

# Computación distribuida sobre railes

Juan Julián Merelo y Juan Lupión

*Resumen*— En este trabajo se presenta una propuesta para usar Ruby on Rails en experimentos de computación masiva distribuida, específicamente en algoritmos evolutivos. El uso de las capacidades computacionales de los navegadores, y su comunicación con un servidor, y la capacidad de distribución de recursos de estos servidores, se combinan para crear una red heterogénea con distribución, en principio, de tipo estrella, que permite aprovechar capacidades no utilizadas de los ordenadores conectados en red. Realizaremos una serie de experimentos sobre una prueba de concepto que nos permitirá evaluar las posibilidades computacionales de una plataforma de este estilo, y medidas de prestaciones en diferentes configuraciones y diferentes navegadores.

*Palabras clave*— Computación distribuida, Ruby, algoritmos evolutivos, AJAX, JavaScript

## I. INTRODUCCIÓN

Las redes a nivel de aplicación ponen recursos computacionales *sobrantes* o *ad hoc* a disposición de experimentos de computación distribuida. Uno de estos recursos son los navegadores, que poseen una cierta capacidad computacional nativa a través de su intérprete del lenguaje de programación JavaScript[1] y la comunicación asíncrona cliente-servidor usando JavaScript y XML, que se denomina AJAX [2]. Por otro lado, Ruby on Rails [3], [4], [5] es un entorno que permite distribuir aplicaciones entre un cliente que use JavaScript y un servidor web.

Las redes a nivel de aplicación, en general, permiten que un cliente, descargándose una aplicación, pueda unirse a una red de computación distribuida y aportar ciclos de CPU a los experimentos que se están ejecutando en la misma. Algunas redes como SETI@Home han tenido un gran éxito, mientras que otros experimentos comerciales, tales como Popular Power, han experimentado un sonado fracaso.

La clave de las redes a nivel de aplicación debe ser la facilidad de uso: idealmente, el usuario no debería descargarse ninguna aplicación para conectarse a las mismas. Por eso, deben basarse en elementos que el usuario ya posea, como, por ejemplo, el navegador. Todos los ordenadores hoy en día, incluso los teléfonos móviles, incluyen un navegador, y en la mayoría de los casos, los navegadores incluyen JavaScript [6], [7] un lenguaje interpretado, propuesto inicialmente por Netscape, pero adoptado como un estándar ECMA [8], [9], [10], [11], lo que hace que la mayoría de los navegadores sean compatibles, al menos a nivel de lenguaje (no necesariamente a nivel de los objetos del navegador manejables desde el mismo, donde existe un nivel razonable de compatibilidad, sin embargo). En principio, el uso de las capacida-

des de este intérprete para computación distribuida surge de la introducción, alrededor del año 2000, del objeto XMLHttpRequest, que permite hacer peticiones asíncronas al servidor, lo que transforma un esquema simple cliente/servidor (el navegador/cliente solicita una página, el servidor la sirve) en algo más paritario, en el que se abre una línea de comunicación bidireccional entre cliente y servidor: el navegador puede hacer llamadas al servidor, realizar cualquier tipo de computación, y colocar el resultado una vez más en el mismo. Esto abre la puerta al uso del navegador en redes a nivel de aplicación, y a su uso como sistema de computación distribuida.

En principio, esta capacidad se puede controlar, desde el servidor, con cualquier lenguaje de programación, incluso combinarse con entornos de programación distribuida basados en OpenGrid. Sin embargo, para permitir un desarrollo ágil de la aplicación, se propone Ruby on Rails, un entorno basado en el lenguaje Ruby y en el paradigma Model/View/Controller, donde se separan claramente el modelo de datos que se usa (y que está respaldado por un sistema de gestión de bases de datos) de las diferentes *vistas* del mismo (plantillas HTML que el servidor llenará con los datos del modelo, para servirlos al cliente), y de su control, las funciones que sirven para alterar y manejar los datos; en RoR, los controladores son los que reciben las peticiones del cliente, y reaccionan con respecto a ellas. Crear una aplicación distribuida usando RoR implica, pues, crear un modelo de datos relacionado con la aplicación, un controlador que responda a las peticiones del cliente y distribuya la computación entre cliente y servidor, y unas vistas que incluirán la computación que se haga en el cliente (puesto que los programas en JavaScript forman parte de las páginas web que se envían al mismo).

Por otro lado, el esquema AJAX es de comunicaciones asíncronas. El cliente genera una petición, que envía al servidor, y junto con la petición adjunta una función que será la que se active cuando el servidor envíe su respuesta. De esta forma, el cliente no se bloquea, y puede haber diferentes hebras activándose cuando se reciban diferentes respuestas; pero en todo caso, lo que nos interesa, es que, al ser la comunicación asíncrona, se puede generar un ciclo de petición-respuesta que no necesita de la intervención del usuario.

En nuestro caso, nos vamos a concentrar en distribuir aplicaciones de computación evolutiva. La computación evolutiva, que ha sido adaptada a diferentes paradigmas de computación paralela y distribuida por nuestro grupo (tales como Jini [12], MPI [13], [14], arquitecturas orientadas a servicio [15] y P2P [16], [17]) se presta a este tipo de ejercicio por

jj@merelo.net, Departamento de Arquitectura y Tecnología de Computadores, ETS Ingeniería Informática y Telecomunicaciones, C/Daniel Saucedo Aranda, s/n 18071 Granada

pantulis@gmail.com, creador y editor de la página web <http://www.sobrerailes.com>

varias razones: al ser métodos basados en una población de soluciones, el cálculo se puede distribuir de muchas formas diferentes; por otro lado, algunos trabajos afirman que hay una sinergia entre el algoritmo evolutivo y la paralelización: las poblaciones aisladas y conectadas esporádicamente evitan la pérdida de diversidad, y, por tanto, hacen que se llegue a mejores soluciones antes, dando lugar en algunos casos a aceleraciones superlineales [18]. Por tanto, son una familia de algoritmos aceptable para una prueba de concepto, tal como la que se trata de presentar en este trabajo.

En este trabajo, por tanto, presentaremos una prueba de concepto de computación distribuida usando Ruby on Rails. Para ello, se implementará un algoritmo evolutivo simple sobre este entorno, y se explicarán las decisiones de diseño que ha habido que tomar para sacar el máximo partido al mismo. Se llevará a cabo una prueba de concepto, que demuestre la posibilidad de hacer aplicaciones distribuidas usando ese sistema. Asimismo, dado que el lenguaje JavaScript es multiplataforma, se evaluarán qué navegadores ofrecen mejores prestaciones para este tipo de aplicaciones.

El resto del trabajo se organiza de la forma siguiente: a continuación, un breve estado del arte sobre computación distribuida *espontánea* o *ad-hoc*; a continuación se expone el entorno computacional usado en este trabajo en la sección III. Los resultados se presentarán en la sección IV, y finalmente, las conclusiones y el trabajo futuro se presentarán en la sección V

## II. ESTADO DEL ARTE

La denominada *computación voluntaria* (*volunteer computing*) [19], [20], [21] consiste en la creación de una infraestructura para que diferentes personas distribuidas por Internet donen ciclos de CPU para un esfuerzo de computación común. El caso más conocido es el de SETI@home<sup>1</sup>, que consiste desde el punto de vista del usuario en un salvapantallas que tiene que descargarse, y que realiza diferentes operaciones de análisis de señal. Las empresas relacionadas con la computación voluntaria, tales como Popular Power (y otras similares; se mencionan, por ejemplo, en [22] experimentaron con clientes basados en Java, pero comercialmente no ha sobrevivido ninguna.

Hay dos problemas en este tipo de redes: no exceder los recursos de CPU que el voluntario quiere donar, y conseguir un número masivo de los mismos, para ofrecer una cantidad de recursos que sea interesante desde el punto de vista experimental y computacional; esto, a su vez, ofrece problemas de escalado. Pero, en todo caso, la forma más fácil de obtener millones de usuarios es usar infraestructura que ya esté presente en cualquier instalación, tal como el navegador. De hecho, ya ha habido sugerencias en ese sentido (como la de Jim Culbert en su weblog [23]), y alguna en listas de correo, pero ningún intento serio

<sup>1</sup>Ver <http://setiathome.berkeley.edu/> para descargar el software y diferentes informes sobre el tema

de aprovechar esta posibilidad.

El problema con el uso de JavaScript es que es un lenguaje interpretado, con instalaciones que varían en eficiencia, y que no está precisamente optimizado para el cálculo numérico, sino más bien para el manejo de árboles de objetos (el denominado DOM, document object model) y de cadenas. Sin embargo, el hecho de que haya tantas posibles clientes usables hace que sea interesante considerarlo, al menos como posibilidad. El hecho de que también se pueda usar de forma totalmente espontánea, sin ninguna intervención consciente del usuario (por ejemplo, al cargar una página web), hace que ofrezca muchísimas posibilidades, (a la vez que peligros, claro está).

En este trabajo exponemos una prueba de concepto de un entorno de desarrollo de aplicaciones de computación distribuida basado en Ruby on Rails, que se aprovecha precisamente de esta capacidad; y que, en ese sentido, es totalmente novedoso. Lo veremos a continuación.

## III. MATERIAL Y MÉTODOS

El experimento necesita de un navegador que ejecute JavaScript, en nuestro caso Firefox, corriendo sobre el sistema operativo Linux, y un servidor con Ruby on Rails. RoR incluye su propio servidor web, WEBrick; en realidad, no es el más adecuado para altas prestaciones, pero se trata de usarlo simplemente en esta prueba de concepto, para más adelante escoger la configuración más adecuada a la computación de altas prestaciones.

La aplicación, siguiendo el modelo MVC, se organiza en un modelo, una vista y los controladores. Nos fijaremos especialmente en el primero y el último.

El **modelo** es una tabla en la base de datos para representar la población, que podría ser la siguiente:

```
create table guy {
  cromosoma varchar(256),
  fitness float
};
```

Esta tabla almacenará la población. En principio hemos optado por una representación tradicional usando una cadena binaria; el cromosoma será simplemente una lista de 0s y 1s. También usamos un fitness escalar, con lo que no se podrán realizar aplicaciones de optimización multiobjetivo. En todo caso, el modelo de datos estará relacionado con la aplicación a optimizar, y diferentes representaciones cromosómicas y tipos de fitness necesitarían de un modelo de datos diferente.

En cuanto al **controlador**, necesitaremos controles para solicitar elementos de la población, y volver a colocarlos, quizás evaluados, en la misma. La elección de controlador estará relacionada con la división de la computación entre el cliente y el servidor. Teniendo en cuenta que el algoritmo evolutivo funciona de la forma siguiente:

1. Creación de una población inicial aleatoria
2. Repetir hasta que se alcance una solución o un número determinado de generaciones

- a) Evaluar los elementos de la población, asignándoles un fitness
- b) Elegir los miembros de la población que serán alterados para dar lugar a la siguiente generación
- c) Mutar (cambiar) y entrecruzar (intercambiar parte del cromosoma) entre esos miembros elegidos de la población
- d) Mezclar la nueva población con la antigua, eliminando, en general, a los que tengan un fitness menor.

Hay, por tanto, diferentes acciones: evaluación, aplicación de operadores genéticos (mutación y entrecruzamiento), y mezclado de la población. El planteamiento más simple de distribución de las tareas es repartir la evaluación del fitness de cada uno de los elementos de la población, que suele ser el paso que necesite más recursos computacionales, en los clientes, y realizar el resto de los pasos en el servidor. Técnicamente, el paso de evaluación se incluiría en la *vista*, puesto que se interpreta en el cliente; en la práctica, será un programa en JavaScript que formará parte de la plantillas almacenadas en el directorio correspondiente de la aplicación RoR. Evidentemente, el hecho de que se ejecute en el cliente tiene implicaciones de autenticación y seguridad que hay que tener en cuenta (incluyendo, posiblemente, fraude, tal como se indica en [19]), pero que no vienen al caso. En principio, supongamos que se usa autenticación por IP de cliente, o, simplemente, que hay control de algún modo de los clientes para que no envíen, por ejemplo, un fitness más alto que el calculado realmente, falseando los resultados.

Habrà que escribir controles para la generación aleatoria y para los operadores genéticos. El esquema puede proceder de la forma siguiente, desde el punto de vista del cliente:

1. Carga de la aplicación-cliente, lo que se llevará a cabo al solicitar una página web del servidor. La página web incluye el código cliente, que comenzará a ejecutarse al cargarse la página web (usando el evento `onLoad` del navegador) o bien a instancias del usuario.
2. Petición de individuos para evaluar al servidor por parte del cliente. El servidor envía un número de individuos relacionado con el número de clientes que haya conectados en cada momento. En estas pruebas iniciales, se usará un solo cliente y servidor, al que se le enviará toda la población para evaluar. Esta petición la recibirá el control `population` del servidor.
3. El cliente evalúa los diferentes individuos, y envía el resultado al servidor, donde será recibido por el control `populationReady`. Se pueden usar diferentes formatos para intercambiar información entre cliente y servidor. Tratándose de AJAX, la elección obvia es XML, pero en nuestro caso hemos elegido JSON (*JavaScript Object Notation*), un tipo de datos con el que JavaScript trabaja de forma nativa, y servidor

RoR puede generar fácilmente.

4. El servidor usa el método de torneo para crear nuevos individuos. En este método se escoge un número  $n$  de individuos, se eliminan los peores  $p < n$ , que se sustituyen por los descendientes del resto de los individuos; también se puede hacer escogiendo a los mejores hasta que se forme una nueva población del tamaño requerido (habitualmente el mismo tamaño que la original). Este torneo se puede hacer en varias fases: realizar una serie de torneos aleatoria, y marcar a los peores, que serán eliminados; y tomar los no marcados, para que se reproduzcan entre si, o bien hacerlo sobre la marcha. El algoritmo termina cuando se ha evaluado un número determinado de individuos, o bien cuando se ha alcanzado un nivel de fitness establecido de antemano. Usando este método, un porcentaje de los nuevos individuos se generan mediante cada uno de los operadores genéticos.
5. La respuesta se envía al cliente, donde se genera un *callback* que hace que se vuelva al primer paso.
6. El algoritmo termina a instancias del cliente, cuando alcanza una condición determinada; por ejemplo, haber alcanzado un fitness máximo predeterminado. En ese momento, deja de solicitar nuevos individuos para evaluar al servidor.

El tener la población del algoritmo evolutivo en la base de datos tiene ventajas adicionales: podría actuar como caché, de forma que no es necesario evaluar un individuo varias veces, y se puede usar como un mecanismo de prevención de duplicados, para que, en caso de que se quiera hacer así, no se generen individuos ya evaluados y se proceda de forma sistemática a la exploración del espacio de búsqueda. A continuación exponemos los experimentos que se han llevado a cabo y sus resultados

#### IV. RESULTADOS

Para llevar a cabo un experimento que pruebe el funcionamiento del sistema, y dé una prueba de su eficiencia, se ha creado una instalación simple cliente-servidor; el servidor aloja RoR, mientras que el cliente usa diferentes navegadores, y puede estar alojado en el mismo ordenador que el servidor o en otro ordenador diferente. Se usará una función relativamente simple para evaluar el algoritmo evolutivo: la función Royal Road [24]. Esta función cuenta el número de unos de una cadena binaria por bloques: no se suma el tamaño del bloque hasta que está completo. En nuestro caso, usaremos un tamaño de bloque igual a 4 y un tamaño de cadena igual a 64. De la misma forma, usaremos una población de 256 individuos, generados aleatoriamente con una probabilidad del 75 % de que cada bit sea uno. Esto crea una distribución ligeramente sesgada, que hace que el algoritmo tarde menos tiempo en alcanzar una solución, y que la dispersión en el tiempo de la duración sea también menor. Se usará una tasa de mutación igual a la tasa de crossover, e igual al 50 % (es decir, la mitad de

los individuos nuevos se genera usando cada uno de los operadores). La mutación cambia un solo bit en el cromosoma, mientras que el crossover es el clásico crossover de dos puntos. La selección se hace usando un torneo de 2, en el que se escogen aleatoriamente parejas de cromosomas, y se toma entre ellos el de mayor fitness, eliminando el de menor fitness. Por tanto, se trata de un algoritmo evolutivo de estado estacionario en el que, en cada ejecución, permanecen los 128 cromosomas que han resultado ganadores en los torneos, y se usan para generar, mediante mutación y entrecruzamiento (crossover), el resto de la población.

Los tiempos se tomarán sobre el registro (*log*) del propio servidor web incluido en RoR; aunque las condiciones de ejecución de las diferentes pruebas es aproximadamente igual, no se asegura que la carga de trabajo es exactamente la misma (ni en cliente, ni en servidor, ni en la red). Para tratar de eliminar variabilidad de la carga de trabajo, se repetirá varias veces cada una de las ejecuciones. Como marca de tiempo, y para asegurar las mismas condiciones iniciales, se reiniciará RoR cada vez que se haga una ejecución.

El servidor es:

```
Linux localhost.localdomain 2.6.16-1.2111_FC5
#1 SMP Thu May 4 21:16:04 EDT 2006
x86_64 x86_64 x86_64 GNU/Linux
```

con un procesador AMD Athlon(tm) 64 X2 Dual Core Processor 4200+, mientras que como cliente se usa el mismo ordenador que contiene el servidor, y un portátil Sony VAIO VGN-S4XP con un Intel Pentium M a 2 GHz, con Windows XP, unido a través de un router Thompson SpeedTouch en una conexión Ethernet a 100 MBits/s. En principio, cabría esperar menos prestaciones de la combinación cliente/servidor que del cliente ejecutándose en el mismo servidor.

Probaremos además diferentes navegadores: Firefox, Opera y Konqueror (ejecutándose en el cliente/servidor Linux), y Firefox, Bon Echo (Firefox 2.0) e Internet Explorer (ejecutándose en el cliente Windows). Para cada una de los clientes hemos hecho un número variable de ejecuciones, entre 5 y 10. El objetivo es, por un lado, examinar la eficiencia de las diferentes disposiciones, y, por otro lado, examinar la eficiencia (e, incidentalmente, la compatibilidad) del intérprete de JavaScript de los diferentes navegadores.

¿Hay, pues, una diferencia entre ejecutar el cliente y el servidor en el mismo ordenador y ejecutarlo en ordenadores diferentes? Para ello, agrupamos todas las ejecuciones hechas en el ordenador que también corre el servidor (*fc5*, de Fedora Core 5) y en un ordenador diferente, ejecutando Windows XP (*wxp*).

Inicialmente, agruparemos las ejecuciones por configuración: un ordenador (*fc5*) y dos ordenadores (*wxp*), incluyendo todos los navegadores, con el objeto de comprobar si hay alguna diferencia notable. Como se observa en la figura 1, el ejecutar el cliente

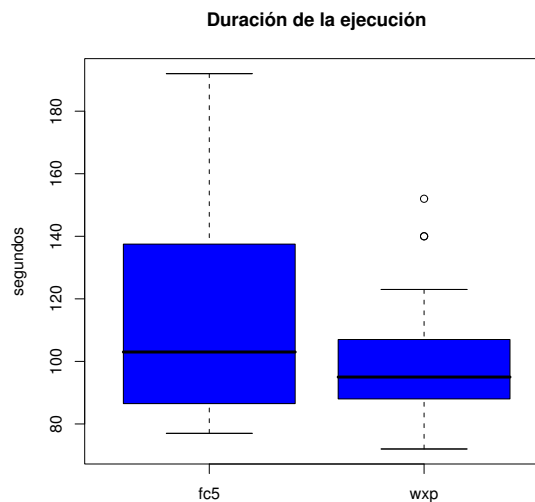


Fig. 1. Boxplot de tiempos por S.O.

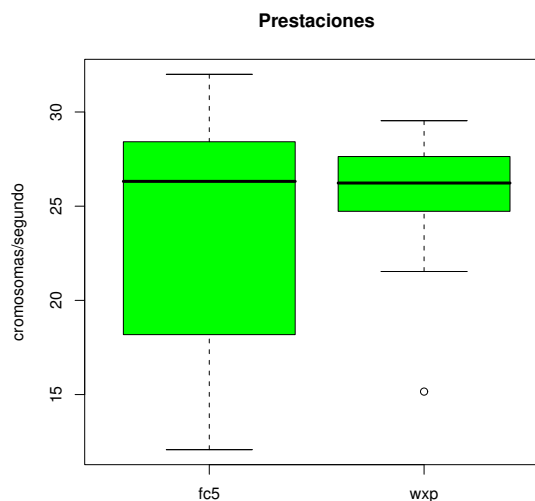


Fig. 2. Boxplot de esfuerzo por S. O.

y el servidor en el mismo ordenador (etiqueta *fc5*) es, en media, menos eficiente que ejecutarlo en dos ordenadores diferentes (*wxp*), a pesar de ser el ordenador cliente, en este caso, menos potente que el servidor (un AMD de núcleo dual 4100+ vs. un Intel Pentium M 2000). Sin embargo, examinando el número de cromosomas evaluado en media por cada configuración, tal como hacemos en la figura 2, se ve que el número de cromosomas medio evaluado en cada uno de las configuraciones es, en media, prácticamente indistinguible, aunque la desviación estándar sea mayor en el caso de la configuración mono-ordenador (*fc5*). Ello viene a indicar que el mayor esfuerzo computacional llevado a cabo en el caso mono-ordenador se compensa por la diferencia de velocidad y los costes de comunicación empleados en dos ordenadores, de forma que, a pesar de tener el ordenador *wxp* menos prestaciones, el número de individuos evaluados vie-

ne a ser el mismo.

Dentro de cada sistema operativo, se pueden examinar también las diferentes prestaciones de cada uno de los navegadores, mirando al número de cromosomas evaluados por segundo por cada navegador. Eso nos dará una idea de las prestaciones esperables en estos casos, así como de qué navegadores contienen una implementación del intérprete de JavaScript más eficiente (al menos en lo relativo a la aplicación probada).

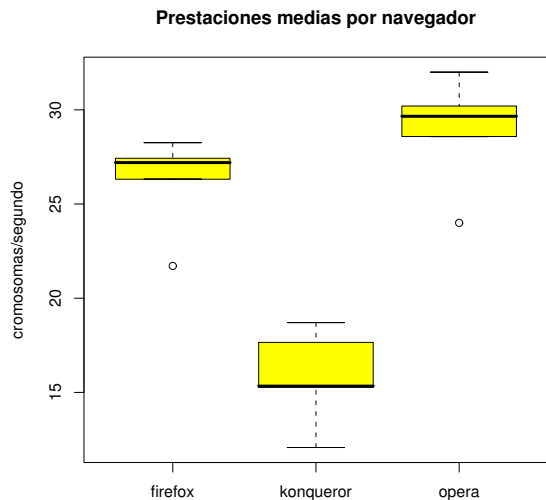


Fig. 3. Número medio de cromosomas por segundo (Linux).

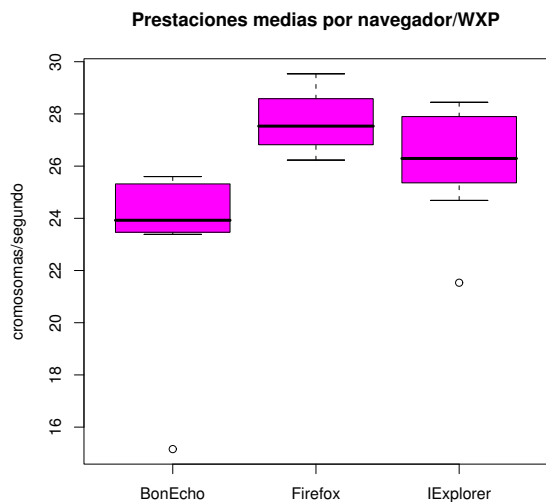


Fig. 4. WinXP: Número medio de cromosomas por segundo.

En la figura 3 se comparan los diferentes navegadores de la plataforma Fedora Core 5; cabe esperar que sus prestaciones sean independientes del hecho de que se ejecutan en el mismo ordenador que el servidor (aunque posiblemente en otro de los núcleos del procesador), y que se deben, principalmente, a diferencias de arquitectura software de los mismos.

En este caso, se aprecian pequeñas diferencias entre Opera y Firefox, pero hay una diferencia abismal con respecto a navegador del entorno KDE, Konqueror. Cualquiera de los dos primeros sería una buena elección para experimentos de computación distribuida (siempre que los resultados de este experimento se puedan generalizar, lo que en el caso de programas de tratamiento de cadenas será posiblemente cierto).

En el caso de Windows XP, las prestaciones para diferentes navegadores se muestran en la figura 4. Las diferencias son menores, aunque significativas, dando Firefox, las mejores prestaciones, seguidas por el Internet Explorer, para terminar con Bon Echo, la versión alfa del Firefox 2.0; estas prestaciones se deberán, posiblemente, a que no se trata de una versión definitiva.

Mirando además al navegador Firefox en ambas plataformas, se ve que las prestaciones son similares; pero sin tener en cuenta el sistema operativo, el navegador que arroja unas mayores prestaciones es el Opera. Por otro lado, si eliminamos el navegador menos eficiente, el Konqueror, en la comparación hecha entre sistemas operativos, el resultado obtenido sería el esperado: una mayor eficiencia de la configuración mono-ordenador (por el uso eficiente de los dos núcleos del procesador).

Finalmente, dado que el objetivo principal de este trabajo era ofrecer una prueba de concepto sobre el uso de sistemas que se encuentren en cada instalación de un ordenador (el navegador) y un sistema de desarrollo rápido como Ruby on Rails, consideramos que el objetivo se ha alcanzado, mostrando que el uso de una configuración cliente-servidor permite una cierta mejora en prestaciones (que en realidad, dado que se trata de ordenadores de potencia muy diferente, se plasma en una diferencia prácticamente nula) con respecto a la ejecución en un solo ordenador, y que cabría esperar que en una configuración para varios clientes, que se ejecutaran de forma concurrente, se daría un escalado, al menos hasta saturar el canal de comunicación del servidor.

## V. CONCLUSIONES Y TRABAJO FUTURO

Como se ha indicado en la introducción y en el resumen, en este trabajo se presenta simplemente una prueba de concepto, que muestra la posibilidad de trabajar con una aplicación de computación distribuida usando un entorno de desarrollo ágil tal como Ruby on Rails y unas herramientas presentes en cada ordenador conectado a la red: un navegador que use JavaScript.

Los experimentos realizados prueban que se puede desarrollar rápidamente una aplicación distribuida usando RubyOnRails; que la infraestructura computacional subyacente es eficiente en el procesamiento de cromosomas (aunque hay diferencias entre navegadores), y que, en principio, se podría basar un sistema de computación distribuida voluntaria en Ruby on Rails.

Evidentemente, el paso siguiente es la creación de un sistema que permita usar más de un cliente, y

que ofrezca mecanismos de bloqueo sobre los cromosomas que han sido enviados a evaluar, así como un cierto modo de controlar la carga que se envían a los diferentes clientes y ajustarla a su potencia computacional.

Como trabajo futuro, queda la creación de una aplicación RoR que sea fácilmente instalable en cualquier ordenador que pueda ejecutar un servidor web y el lenguaje Ruby, así como la creación de un entorno de programación distribuida, donde, por ejemplo, se puedan hacer lo siguiente:

- Elegir entre diferentes aplicaciones a realizar, en función de los gustos personales o del tiempo disponible.
- Evaluación y predicción de prestaciones, para poder estimar de antemano el tiempo necesario para ejecutar una aplicación, los recursos necesarios.
- Equilibrado de carga entre diferentes clientes, en función de sus prestaciones y de los recursos que presten
- Conexión entre diferentes instalaciones RoR, que permitan llevar a cabo aplicaciones más complejas, y que permitan crear un sistema masivamente paralelo.
- Resolver problemas de autenticación e identificación ante el usuario, para poder llevar a cabo aplicaciones seguras.

#### RECONOCIMIENTOS

Este artículo ha sido financiado en parte por el proyecto NADEWeb: Nuevos Algoritmos Distribuidos Evolutivos en la web, proyecto CICYT código TIC2003-09481-C04.

#### REFERENCIAS

- [1] Gilorien. *DHTML and JavaScript*. Prentice-Hall PTR, Upper Saddle River, NJ 07458, USA, 2000.
- [2] Wikipedia. Ajax — wikipedia, la enciclopedia libre, 2006. [Internet; descargado 14-mayo-2006].
- [3] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby: The Pragmatic Programmers' Guide, Second Edition*. Pragmatic Bookshelf, October 2004.
- [4] Reuven M. Lerner. At the forge: assessing ruby on rails. *Linux J.*, 2006(142), February 2006.
- [5] Dave Thomas, David Hansson, Leon Breedt, Mike Clark, Thomas Fuchs, and Andrea Schwarz. *Agile Web Development with Rails (The Facets of Ruby Series)*. Pragmatic Bookshelf, July 2005.
- [6] Rawn Shah. A beginner's guide to JavaScript. *JavaWorld: IDG's magazine for the Java community*, 1(1):??-??, March 1996.
- [7] David Flanagan. *JavaScript Pocket Reference (2nd Edition)*. O'Reilly, October 2002.
- [8] ECMA. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, December 1999.
- [9] ECMA. *ECMA-290: ECMAScript Components Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, June 1999.
- [10] ECMA. *ECMA-327: ECMAScript 3: Compact Profile*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, June 2001.
- [11] ECMA. *ECMA-357: ECMAScript for XML (E4X) Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, June 2004.

- [12] M. García-Arenas; J. G. Castellano; P. A. Castillo; J. Carpio; M. Cillero; Juan-Julián Merelo-Guervós; A. Prieto; V. Rivas; G. Romero. Speedup measurements for a distributed evolutionary algorithm that uses jini. In Universidad de Granada Depto. ATC, editor, *XI Jornadas de Paralelismo*, pages 241–246, 2000.
- [13] J. G. Castellano; M. García-Arenas; P. A. Castillo; J. Carpio; M. Cillero; Juan-Julián Merelo-Guervós; A. Prieto; V. Rivas; G. Romero. Objetos evolutivos paralelos. In Universidad de Granada Depto. ATC, editor, *XI Jornadas de Paralelismo*, pages 247–252, 2000.
- [14] J.G. Castellano, P.A. Castillo, Juan-Julián Merelo-Guervós, and G. Romero. Paralelización de evolving objects library usando MPI. In *Actas Jornadas de Paralelismo* [25].
- [15] Juan-Julián Merelo-Guervós, J.G. Castellano, P.A. Castillo, and G. Romero. Algoritmos genéticos distribuidos usando SOAP. In *Actas Jornadas de Paralelismo* [25].
- [16] M.G. Arenas, L. Foucart, Juan-Julián Merelo-Guervós, and P. A. Castillo. JEO: a framework for Evolving Objects in Java. In *Actas Jornadas de Paralelismo* [25].
- [17] Víctor Martín Molina; Juan Julián Merelo Guervós; Juan Luis Jiménez Laredo; Maribel García Arenas. Algoritmos evolutivos en Java: resolución del TSP usando DREAM. In *Actas XVI Jornadas de Paralelismo, incluido en CEDI'2005. Granada, septiembre 2005*, pages 667–683. Thomson, Septiembre 2005.
- [18] Enrique Alba. ¿Puede un algoritmo evolutivo paralelo proporcionar ganancia superlineal?, 1999. *Actas CAEPIA-TTIA'99 Vol. 2*, pp. 89–97.
- [19] L. F. G. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. *Future Generation Computer Systems*, 18(4):561–572, 2002.
- [20] L. F. G. Sarmenta and Satoshi Hirano. Bayanihan: building and studying Web-based volunteer computing systems using Java. *Future Generation Computer Systems*, 15(5-6):675–686, 1999.
- [21] David P. Anderson, Eric Korpela, and Rom Walton. High-performance task distribution for volunteer computing. In *E-SCIENCE '05: Proceedings of the First International Conference on e-Science and Grid Computing*, pages 196–203, Washington, DC, USA, 2005. IEEE Computer Society.
- [22] Peter Cappello and Dimitros Mourtoukos. A scalable, robust network for parallel computing. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 78–86, New York, NY, USA, 2001. ACM Press.
- [23] Jim Culbert. Ajax and distributed computation thoughts. Published at <http://culbert.net/?p=6>, March 2006. Último acceso Mayo 2006.
- [24] Janet Wiles and Bradley Tonkes. Mapping the royal road and other hierarchical functions. *Evol. Comput.*, 11(2):129–149, 2003.
- [25] *Actas XII Jornadas de Paralelismo*. Universidad Politécnica de Valencia, 2001.