentos realizados en un servidor común (8 procesadores Pentium Xeon a 2.6 GHz) con uno de los últimos *kernel* de *linux* (2.6.23) muestra que en realidad sólo hay una diferencia de 160 watios entre el servidor sin usar (IDLE) y usándolo al 100 %. Como podemos ver en la figura 6.1, consumimos 420 y 580 watios, respectivamente (medicianes realizadas en el laboratorio mediante osciloscopio).

Con estos resultados es conveniente compartir un servidor. La virtualización reduce costes (mantenimiento, requerimientos de energía, espacio. . .) y mejora el mantenimiento de los servidores y de sus recursos. Podemos usar servicios distintos con los mismos

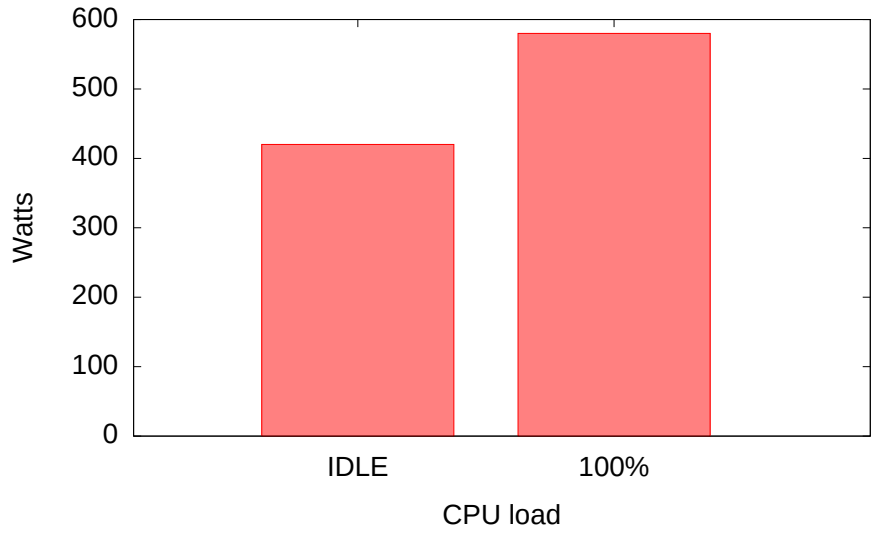


Figura 6.1: Diferencia de consumo eléctrico entre un servidor sin carga y el mismo con carga.

recursos mientras mantenemos los servidores aislados.

En este capítulo mostraremos la creación de un entorno autogestionable en un entorno heterogéneo, similar al presentado anteriormente, tomando decisiones con la información que se obtiene a través de la predicción. Nos centraremos en la aplicación práctica; el estudio de cada uno de los componentes se puede encontrar en los capítulos previos.

El prototipo que utilizaremos utilizará las dos simulaciones creadas anteriormente para distribuir los recursos del servidor en dos máquinas virtuales, una con Tomcat y la otra con Globus. Al compartir una máquina deberemos decidir el número de procesadores que debemos repartir; para ello aplicaremos dos políticas de *SLA* (*Service Level Agreement*):

1. Aplicar un *SLA* a nivel de throughput en la máquina virtual con Tomcat.
2. Aplicar un *SLA* a nivel de tiempo de respuesta en las peticiones de sesiones de Globus.

De estas dos políticas, para simplificar el análisis, tendrá prioridad la primera sobre la segunda, aunque podría asignarse al revés o incluso aplicar un tercer *SLA* de prioridad. De este modo Tomcat rendirá al máximo, siempre con su *SLA* de guía, y Globus reducirá su carga (cancelando sesiones y reduciendo el nivel de paralelismo) para mantener la *QoS* utilizando los recursos que le deje la primera aplicación.

Podemos encontrar trabajo relacionado utilizando virtualización en [Padala et al., 2007], donde un *data center* con tres aplicaciones (transaccionales) comparten recursos. Utiliza técnicas clásicas de teoría de control con una caja negra para tomar decisiones. Se ha realizado un trabajo similar en [Karve et al., 2006], ofreciendo la posibilidad de colocar aplicaciones (sólo web) de forma dinámica en un clúster y ajustar sus recursos según su demanda. Su objetivo es maximizar la utilización de los servidores. Nuestro trabajo presenta una carga heterogénea, como la presentada en el capítulo 4, y el uso de las simulaciones anteriores, que en muchos casos pueden ofrecer más flexibilidad.

6.2. Diseño del prototipo

Podemos encontrar un diagrama del sistema global en la figura 6.2. Tenemos un conjunto de 6 generadores de carga, 5 para Tomcat (para no saturar los clientes) y 1 para Globus que generan peticiones hacia los servidores virtualizados (*Tomcat VM* y *Globus VM*). Tenemos un analizador del workload y del *SLA* para Tomcat que escucha el volumen y la tipología de las conexiones para caracterizar las peticiones de los clientes. Actualmente el workload es conocido, ya que está basado en *RUBiS* [Amza et al., 2002], pero podríamos colocar un bloque de aprendizaje automático. Cuando el sistema detecta un cambio en este workload o en el *SLA* ejecutamos una serie de simulaciones para seleccionar el mínimo número de procesadores que nos asegura el *SLA* pactado (algoritmo siguiente). Finalmente, cuando obtenemos el resultado, se informa al *VM Resource Manager* (*VMRM*) de la decisión. Podemos ver como el *VMRM* tiene una flecha de regreso hacia el Analizador de Tomcat (figura 6.2) : su función es la de informar si al final se ha decidido asignar menos recursos. En el lado de Globus tenemos a *Globus QoS Broker*, mostrado anteriormente, que decide descartar, cancelar, asignar menos paralelismo o aceptar una petición de sesión de Globus cuando llega al sistema. Sus decisiones se realizan utilizando el simulador de Globus con los recursos disponibles como parámetro de entrada. Finalmente, cuando se ha de realizar un cambio en la asignación de recursos, *VMRM* realiza el cambio utilizando *Xen* en el *Dominio-0*. A continuación detallamos el procedimiento en el pseudocódigo siguiente:

```
1 if (|oldSLA − newSLA| < ε ∩
2     |oldRequest − newRequest| < ε ∩
3     |oldClientRate − newClientRate| < ε) do
4 begin
5  numCPU := 0
```

```
6   found := false
7   replies := ∅
8   while (CPU ≤ availableCPU ∪ ¬ found) do
9   begin
10    numCPU := numCPU + 1
11    replies[numCPU] := Sim (numCPU, newRequest, newClientRate)
12    found := replies[numCPU] ≥ newSLA
13  end
14  if (found) do
15     XenAssign (Tomcat, numCPU)
16  if (¬ found) do
17  begin
18   numCPU := availableCPU
19   diff := 0
20   while (diff < ε ∪ numCPU > 1 ) do
21   begin
22    diff := |replies[numCPU] − replies[numCPU − 1]|
23    numCPU := numCPU − 1
24   end
25   if (diff < ε ) do
26      XenAssign (Tomcat, numCPU)
27   else do
28      XenAssign (Tomcat, numCPU + 1)
29  end
30 end
```

Del diagrama de la figura 6.2 destacamos que el gestor de recursos (*VM Resource Manager*) sólo informa al *Broker* de Globus del número de recursos disponibles; en cambio, informa y recibe el resultado de la predicción/decisión del gestor de Tomcat. Esto es asi por la particularidad de la asignación de prioridades a las aplicaciones. El diagrama representa un escenario específico, pero puede ampliarse y crecer para soportar un número no definido de aplicaciones (servidores virtuales) o incluso abrir la posibilidad de que el *QoS Broker* de Globus reparta sesiones a otros clústers (físicos o virtuales) cuando no tiene suficiente capacidad.

En las próximas subsecciones comentaremos brevemente los cambios realizados en las dos simulaciones.

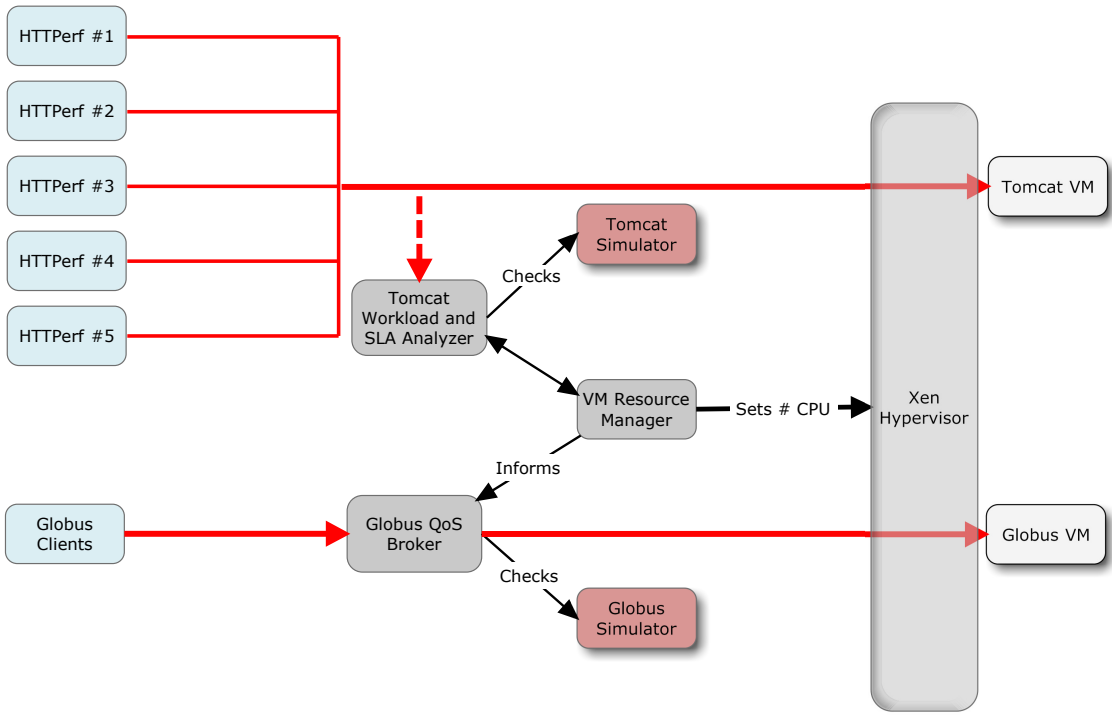


Figura 6.2: Diagrama de la composición y el funcionamiento del sistema.

6.2.1. Simulación de Tomcat

La simulación de Tomcat está basada en la primera simulación que hemos mostrado en esta tesis. Está modificada para permitir otros comportamientos: en primer lugar hemos realizado una simulación a nivel de procesador, de gran importancia en el entorno que nos ocupa. Ahora en cada una de las fases del procesado de una petición se realizan los cambios de contexto y similares que realizarían en la realidad. Otro de los cambios es la inclusión de la base de datos dentro de la máquina virtual; esto provoca que los procesadores se utilicen tanto en Tomcat como en MySQL.

Finalmente se han incluído mecanismos para permitir parar y continuar la simulación (guardar el estado), aunque no se ha utilizado finalmente. También incluimos la posibilidad de inicializar la simulación en un determinado estado; en un entorno dinámico y continuo es necesario que la simulación no parta del estado inicial sino que tenga un poco de historial. En los siguientes puntos mostraremos las particularidades de esta simulación.

Diseño de la simulación de Tomcat

Como podemos ver en la figura 6.3, la simulación de Tomcat tiene los siguientes bloques, algunos similares a la primera simulación aunque hemos ampliado algunos de ellos:

CLIENT Tiene todo el código responsable de crear las distintas peticiones, en nuestro caso simulando RUBiS, pero nada nos impide colocar algoritmos de aprendizaje para conocer la tipología de la carga del sistema. Las principales modificaciones respecto la primera simulación son la posibilidad de parar y continuar la simulación (salvando el estado) o de iniciarla en un estado diferente, obtenidos una serie de parámetros del servidor.

ACCESSProc Se comporta de igual manera que en la simulación anterior, mantenemos el control sobre la cola de backlog y añadimos los distintos controles de admisión que podamos aplicar en el sistema real.

LOADBALANCERT Es un bloque nuevo, se ha creado para facilitar la tarea de enviar y recibir trabajos hacia los distintos *HTTPProcessors*. Su tarea es la de repartir trabajo.

HTTPProcessor En realidad se trataría de una unión de los bloques que mostraremos a continuación (*Proc) y que simularía a un HTTPProcessor real. En nuestro simulador separaremos los distintos procesos para facilitar la construcción del mismo.

SSLProc Se encarga de hacer el tratamiento de seguridad de una petición.

staticProc Bloques similares a los de nuestra primera simulación. Procesa las peticiones estáticas.

dynProc Igual que el anterior bloque, pero procesa las peticiones dinámicas y envía peticiones a la base de datos *MYSQL* si es necesario.

MYSQL Simula la base de datos. A nivel de caja negra.

LOADBALANCER Realiza un balanceo de carga entre las *CPU*, hace el papel de scheduler.

CPU Procesa las tareas que le da el *LOADBALANCER*.

Como podemos ver, el principal cambio es la modelización del comportamiento de los threads y de los procesadores de la máquina virtual. Debemos realizar también un

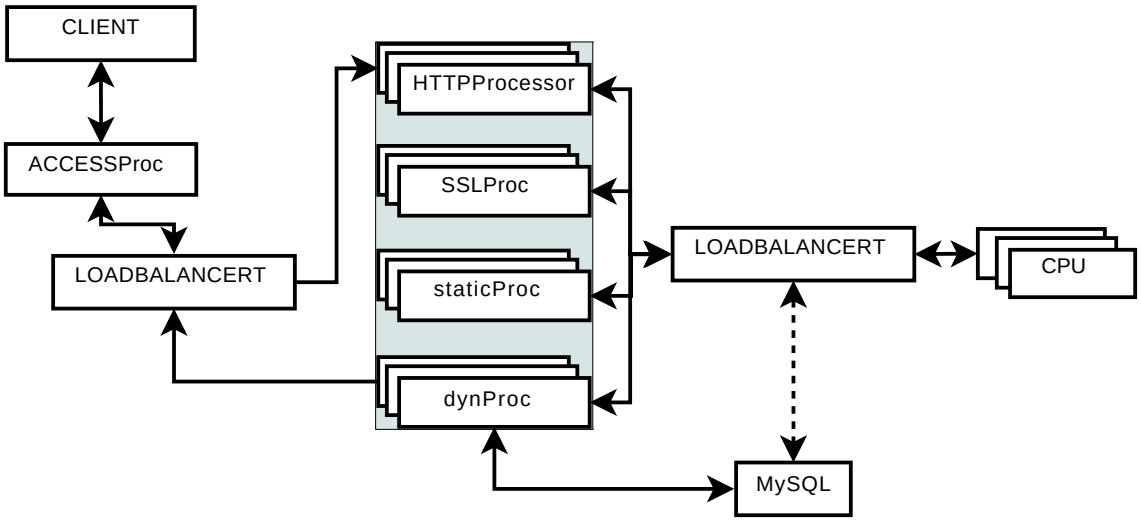


Figura 6.3: Bloques que forman la simulación de Tomcat para el nuevo entorno.

apunte, ya realizado con anterioridad, sobre la modelización de los procesadores y más concretamente del planificador: según la ley de *Little* [Little, 1961] la planificación no afecta en la obtención del tiempo de respuesta. Aunque esto es cierto en sistemas estables, aquí no sabemos cuánto tiempo vamos a disponer para simular (ahora hemos impuesto un límite de 10 segundos de CPU máximos). Por lo tanto, necesitamos que los datos en situaciones no estables sean lo más cercanos al sistema real. Por eso la planificación y otros componentes adquieren una importancia especial.

Parámetros

Podemos definir los parámetros en dos grupos: Parámetros estructurales, como lo son el número de procesadores (numCPU) y el número de *HTTPProcessors* (numHTTP, 100 por defecto); Parámetros internos, cómo el tiempo de servicio de las distintas fases (*SSL*, procesado estático, procesado dinámico. . .); y parámetros de definición de la carga, como son la duración de una sesión, el tiempo entre peticiones y la tasa de creación de sesiones.

Los parámetros de entrada de la simulación son los siguientes: Número de procesadores (numCPU) y número de *HTTPProcessors* (numHTTP), que por defecto son 100.

Los parámetros internos definidos en comparación con los de nuestro primer entorno simulado se puede encontrar en el cuadro 6.1.

Cuadro 6.1: Parámetros del nuevo entorno.

	4-way 1.4 GHz Pentium Xeon	**4-way 3.4 GHz Pentium Xeon**
SSL handshake (Primera conexión)	170 ms.	16,6 ms.
SSL handshake (Reusado)	2 ms.	0,7 ms.
Conexión, petición	3 ms.	1 ms.
Procesado página x	$1x$ s.	$0,44x$ s.

Evaluación

Para finalizar el simulador del servidor Tomcat mostraremos el resultado obtenido con un perfil de carga dinámica, que completaría el estudio realizado en el capítulo 2.

En la figura 6.4 vemos cómo se comporta el servidor ante una carga dinámica (sin control de admisión) en un servidor con 1, 2 ó 3 procesadores. Las líneas corresponden a las mediciones experimentales y el área marcada corresponde a los resultados obtenidos con simulación del mismo entorno. Vemos como el comportamiento del simulador en una prueba continua y compleja se acerca a la realidad.

6.2.2. Simulación de Globus

Para la simulación de Globus utilizamos el simulador creado en el capítulo anterior (para un clúster de nodos) realizando un cambio de semántica: hemos definido que tenemos *x* servidores de 1 CPU, donde *x* es el número de procesadores. Este cambio semántico en la definición de los servidores nos facilita el uso del mismo simulador sin realizar ningún cambio adicional. El gestor o *QoS Broker* sí que se ha modificado para que conozca la nueva realidad, ya que tiene algunas particularidades: los movimientos de trabajos entre procesadores de la misma máquina son innecesarios, por lo que sólo tiene sentido reducir el nivel de paralelismo o cancelar la sesión en el caso de reducción de recursos.

6.3. Entorno experimental

El entorno experimental se compone de un entorno virtualizado basado en la máquina virtual de Xen. Tenemos los dos servidores (Globus, Tomcat) en una máquina con 4 procesadores (Pentium Xeon a 3.16 GHz con 10 GB de memoria). Los cuatro procesadores (CPU_{Total}) se asignarán de forma dinámica a las dos máquinas virtuales, de manera que

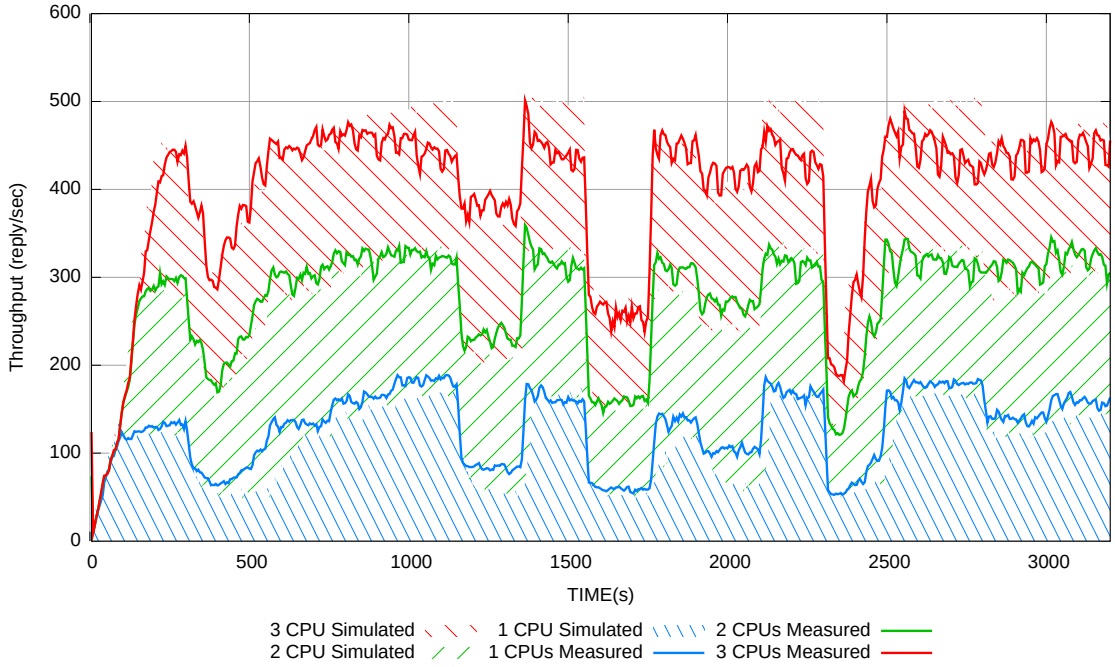


Figura 6.4: Comparación del resultado obtenido mediante simulación de un servidor Tomcat, usando 1,2 ó 3 procesadores y una carga dinámica. Las líneas representan el resultado medido experimentalmente.

$CPU_{Total} = VCPU_{Globus} + VCPU_{Tomcat}$ y $VCPU_{GLOBUS} > 0 \cup VCPU_{TOMCAT} > 0$. De esta forma conseguimos que no hayan colisiones entre dominios y que los procesadores físicos se asignen a un procesador virtual. La granularidad del reparto de procesadores se realiza a nivel procesador, aunque sería posible utilizar medios procesadores u otras unidades más pequeñas. Esto provocaría un aumento del tiempo en tomar una decisión ya que tendríamos más espacio de búsqueda. Concretando, en el caso de repartir medias CPUs, se necesitaría una simulación extra.

En la máquina virtual de Globus tenemos un Globus Toolkit 4.0.5, sin modificar. En la máquina virtual correspondiente a Tomcat, tenemos un Apache Tomcat 6.0.16, sin modificar, junto con la base de datos MySQL. Dado que el nivel de carga es mucho más elevado que en las pruebas previas, la base de datos ha de colocarse dentro de la misma máquina o realizar balanceo de carga en la base de datos. Hemos escogido colocar juntos el servidor y la base de datos por simplicidad. El cambio en el simulador es mínimo, sólo hemos de conectar las peticiones de MySQL a los mismos procesadores que Tomcat.

En cuanto a los clientes, utilizamos una sola máquina para los clientes de Globus y una

serie de 5 máquinas para los clientes de Tomcat (*Httperf* con el workload de *RUBiS*) para poder generar una carga elevada. Finalmente, una máquina de dos procesadores realiza las funciones de proxy y se encarga de controlar las conexiones y ejecutar las simulaciones. El gestor de recursos está situado en esta máquina y envía peticiones al *Domain-0* de *Xen* que contiene los servidores virtuales.

6.4. Evaluación

Para la evaluación escogeremos unas gráficas similares a las del capítulo 4. En la figura 6.5 podemos encontrar 3 gráficas. En la primera encontramos la carga o workload de los dos servidores: una línea con la carga de Tomcat que representa una carga dinámica que varía en el tiempo (varía el número de nuevos clientes por segundo que se generan). Cada nuevo cliente sigue una distribución de peticiones específicada por *RUBiS*. En segundo lugar tenemos las peticiones de nuevas sesiones(que hemos explicado en el capítulo anterior) para Globus en el periodo comprendido entre 0 y 3.000 segundos. Cada una de estas sesiones tiene un *SLA* de tiempo de respuesta asociado, así como un tipo de servicio y el número de peticiones que se realizarán en el mismo. Cada sesión Globus tiene un tiempo de vida medio de unas 80 peticiones de servicios. Hemos simplificado el *SLA* a tiempo de respuesta, pero podríamos usar otros más complejos, como *WSLA* [Dan et al., 2003], y perder beneficios por rotura de *SLA*, por ejemplo.

En el lado de la carga de Tomcat se ha de tener en cuenta que dada la naturaleza de *RUBiS* un cliente se mantiene en el sistema aproximadamente 1.900 segundos; de este modo, aunque la carga sea relativamente baja, las peticiones se van acumulando, como veremos en algunos puntos de la siguiente gráfica.

Si pasamos a la segunda gráfica veremos en primer lugar el *SLA* del servidor Tomcat. La persona que contrata el servicio nos ha ofrecido el perfil de *throughput* que quiere. Hemos escogido throughput por similitud a los otros tests, así como una mejor adaptabilidad a *RUBiS*. El *SLA* se representa en bloques, siendo independientes de la carga mostrada en la primera gráfica. La función del gestor será garantizar, como mínimo, el *SLA* pactado (si la carga lo permite) utilizando el mínimo número de procesadores posible. El throughput obtenido se representa con una línea: podemos ver como va dibujando el perfil del *SLA*, aunque hay algunas zonas que merecen especial atención, que se comentarán posteriormente.

Para finalizar esta gráfica tenemos el perfil de asignación de procesadores. Observemos como siguen el *SLA* siempre excepto en un par de zonas. Será en estas zonas donde la simulación tendrá un papel importante (en esta prueba).

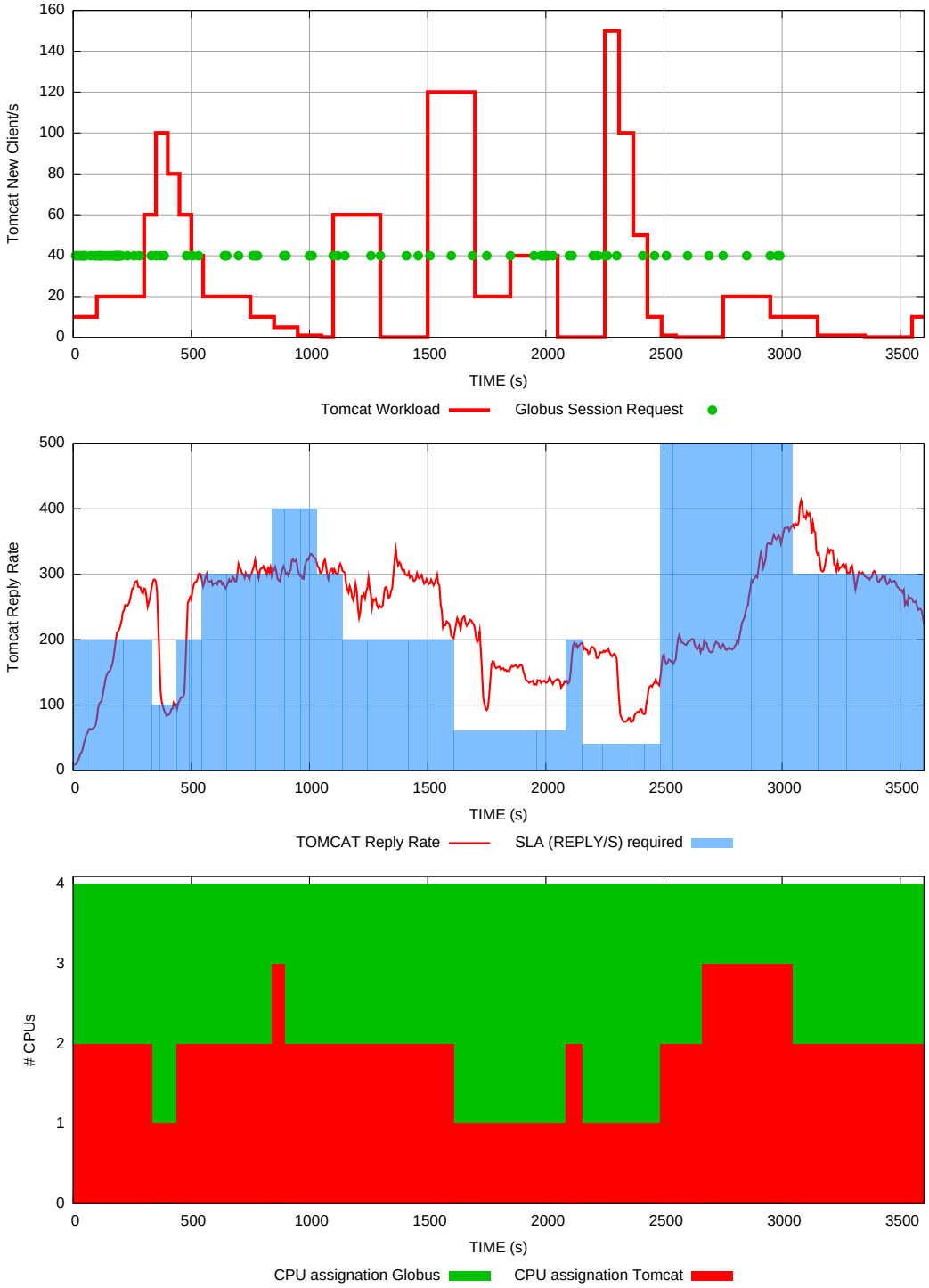


Figura 6.5: Ejecución de un entorno heterogéneo con un gestor de recursos utilizando predicción.

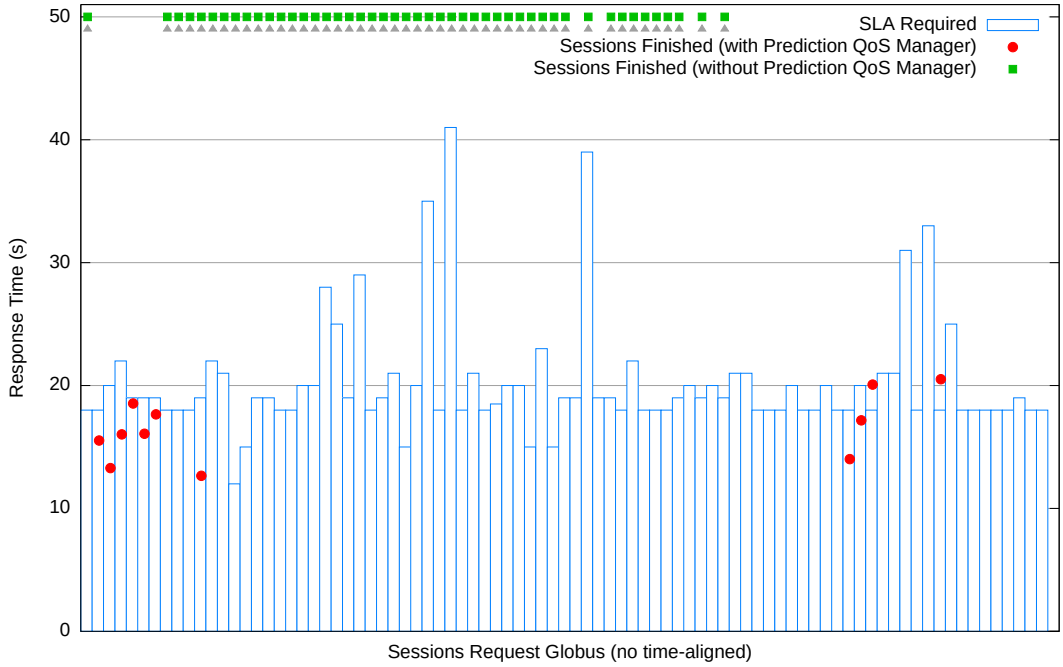


Figura 6.6: Resultado del servidor Globus utilizando el gestor de recursos con predicción. Los tiempos de respuesta que superan los valores de la escala se muestran con un triángulo.

Dentro de la misma prueba hemos separado el resultado de Globus para que resulte más claro, puesto que el resultado presentado no tiene eje temporal (como en el capítulo anterior) aunque se ha ejecutado simultáneamente. El resultado se puede observar en la figura 6.6 y tiene el mismo aspecto que las pruebas del capítulo anterior: las cajas representan el tiempo de respuesta máximo pactado con el cliente (*SLA*), los puntos son las sesiones finalizadas con éxito (observemos que hay dos sesiones por encima del valor pactado, pero con una diferencia de 2 segundos) y finalmente los cuadrados representan las sesiones finalizadas correctamente sin utilizar el control de calidad basado en predicción (el tiempo de respuesta supera la escala del gráfico).

Aunque parezcan pocas sesiones, hemos de tener en cuenta que el test de Globus es el mismo que hemos utilizado en el capítulo anterior (donde teníamos más procesadores disponibles). Además, las sesiones son largas, lo que dificulta que puedan mantenerse un largo periodo al reducir la capacidad del servidor. Por ello una vez se estan ejecutando sesiones es muy complicado que entren nuevas hasta que las que se estan ejecutando no acaben.

Para analizar los resultados de la prueba, seleccionaremos una serie de intervalos en los que ocurren situaciones interesantes. Cabe destacar que aunque la simulación o el gestor sólo tiene conocimiento del comportamiento de la carga (*RUBiS*), se puede crear una solución más general utilizando técnicas de aprendizaje, como hemos realizado en el capítulo anterior. Pasemos a explicar los distintos intervalos de la gráfica 6.5:

0-250 Como hemos comentado anteriormente, la carga es creciente. En este punto tenemos una carga cercana a los 20 clientes por segundo; el sistema es nuevo y requiere tiempo para incrementar la carga. Por eso el rendimiento necesita un período de adaptación para conseguir su carga estable.

400-500 Aunque la carga es elevada, el *SLA* nos pide que sólo ofrezcamos 100 respuestas por segundo; el gestor decide con la ayuda del simulador que el número de procesadores puede bajar a 1, con lo que el rendimiento cae hasta el valor establecido.

800-1.000 En esta zona encontramos una decisión inteligente del gestor; en primer lugar el *SLA* nos pide 400 respuestas por segundo, lo que en un primer instante y dada la carga generada podemos obtener con 3 procesadores. Pero en breves instantes la carga generada disminuye, con lo que el gestor decide que con 2 procesadores es suficiente, ya que no podremos llegar a ese nivel.

2.500-3.000 En este intervalo ocurre algo similar: el *SLA* que nos piden es muy elevado, pero la carga que detectamos no es suficiente para que podamos aumentar el rendimiento. De este modo seleccionamos 2 procesadores (1 bajaría el rendimiento). Finalmente, cuando la carga aumenta podemos pedir 3 procesadores, con lo que el rendimiento vuelve a crecer.

En la figura 6.7 comparamos la utilización de una simulación (que puede adaptarse a diferentes situaciones) con la utilización de una política estática basada en niveles, como encontramos en la ecuación 6.1. Podemos observar como hay dos momentos en los que el simulador decide utilizar menos procesadores, y si miramos el rendimiento obtenido éste no mejora respecto a la política estática. Estas situaciones se agravarían al incluir más grados de libertad en la selección de recursos.

$$\begin{cases} \text{si } 0 \leq SLA < 200 \Rightarrow VCPU_{Tomcat} = 1 \\ \text{si } 200 \leq SLA < 400 \Rightarrow VCPU_{Tomcat} = 2 \\ \text{si } SLA \geq 400 \Rightarrow VCPU_{Tomcat} = 3 \end{cases} \tag{6.1}$$

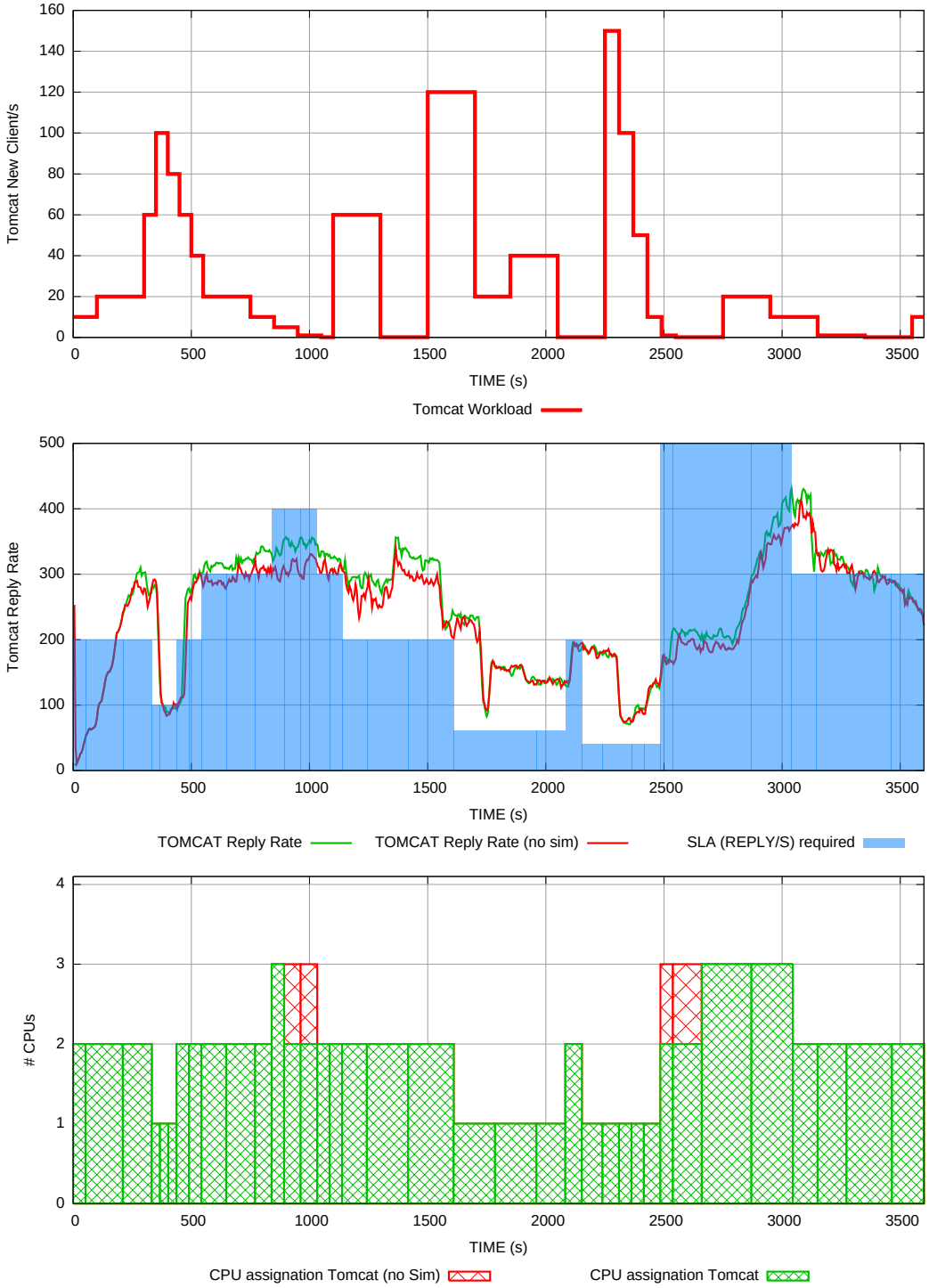


Figura 6.7: Ejecución de un entorno heterogéneo con un gestor de recursos sin predicción (se utiliza una política estática). Nótese la diferencia de asignación de procesadores sin producir beneficios en el throughput.

6.5. Trabajo relacionado

El trabajo relacionado es similar al encontrado en el capítulo 4. Destacamos los siguientes trabajos: en [Liu et al., 2005] los autores proponen el diseño de algoritmos de control basados en el feedback (online) para ajustar dinámicamente los valores de un contenedor de recursos en un servidor compartido por múltiples aplicaciones. *Cataclysm* [Uragonkar and Shenoy, 2004] realiza control de sobrecarga utilizando control de admisión, degradación del servicio adaptativo y asignación de recursos dinámicamente, demostrando que el modo más efectivo de manejar la sobrecarga es considerar una combinación de estas técnicas. Hay otras propuestas, en [Menascé, 2005] usan entornos virtuales y utilizan métodos analíticos para ajustar los recursos que se le dan a cada entorno virtual, nuestra propuesta es más versátil y ofrece un abanico más amplio de escenarios.

6.6. Conclusiones

En esta tesis hemos podido ver la aplicación de simuladores dentro de un gestor de recursos. Los resultados en el caso de nuestro prototipo de gestor de Globus han sido exitosos. En este capítulo unimos el conocimiento adquirido en los distintos capítulos para crear un prototipo gestor de recursos para un entorno heterogéneo, similar al mostrado en el capítulo 4, con capacidades de predicción mediante simulación. Hemos incorporado con éxito los simuladores creados anteriormente al sistema y hemos conseguido resultados similares sin tener que modificar el *middleware* (Tomcat y Globus). Dependiendo de las características de las aplicaciones podemos necesitar la incorporación del conocimiento del entorno virtual, es decir, número de máquinas virtuales o interacciones entre dispositivos de E/S, aunque en la mayoría de casos no nos será necesario este nivel de detalle. Gracias a estos resultados se abre la posibilidad de utilizar la simulación en este tipo de aplicación.

La simulación nos permite una mayor adaptabilidad a situaciones que serían difíciles de específicar de otra forma. En el experimento realizado en este capítulo hemos mostrado cómo se puede mantener el rendimiento y adaptar los recursos a las necesidades de las aplicaciones en el caso de Tomcat, y adaptar las aplicaciones, o su carga, a los recursos disponibles, como ocurre en el servidor Globus. Esta adaptación de la carga de Globus hace que se cumplan la mayoría de los *SLA* de los clientes (y los que no se cumplen tienen un error despreciable). La gestión se realiza con las técnicas que hemos utilizado en el anterior capítulo, con menor complejidad, puesto que no debemos recolocar trabajos. El prototipo que hemos propuesto en este capítulo permite vigilar, sin necesidad de que las decisiones se tomen simultáneamente para todas las aplicaciones, el comportamiento

y el rendimiento del *middleware*. Podemos utilizar este sistema para otros menesteres: si el rendimiento detectado es menor que el que se predice, y suponemos que el simulador funciona correctamente, puede ser que el sistema simulado esté en un proceso de *software aging* y necesite tareas de mantenimiento, como podría ser un reinicio o microreinicio.

Si realizáramos el mismo experimento pero colocando la prioridad en el servidor Globus, deberíamos decidir los clientes que debemos dejar entrar en el sistema (utilizando el control de admisión con la información obtenida en la predicción).

Capítulo 7

Conclusiones y trabajo futuro

7.1. Conclusiones

En esta tesis hemos demostrado la viabilidad de la aplicación de la simulación en tareas de gestión de recursos, utilizando aplicaciones saturadas o complejas que difícilmente pueden ser resueltas con modelos analíticos.

Hemos mostrado cómo simular un entorno complejo transaccional con una particularidad: estar saturado y sobrecargado. Gracias a la simulación hemos podido predecir con gran aproximación el comportamiento de esta aplicación y aplicar, por ejemplo, un control de admisión en la parte saturada y ver su comportamiento dentro de la simulación. Con métodos analíticos no podríamos conseguir predecir el comportamiento de esta parte.

Por otro lado, hemos ampliado la herramienta de monitorización *JIS*, convirtiéndola en *BSC-MF* (Barcelona Supercomputing Center - Monitoring Framework). Gracias a ella podemos monitorizar el comportamiento de aplicaciones mixtas, como puede ser *Globus Toolkit* (C y JAVA), y capturar los distintos parámetros o problemas de rendimiento que posteriormente modelaríamos en el simulador. Mientras utilizábamos esta herramienta hemos creado soluciones autónomas para mejorar el rendimiento de dicha aplicación.

Las dos aportaciones siguientes utilizan la simulación en gestores de recursos: en primer lugar en un entorno *Grid*, donde se aplican y cumplen políticas de *QoS* en diferentes entornos, incluso entornos dinámicos donde los servidores o recursos pueden aparecer o desaparecer; Posteriormente, ampliando el protótipo mostrado en el capítulo 4 hemos introducido los dos simuladores creados (entornos transaccionales y entorno Grid) para crear un gestor de recursos capaz de repartir procesadores entre dos aplicaciones que comparten un mismo servidor con virtualización.

Con todo esto vemos que la simulación, ligera, es una herramienta válida para utilizar en estos casos donde se necesitan respuestas rápidas y para enriquecer los gestores de recursos ofreciendo predicciones de como va a responder el sistema en el futuro. Su uso nos abre amplias posibilidades en la creación de entornos autónomos. Vamos a exponer las conclusiones separando los subobjetivos mostrados en el capítulo 1.

7.1.1. Creación de modelos de simulación para aplicaciones complejas

Utilizando como base para nuestros prototipos dos aplicaciones representativas y con diferentes naturalezas, transaccional y grid, hemos logrado simular su comportamiento con gran precisión, además de hacerlo de forma rápida.

En el primer caso, entorno transaccional, hemos seleccionado *Apache Tomcat*, servidor de aplicaciones ampliamente utilizado y cumple los estandáres. Esta aplicación (en nuestro entorno experimental) está saturada, por lo que la utilización de modelos analíticos no es posible. Gracias a la simulación podemos continuar prediciendo su comportamiento en estas zonas mientras aplicamos políticas de control de admisión. Con este prototipo podemos predecir el comportamiento en diferentes entornos o conocer las necesidades de proceso para distintas cargas.

En el caso del entorno *Grid*, hemos seleccionado *Globus Toolkit*. En su versión 4, ya que es una implementación de referencia para un entorno *Grid* además de ser una de los más extendidas. En este caso, al no tratarse de un modelo saturado podemos utilizar, parcialmente, modelos analíticos. *QPN* (*Queue Petri Nets*) nos permite modelar además contención software, es decir, casos en los que por ejemplo sólo podemos ejecutar 4 threads a la vez. Aun así, la resolución del modelo *QPN* es iterativa y se asemeja mucho a un modelo de simulación. Finalmente, hemos aplicado la misma técnica que en las aplicaciones transaccionales, utilizando simulación, con *OMNeT++*, para la predicción de su comportamiento con resultados similares.

En este subobjetivo también hemos ampliado la herramienta de monitorización *JIS* para permitirnos extraer métricas y comportamientos anómalos que serían imposibles de ver de otra forma.

7.1.2. Estudio de las capacidades de autogestión de las aplicaciones complejas

La predicción necesita un lugar para poder utilizarse. En nuestro caso queremos aplicarla en gestores de recursos; para ello, hemos creado una serie de componentes autogestionables para un entorno *Grid*, los cuales se pueden alimentar de la predicción para mejorar el rendimiento y la calidad de servicio. Además de estos componentes también hemos creado un prototipo de gestor de recursos para un nodo heterogéneo (primero sin virtualización y posteriormente con virtualización y predicción), el cuál reparte los recursos de forma inteligente entre las distintas aplicaciones. Mostramos también como la autogestión, en el caso de *Globus*, es una de las herramientas más potentes para salvar la complejidad de la aplicación y mejorar su rendimiento.

7.1.3. Creación de entornos de autogestión para entornos complejos

Junto con las técnicas de los subobjetivos anteriores creamos finalmente dos gestores de recursos para dos niveles distintos. El primer nivel, *middleware* Grid, nos permite tener un gestor de recursos con capacidades para controlar la *QoS* utilizando predicción. Gracias a él y utilizando dos de las técnicas y métodos de simulación utilizados (*QPN* y *OMNeT++*), conseguimos mejorar la calidad de servicio de Globus, cumpliendo las peticiones de *SLA* de los clientes. En un segundo nivel utilizamos todos los simuladores y gestores creados para incorporarlos en un gestor de recursos global. Este gestor controla las aplicaciones que se ejecutan en un nodo compartido por varias aplicaciones de naturalezas distintas. En el último capítulo, este prototipo se alimenta de las distintas predicciones para realizar un control más inteligente y sin la necesidad de modificar las aplicaciones. Esta gestión se realiza en tiempo real y permite mantener la calidad de servicio en Globus (manteniendo los *SLA* de los clientes) y gestionar los recursos que se ofrecen a Tomcat según la carga dinámica que le llega.

7.2. Trabajo Futuro

La aplicación de la simulación en un campo dinámico, como puede ser la gestión de recursos, nos abre un abanico de posibilidades para continuar el trabajo desarrollado en esta tesis:

Mejora de la velocidad de la simulación Tanto usando *OMNeT++* como utilizando *QPN* podemos mejorar la velocidad utilizando el paralelismo. Los frameworks ofrecen

en muchas ocasiones la posibilidad de dividir la simulación para poder realizarla en diferentes máquinas. En el caso de *QPN*, hemos introducido brevemente esta posibilidad con los modelos jerárquicos, aun así no es trivial conseguir este paralelismo.

Mejora de la velocidad en la toma de decisiones La simulación aunque sea rápida necesita su tiempo. En nuestro caso podemos mejorar la toma de decisiones (reducir el tiempo para escoger una configuración) utilizando una caché de resultados. También la utilización de heurísticas nos ofrecerían mejoras en cuanto a velocidad mientras mantenemos la calidad del gestor. Por ejemplo, si tenemos una simulación en la que un trabajo en un entorno similar ya ha sido realizado, podemos tomar la misma decisión si un nuevo trabajo tiene más requisitos.

Combinación con algoritmos genéticos Relacionado con el punto anterior podemos explorar diferentes algoritmos evolutivos o de inteligencia artificial (machine learning) para mejorar las decisiones tomadas.

Incorporación de inteligencia económica La incorporación en la toma de decisiones de parámetros económicos, o incluso de parámetros medioambientales, es importante en entornos empresariales, donde nuestras decisiones han de producir el máximo beneficio.

Simulación de otros entornos En particular, un entorno en el que trabajamos es el de simular el comportamiento de *Xen*, y concretamente su planificador e hypervisor, cuando tenemos distintas máquinas virtuales y distintas aplicaciones con necesidades dispares. Gracias a ello podríamos llegar a mostrar con seguridad si un determinado *SLA* se va a cumplir o no dados una serie de parámetros del scheduler (credit_scheduler).

Bibliografía

[Abdelzaher et al., 2002] Abdelzaher, T., Shin, K., and Bhatti, N. (2002). Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE TPDS*, 13(1):80–96.

[Adam and Stadler, 2006] Adam, C. and Stadler, R. (2006). A Middleware Design for Large-scale Clusters Offering Multiple Services. *IEEE electronic Transactions on Network and Service Management*, 3(1).

[Agarwal et al., 2003] Agarwal, M., Bhat, V., Liu, H., and Matossi, V. (2003). Automate: Enabling autonomic applications on the grid. *Proceedings of Active Middleware Services (AMS)*.

[AIX, 2006] AIX, I. (2006). trace utility on ibm's aix. `http://publib.boulder.ibm.com`.

[Al-Ali et al., 2004] Al-Ali, R., Amin, K., von Laszewski, G., Rana, O., Walker, D., Hategan, M., and Zaluzec, N. (2004). Analysis and Provision of QoS for Distributed Grid Applications. *Journal of Grid Computing*, 2(2).

[Al-Ali et al., 2005] Al-Ali, R., Rana, O., von Laszewski, G., Hafid, A., Amin, K., and Walker, D. (2005). A Model for Quality-of-Service Provision in Service Oriented Architectures. *Journal of Grid and Utility Computing*.

[Alliance, 2005a] Alliance, G. (2005a). Globus toolkit documentation java WS-Core component. `http://www.globus.org/toolkit/docs/4.0/common/javawscore/`.

[Alliance, 2005b] Alliance, G. (2005b). Globus toolkit documentation, WS-GRAM. `http://www.globus.org/toolkit/docs/4.0/execution/`.

[Alliance, 2005c] Alliance, G. (2005c). Globus toolkit WS-GRAM developer documentation. `http://www.globus.org/toolkit/docs/4.0/execution/wsgram/developer-index.html`.

[Almeida et al., 2006] Almeida, J., Almeida, V., Ardagna, D., Francalanci, C., and Trubian, M. (2006). Resource management in the autonomic service-oriented architecture. In *ICAC*, pages 84–92.

[Amza et al., 2002] Amza, C., Cecchet, E., Chandra, A., Cox, A., Elnikety, S., Gil, R., Marguerite, J., Rajamani, K., and Zwaenepoel, W. (2002). Specification and implementation of dynamic web site benchmarks. *WWC-5, Austin, Texas, USA*.

[Andrzejak et al., 2002] Andrzejak, A., Arlitt, M., and Rolia, J. (2002). Bounding the resource savings of utility computing models. *HPL-2002-339, HP Labs*.

[Andrzejak et al., 2006] Andrzejak, A., Graupner, S., and Plantikow, S. (2006). Predicting resource demand in dynamic utility computing environments. *International Conference on Autonomic and Autonomous Systems (ICAS'06)*, 0:6.

[Anjomshoaa et al., 2005] Anjomshoaa, A., Brisard, F., Drescher, M., Fellows, D., Ly, A., McGough, S., Pulsipher, D., and Savva, A. (2005). Job submission description language (jsdl) specification. `http://www.gridforum.org/documents/GFD.56.pdf`.

[Appleby et al., 2001] Appleby, K., Fakhouri, S., Fong, L., Goldszmidt, G., Krishnakumar, S., Pazel, D., Pershing, J., and Rochwerger, B. (2001). Oceano :SLA-based management of a computing utility. *IM 2001, Seattle, Washington, USA*, pages 855–868.

[Backlog, 2008] Backlog (2008). TCP connection, backlog queue information. www.cs.rice.edu/CS/Systems/Web-measurement/paper/node3.html.

[Banga et al., 1999] Banga, G., Druschel, P., and Mogul, J. C. (1999). Resource containers: A new facility for resource management in server systems. *OSDI'99,New Orleans, Louisiana, USA.*, pages 45–58.

[Bause, 1993] Bause, F. (1993). "QN + PN = QPN-- Combining Queueing Networks and Petri Nets. Technical report no.461, Department of CS, University of Dortmund, Germany.

[Bause and Buchholz, 1998] Bause, F. and Buchholz, P. (1998). Queueing Petri Nets with Product Form Solution. *Performance Evaluation*, 32(4):265–299.

[Bause et al., 1995] Bause, F., Buchholz, P., and Kemper, P. (1995). Qpn-tool for the specification and analysis of hierarchically combined queueing petri nets. *In Quantitative Evaluation of Computing and Communication Systems, Lecture Notes in Computer Science, No. 977, Springer-Verlag*.

[Bause et al., 1997] Bause, F., Buchholz, P., and Kemper, P. (1997). Integrating Software and Hardware Performance Models Using Hierarchical Queueing Petri Nets. In *Proc. of the 9. ITG / GI- Fachtagung Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen*.

[Bennani and Menascé, 2005] Bennani, M.Ñ. and Menascé, D. A. (2005). Resource Allocation for Autonomic Data Centers using Analytic Performance Models. In *Proc. of the 2nd Intl. Conference on Automatic Computing*.

[Bolch et al., 1998] Bolch, G., Greiner, S., de Meer, H., and Trivedi, K. S. (1998). *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. Wiley-Interscience, New York, NY, USA.

[Borland, 2006] Borland (2006). Borland's optimizeit product. `http://www.borland.com/us/products/optimizeit/index.html`.

[Bratley et al., 1987] Bratley, P., Fox, B., and Schrage, L. (1987). *A Guide to Simulation*. Springer-Verlag.

[Buch and Pentkovski, 2001] Buch, D. and Pentkovski, V. (2001). Experience in characterization of typical multi-tier e-Business systems using operational analysis. *Proc. CMG Conference*, pages 671–681.

[Buyya et al., 2005] Buyya, R., Abramson, D., and Venugopal, S. (2005). The grid economy. *Proceedings of the IEEE*, 93(3):698–714.

[Carrera et al., 2005] Carrera, D., Guitart, J., Beltran, V., Torres, J., and Ayguadé, E. (2005). Performance impact of the grid middleware. *Engineering the Grid: Status and Perspective, B. DiMartino, J.Dongarra, A. Hoisie, L. Yang and H. Zima. Nova Science Publisher 2005, ISBN:1-58883-038-1*, pages 220–242.

[Carrera et al., 2003] Carrera, D., Guitart, J., Torres, J., Ayguadé, E., and Labarta, J. (March 2003). Complete instrumentation requirements for performance analysis of web based technologies. *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Austin, USA*.

[Caubet et al., 2006] Caubet, J., Hogan, K., Nou, R., Labarta, J., and Torres, J. (2006). Supporting the Introduction of Autonomic Computing in Middleware. *Engineering Conference VII, IBM Academy of Technology Conference IBM Hursley , England , October 2006*.

[Chandra et al., 2003a] Chandra, A., Gong, W., and Shenoy, P. (June 2-4, 2003a). Dynamic resource allocation for shared data centers using online measurements. *11th International Workshop on Quality of Service (IWQoS 2003), Berkeley, California,USA*, pages 381–400.

[Chandra et al., 2003b] Chandra, A., Goyal, P., and Shenoy, P. (2003b). Quantifying the benefits of resource multiplexing in on-demand data centers. *Self-Manage 2003, San Diego, California, USA*.

[Chandra and Shenoy, 2003] Chandra, A. and Shenoy, P. (2003). Effectiveness of dynamic resource allocation for handling internet flash crowds. *TR03-37, Department of Computer Science, University of Massachusetts, USA*.

[Chao et al., 2003] Chao, H., Cheng, F., Xi, H., Ettl, M., and Buckley, S. (2003). A simulation-based tool for inventory analysis in a server computer manufacturing environment. *Winter Simulation Conference*.

[Chen and Mohapatra, 2003] Chen, X. and Mohapatra, P. (2003). Overload control in QoS-aware web servers. *Computer Networks*, 42 (1):119–133.

[Cherkasova and Phaal, 2002] Cherkasova, L. and Phaal, P. (2002). Session-based admission control: A mechanism for peak load management of commercial web sites. *IEEE Transactions on Computers*, 51 (6):669–685.

[Cherkasova and Rolia, 2006] Cherkasova, L. and Rolia, J. (2006). R-Opus: A composite framework for application performability and qos in shared resource pools. In *DSN'06*, pages 526–535, Washington, DC, USA.

[Chess et al., 2005] Chess, D. M., Pacifici, G., Spreitzer, M., Steinder, M., and Tantawi, A. (2005). Experience with collaborating managers: Node group manager and provisioning manager. *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 39–50.

[CHIBA, 1998] CHIBA, S. (1998). Javassist - a reflection-based programming wizard for java. *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*.

[Chinnici et al., 2006] Chinnici, R., Moreau, J.-J., Ryman, A., and Weerawarana, S. (2006). Web Services Description Language (WSDL) Version 2.0. Technical report, W3C. http://www.w3.org/TR/wsdl20.

[Chu, 1998] Chu, Y. (1998). Analytic modeling and measurement methodology for multi-tier client-server systems. *Ph.D. Dissertation, University of Michigan*.

[Coarfa et al., 2002] Coarfa, C., Druschel, P., and Wallach, D. (2002). Performance analysis of TLS web servers. *NDSS'02,San Diego, California, USA*.

[Corp., 2002] Corp., M. (2002). Event tracing for windows (etw). `http://windowssdk.msdn.microsoft.com`.

[Crovella et al., 1999] Crovella, M., Frangioso, R., and Harchol-Balter, M. (1999). Connection scheduling in web servers. *USITS'99, Boulder, Colorado, USA*.

[Dan et al., 2003] Dan, A., Ludwig, H., and Pacifici, G. (2003). Web Service Differentiation with Service Level Agreements. In *White Paper, IBM Corporation*.

[Denning and Buzen, 1978] Denning, P. and Buzen, J. P. (1978). The Operational Analysis of Queueing Network Models. *ACM Computing Surveys*, 10(3):225–261.

[Diao et al., 2006] Diao, Y., Hellerstein, J. L., Parekh, S., Shaikh, H., Surendra, M., and Tantawi, A. (2006). Modeling Differentiated Services of Multi-Tier Web Applications. In *14th IEEE International Symposium on Modeling, Analysis, and Simulation*.

[Dierks and Allen, 1999] Dierks, T. and Allen, C. (1999). The TLS protocol, version 1.0. *RFC 2246*.

[Dmitriev, 2003] Dmitriev, M. (2003). Design of jfluid: Profiling technology and tool bases on dynamic bytecode instrumentation. *Sun Microsystems Technical Report 2003-0820*.

[Doyle et al., 2003] Doyle, R., Chase, J., Asad, O., Jin, W., and Vahdat, A. (2003). Model-based resource provisioning in a web service utility. *USITS'03, Seattle, Washington, USA*.

[Dragovic et al., 2003] Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Pratt, I., Warfield, A., Barham, P., and Neugebauer, R. (2003). Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*.

[Figueiredo et al., 2003] Figueiredo, R., Dinda, P., and Fortes, J. (19-22 May 2003). A case for grid computing on virtual machines. *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pages 550–559.

[Foster, 2005] Foster, I. (2005). Globus Toolkit Version 4: Software for Service-Oriented Systems. In *Proceedings of the 2005 IFIP International Conference on Network and Parallel Computing*, pages 2–13.

[Foster and Kesselman, 2003] Foster, I. and Kesselman, C. (2003). *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, second edition. ISBN: 1558609334.

[Foster et al., 2002a] Foster, I., Kesselman, C., Nick, J., and Tuecke, S. (2002a). The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration.

[Foster et al., 2002b] Foster, I., Kesselman, C., Nick, J. M., and Tuecke, S. (2002b). Grid Services for Distributed System Integration. *Computer*, 35(6):37–46.

[Foster et al., 2001] Foster, I., Kesselman, C., and Tuecke, S. (2001). The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. J. High Perform. Comput. Appl.*, 15(3).

[FreeBSD, 2008] FreeBSD (2008). ktrace for freebsd. `http://www.freebsd.org`.

[Freier et al., 1996] Freier, A. O., Karlton, P., and Kocher, C. (1996). The SSL protocol, version 3.0. *INTERNET-draft*.

[Gmach et al., 2007] Gmach, D., Rolia, J., Cherkasova, L., and Kemper, A. (27-29 Sept. 2007). Workload analysis and demand prediction of enterprise data center applications. *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, pages 171–180.

[GridForum, 2006] GridForum (2006). Open Grid Services Infrastructure Working Group (OGSI-WG). `http://forge.gridforum.org/projects/ogsi-wg`.

[Guitart, 2005] Guitart, J. (2005). *Performance Improvement of MultiThreaded Java Applications Execution on Multiprocessor Systems*. PhD thesis, Universitat Politècnica de Catalunya.

[Guitart et al., 2005a] Guitart, J., Beltran, V., Carrera, D., Torres, J., and Ayguadé, E. (2005a). Characterizing secure dynamic web applications scalability. *19th International Parallel and Distributed Processing Symposium, Denver, Colorado, USA*, pages 166–176.

[Guitart et al., 2005b] Guitart, J., Beltran, V., Carrera, D., Torres, J., and Ayguadé, E. (2005b). Session-based adaptative overload control for secure dynamic web application. *ICPP-05, Oslo, Norway*.

[Guitart et al., 2006] Guitart, J., Carrera, D., Beltran, V., Torres, J., and Ayguadé, E. (2006). Preventing secure web applications overload through dynamic resource provisioning and admission control. *UPC-DAC-RR-2006-37*.

[Harkema et al., 2003] Harkema, M., Quartel, D., van der Mei, R., and Gijsen, B. (2003). Jpmt: a java performance monitoring tool. *In Proc. of TOOLS 2003, Urbana, Illinois, USA*.

[Hau et al., 2003] Hau, J., Lee, W., and Newhouse, S. (2003). Autonomic service adaptation in iceni using ontological annotation. In *GRID*, pages 10–17.

[He et al., 2002] He, X., Sun, X., and Laszewski, G. (2002). A QoS Guided Scheduling Algorithm for Grid Computing. In *Proceedings of the Int'l Workshop on Grid and Cooperative Computing*.

[IBM-Corporation, 2004] IBM-Corporation (2004). An architectural blueprint for autonomic computing. http://www.ibm.com/autonomic.

[Iosup et al., 2006] Iosup, A., Dumitrescu, C., Epema, D., Li, H., and Wolters., L. (2006). How are real grids used? the analysis of four grid traces and its implications. In *7th IEEE/ACM International Conference on Grid Computing (Grid2006)*.

[JIP, 2006] JIP (2006). Java interactive profiler. `http://jiprof.sourceforge.net`.

[Jost et al., 2003] Jost, G., Jin, H., Labarta, J., Gimenez, J., and Caubet, J. (2003). Performance analysis of multilevel parallel applications on shared memory architectures. *International Parallel and Distributed Processing Symposium (IPDPS), Nice, France*.

[Kapadia et al., 1999] Kapadia, N. H., Fortes, J., and Brodley, C. E. (1999). Predictive Application-Performance Modeling in a Computational Grid Environment. In *8th IEEE International Symposium on High Performance Distributed Computing*.

[Karve et al., 2006] Karve, A., Kimbrel, T., Pacifici, G., Spreitzer, M., Steinder, M., Sviridenko, M., and Tantawi, A. (2006). Dynamic placement for clustered web applications. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 595–604, New York, NY, USA. ACM.

[Kephart, 2005] Kephart, J. O. (2005). Research challenges of autonomic computing. *ICSE*, pages 15–22.

[Kounev, 2006] Kounev, S. (2006). Performance Modeling and Evaluation of Distributed Component-Based Systems using Queueing Petri Nets. *IEEE Transactions on Software Engineering*, 32(7):486–502.

[Kounev and Buchmann, 2003] Kounev, S. and Buchmann, A. (2003). Performance modelling of distributed e-business applications using queuing petri nets. In *Proc. of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'03)*.

[Kounev and Buchmann, 2006] Kounev, S. and Buchmann, A. (2006). SimQPN- a tool and methodology for analyzing queueing Petri net models by means of simulation. *Performance Evaluation*, 63(4-5):364–394.

[Kounev et al., 2007] Kounev, S., Nou, R., and Torres, J. (2007). Autonomic QoS-aware resource management in grid computing using online performance models. *Second International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS-2007), Nantes, France*.

[Lambert and Podgurski, 1999] Lambert, J. and Podgurski, H. A. (1999). xdprof: a tool for the capture and analysis of stack traces in a distributed java system. *Proceedings-SPIE, International Society for Optical Engineering, USA*.

[Law and Kelton, 2003] Law, A. and Kelton, W. (2003). *Simulation Modeling and Analysis*. McGraw Hill.

[Little, 1961] Little, J. D. C. (1961). A proof of the queuing formula l = λ w. *Operations Research, vol.9, pp. 383-387*.

[Liu et al., 2005] Liu, X., Zhu, X., Singhal, S., and Arlitt, M. (2005). Adaptive entitlement control to resource containers on shared servers. *IM 2005, Nice, France*.

[Liu et al., 2001] Liu, Z., Squillante, M., and Wolf, J. (2001). On maximizing service-level-agreement profits. *EC 2001, Tampa, Florida, USA.*, pages 213–223.

[Manual, 2008] Manual, J. U. (2008). Sun microsystems - jvmti. `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html`.

[McGuinness and Murphy, 2005] McGuinness, D. and Murphy, L. (May, 2005). A simulation model of a multi-server ejb system. *A-MOST 2005,St Louis, Missouri - USA*.

[McGuinness et al., 2004] McGuinness, D., Murphy, L., and Lee, A. (December, 2004). Issues in developing a simulation model of an ejb system. *CMG 2004, Las Vegas, Nevada*.

[Menascé, 2005] Menascé, D. A. (2005). Virtualization: Concepts, applications, and performance modeling. *Int. CMG Conference, Orlando, Florida, USA*, pages 407–414.

[Menascé et al., 2004a] Menascé, D. A., Almeida, V., and Dowdy, L. (2004a). *Performance by Design*. Prentice Hall.

[Menascé and Bennani, 2003] Menascé, D. A. and Bennani, M.Ñ. (2003). On the Use of Performance Models to Design Self-Managing Computer Systems. *Proc. 2003 Computer Measurement Group Conference*, pages 7–12.

[Menascé et al., 2005] Menascé, D. A., Bennani, M.Ñ., and Ruan, H. (2005). *Self-Star Properties in Complex Information Systems*, volume 3460 of *LNCS*, chapter On the Use of Online Analytic Performance Models in Self-Managing and Self-Organizing Computer Systems. Springer Verlag.

[Menascé and Casalicchio, 2004] Menascé, D. A. and Casalicchio, E. (2004). A Framework for Resource Allocation in Grid Computing. In *Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*.

[Menascé and Gomaa, 2000] Menascé, D. A. and Gomaa, H. (2000). A Method for Design and Performance Modeling of Client/Server Systems. *IEEE Transactions on Software Engineering*, 26(11).

[Menascé et al., 2004b] Menascé, D. A., Ruan, H., and Gomaa, H. (2004b). A Framework for QoS-Aware Software Components. In *Proceedings of the 4th International Workshop on Software and Performance*.

[Microsystems, 2008] Microsystems, S. (2008). Dtrace for solaris. http://www.sun.com/bigadmin/content/dtrace.

[Mosberger and Jin, 1998] Mosberger, D. and Jin, T. (1998). httperf: A tool for measuring web server performance. *Workshop on Internet Server Performance (WISP'98) (in conjunction with SIGMETRICS'98), pp. 59-67. Madison, Wisconsin, USA*.

[Nou et al., 2005] Nou, R., Guitart, J., Beltran, V., Carrera, D., Montero, L., Torres, J., and Ayguadé, E. (2005). Simulating Complex Systems with a Low-Detail Model. In *XVI Jornadas de Paralelismo*, pages 201–308, Granada, Spain.

[Nou et al., 2006a] Nou, R., Guitart, J., Carrera, D., and Torres, J. (2006a). Experiences with simulations - a light and fast model for secure web applications. In *ICPADS (1)*, pages 177–186. IEEE Computer Society.

[Nou et al., 2006b] Nou, R., Guitart, J., and Torres, J. (2006b). Simulating and modeling secure web applications. In Alexandrov, V.Ñ., van Albada, G. D., Sloot, P. M. A., and

Dongarra, J., editors, *International Conference on Computational Science (1)*, volume 3991 of *Lecture Notes in Computer Science*, pages 84–91. Springer.

[Nou et al., 2006c] Nou, R., Julià, F., Carrera, D., Hogan, K., Caubet, J., Labarta, J., and Torres, J. (2006c). Monitoring and analysing a grid middleware node. *Proc. 7th IEEE/ACM International Conference on Grid Computing 2006, GRID 2006, Barcelona, Spain*.

[Nou et al., 2007a] Nou, R., Julià, F., Carrera, D., Hogan, K., Caubet, J., Labarta, J., and Torres, J. (2007a). Monitoring and analysis framework for grid middleware. In *PDP*, pages 129–133. IEEE Computer Society.

[Nou et al., 2007b] Nou, R., Julià, F., Guitart, J., and Torres, J. (2007b). Dynamic resource provisioning for self-adaptive heterogeneous workloads in SMP hosting platforms. *International Conference on E-business (2nd) ICE-B 2007, Barcelona, Spain.*

[Nou et al., 2007c] Nou, R., Julià, F., and Torres, J. (2007c). Should the grid middleware look to self-managing capabilities? *The 8th International Symposium on Autonomous Decentralized Systems (ISADS 2007) Sedona, Arizona*.

[Nou et al., 2007d] Nou, R., Julià, F., and Torres, J. (2007d). The need for self-managed access nodes in grid environments. *4th IEEE Workshop on Engineering of Autonomic and Autonomous Systems (EASe 2007) Tucson, Arizona*.

[Nou et al., 2007e] Nou, R., Kounev, S., and Torres, J. (2007e). Building online performance models of grid middleware with fine-grained load-balancing: A globus toolkit case study. In Wolter, K., editor, *European Performance Engineering Workshop (EPEW)*, volume 4748 of *Lecture Notes in Computer Science*, pages 125–140. Springer.

[OGF, 2008] OGF (2008). Open Grid Forum. www.ogf.org.

[Othman et al., 2003] Othman, A., Dew, P., Djemamem, K., and Gourlay, I. (2003). Adaptive Grid Resource Brokering. In *Proceedings of the 2003 IEEE International Conference on Cluster Computing*, pages 172–179.

[Pacifici et al., 2005] Pacifici, G., Spreitzer, M., Tantawi, A., and Youssef, A. (2005). Performance Management of Cluster-Based Web Services. *IEEE Journal on Selected Areas in Communications*, 23(12):2333–2343.

[Padala et al., 2007] Padala, P., Shin, K., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A., and Salem, K. (2007). Adaptive Control of Virtualized Resources in Utility Computing Environments. In *Proceedings of the 2007 conference on EuroSys*.

[Parashar and Browne, 2005] Parashar, M. and Browne, J. (March 2005). Conceptual and implementation models for the grid. *Proceedings of the IEEE, Special Issue on Grid Computing, IEEE Press*, 93, 3:653–668.

[Parashar and Hariri, 2005] Parashar, M. and Hariri, S. (2005). Autonomic computing: An overview. *UPP 2004, Mont Saint-Michel, France, Springer Verlag*, 3566:247–259.

[Parashar and Lee, 2005] Parashar, M. and Lee, C. (March 2005). Grid computing - an evolving vision. *Proceedings of the IEEE, Special Issue on Grid Computing*, 93, 3:479–484.

[Pradhan et al., 2002] Pradhan, P., Tewari, R., Sahu, S., Chandra, A., and Shenoy, P. (2002). An observation-based approach towards self-managing web servers. *IWQoS 2002,Miami Beach, Florida, USA.*, pages 13–22.

[Quest, 2008] Quest (2008). Quest profiler. `http://www.quest.com`.

[Ranjan et al., 2002] Ranjan, S., Rolia, J., Fu, H., and Knightly, E. (2002). Qos-driven server migration for internet data centers. *IWQoS 2002, Miami Beach, Florida, USA.*, pages 3–12.

[Romberg, 2002] Romberg, M. (2002). The unicore grid infrastructure. `http://www.unicore.org`.

[Shivam et al., 2006] Shivam, P., Babu, S., and Chase, J. (2006). Learning Application Models for Utility Resource Planning. In *Proceedings of the 3rd International Conference on Autonomic Computing*.

[Sotomayor and Childers, 2005] Sotomayor, B. and Childers, L. (2005). *Globus Toolkit 4 : Programming Java Services*. Morgan Kaufmann.

[Stewart and Shen, 2005] Stewart, C. and Shen, K. (2005). Performance modeling and system management for multi-component online services. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 71–84, Berkeley, CA, USA. USENIX Association.

[Sun and Wu, 2003] Sun, X.-H. and Wu, M. (2003). Grid Harvest Service: A System for Long-Term, Application-Level Task Scheduling. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*.

[Sun Microsystems, 2005] Sun Microsystems (2005). HPROF Profiler Agent documentation. `http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html#hprof`.

[Tesauro et al., 2005] Tesauro, G., Das, R., Walsh, W., and Kephart, J. O. (2005). Utility-Function-Driven Resource Allocation in Autonomic Systems. In *Proceedings of the Second International Conference on Autonomic Computing*.

[Tesauro et al., 2006] Tesauro, G., Jong, N., Das, R., and Bennani, M.Ñ. (2006). A hybrid reinforcement learning approach to autonomic resource allocation. In *Proceedings of the 3rd International Conference on Autonomic Computing*.

[Uragonkar et al., 2005] Uragonkar, B., G.Pacifi, Shenoy, P., Spreitzer, M., and Tantawi, A. (2005). An analytical model for multi-tier internet services and its applications. *SIGMETRICS'05, Alberta, Canada*.

[Uragonkar and Shenoy, 2004] Uragonkar, B. and Shenoy, P. (2004). Cataclysm: Handling extreme overloads in internet services. *TR03-40, Department of Computer Science, University of Massachusetts, USA*.

[user manual, 2008] user manual, J. (2008). Sun microsystems - jvmpi. `http://java.sun.com/j2se/1.5.0/docs/guide/jvmpi/jvmpi.html`.

[Vargas, 2001] Vargas, A. (2001). The OMNeT++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'2001)*, Prague, Czech Republic. The Society for Modeling and Simulation International (SCS).

[Walsh et al., 2004] Walsh, W. E., Tesauro, G., Kephart, J. O., and Das, R. (2004). Utility Functions in Autonomic Systems. *ICAC: Proceedings of the IEEE International Conference on Autonomic Computing*, pages 70–77.

[WebPage, 2004] WebPage (2004). Adaptj/*j profiler. `http://www.sable.mcgill.ca/starj`.

[WebPage, 2006] WebPage (2006). Bea's jrocket 5.0. `http://www.bea.com`.

[WebPage, 2008] WebPage (2008). Mysql. http://www.mysql.com.

[Wilytech, 2006] Wilytech (2006). Wily technology. `http://www.wilytech.com`.

[Wolczko, 1999] Wolczko, M. (1999). Using a tracing javatm virtual machine to gather data on the behavior of java programs. *Technical Report SML 98-0154, Sun Microsystems Laboratories*.

[Xu et al., 2000] Xu, D., Nahrstedt, K., Viswanathan, A., and Wichadakul, D. (2000). QoS and Contention-Aware Multi-Resource Reservation. In *Proceedings of 9th IEEE International Symposium on High Performance Distributed Computing*.

[Xu et al., 2005] Xu, J., Oufimtsev, A., Woodside, M., and Murphy, L. (September, 2005). Performance modeling and prediction of enterprise javabeans with layered queuing network templates. *SAVCBS 2005, Lisbon, Portugal*.

[Xu et al., 2007] Xu, J., Zhao, M., Fortes, J., Carpenter, R., and Yousif, M. (2007). On the use of fuzzy modeling in virtualized data center management. In *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*, page 25, Washington, DC, USA. IEEE Computer Society.

[Yaghmour and Dagenais, 2000] Yaghmour, K. and Dagenais, M. R. (June 2000). Measuring and characterizing system behavior using kernel-level event logging. *Usenix Annual Technical Conference, San Diego, CA*.

Siglas y definiciones

ACID	Propiedades de una transacción en una base de datos transaccional
AM	Autonomic Manager
API	Application Programming Interface - conjunto de funciones y procedimientos disponibles para el programador
Arrival Rate	Número de llegadas por segundo
BSC-MF	Barcelona Supercomputing Center - Monitoring Framework
Burst	Peticiones simultáneas
ECM	eDragon CPU Manager
EJB	Enterprise Java Beans
E/S	Entrada - Salida - Típicamente tiempo en el que una aplicación no consume CPU y está esperando datos externos
FIFO	First In First Out, los elementos de la cola se sirven (totalmente) en el orden en el que llegan
GM	General Manager
HPC	High Performance Computing - computación de alto rendimiento
JIS	Java Instrumentation Suite, permite la instrumentación de aplicaciones JAVA
JNI	Java Native Implementation
JSP	Java Server Pages
JVMPI	Java Virtual Machine Profiler Interface
JVMTI	Java Virtual Machine Tool Interface
MR	(Recursos gestionados) Managed Resources
OGSI	Open Grid Services Infrastructure
Overhead	Sobrecarga, normalmente necesidad de proceso extra que requiere un trabajo

PN	(Red de Petri) Petri Net
PS	Processor Sharing, el tiempo de CPU se reparte entre los diferentes procesos utilizando quantums pequeños (ej. 10 ms.)
QN	(Red de Colas) Queue Network
QPN	(Red de Petri con colas) Queue Petri Net
QoS	(Calidad de servicio) Quality of Service
QoS Control	Control de la calidad del servicio
SD	(Tiempo de servicio) Service Demand
SDK	Software Developer Kit - conjunto de herramientas de desarrollo para el programador
SLA	(Contrado de nivel de servicio) Service Level Agreement
SMP	Symmetric Multi-Processing
SSL	Secure Sockets Layer, protocolo criptográfico en conexiones de red
Thinktime	tiempo de espera entre peticiones
Thread	Hilo de ejecución
Thread Pool	Conjunto de threads creados previamente para reducir el coste de creación
Timeout	Error que se produce cuando un cliente o un servidor no responde a una petición en un tiempo establecido
WSLA	Web Service Level Agreement