RoboDist: Una Plataforma de juego P2P para incentivar la programación en los grados de Ingeniería

J. Lladós[1], I. Arroyo[1], F. Cores[1], JL. Lérida[1], F. Giné[1]

Resumen—**La dificultad en los procesos de abstracción tan necesarios en las asignaturas de Ingeniería y en especial aquellas con contenidos de programación, dificulta el aprendizaje de sus contenidos y produce falta de motivación en los alumnos. La programación de comportamiento inteligente en robots, videojuegos, etc. se ha aplicado con éxito en los currículos de informática en los últimos años. En este artículo se presenta una plataforma de juego en red, denominada *RoboDist*, que trata de facilitar y motivar al alumno de Ingeniería en el aprendizaje de la programación. Se describen los principales objetivos de diseño, así como su arquitectura y finalmente se evalúan los parámetros de rendimiento de la misma.**

Palabras clave—**Peer-to-Peer, Programación, Juegos en Red.**

I. Introducción

El Espacio Europeo de Educación Superior (EEES) presenta un nuevo marco formativo centrado en el aprendizaje autónomo del alumno y el desarrollo de competencias. Este nuevo escenario requiere la implantación de nuevas metodologías de aprendizaje que permitan al alumno la integración natural de los conocimientos impartidos mediante la resolución de casos prácticos, el trabajo en equipo, la colaboración y cooperación en la búsqueda de soluciones, incorporando además mecanismos de autoevaluación que lo hagan partícipe y consciente de su propio proceso de aprendizaje.

En los últimos años se han implantado distintos casos de éxito que promueven, mediante el trabajo en equipo, la búsqueda de soluciones a un reto de Ingeniería y su puesta en común mediante entornos competitivos y colaborativos, como son los torneos de robótica [1][2], la programación de videojuegos [3] o la programación de *bots* (tanques programables) en entornos de competición virtuales como *Robocode* u otros [4][5].

En el presente trabajo presentamos una plataforma de juego en red, denominada *RoboDist*, que nos permite la creación de nuevos retos de programación, así como la colaboración y cooperación en la búsqueda de nuevas soluciones y su puesta en práctica. En nuestro caso presentamos nuestra experiencia en la implantación de un modelo de aprendizaje basado en proyectos aplicado a la programación de *bots*, utilizando el entorno *Robocode*.

RoboDist ha sido implementada sobre un entorno de cómputo P2P, denominado *CoDiP2P* [6], que nos permite la interconexión de los equipos de trabajo a nivel global a través de Internet.

El presente artículo se estructura en los siguientes apartados. Los conceptos previos son descritos en la Sección II, el diseño y arquitectura de la plataforma *RoboDist* se explica en la Sección III, en la Sección IV se muestra la experimentación realizada y finalmente se describen las principales conclusiones del trabajo.

II. Conceptos previos

En esta Sección se describen las dos plataformas que integran *RoboDist*. *Robocode* es un entorno educativo de programación de *bots* y *CoDiP2P* es una plataforma P2P de cómputo distribuido.

A. Robocode

Robocode es un juego de programación que tiene por objetivo principal despertar el interés de estudiantes y del público en general por la programación, la informática y los estudios de Ingeniería en general [6].

Este objetivo lo logra *Robocode* mediante un juego que simula combates entre *bots* virtuales, de modo que el jugador controla el comportamiento de su *bot* mediante un pequeño programa java. Este programa aplica algoritmos de diferentes campos (física, programación estructurada, inteligencia artificial, etc.) con el objeto de detectar y disparar a los objetivos enemigos, así como realizar las acciones necesarias para poder esquivar/escapar de los ataques enemigos.

En *Robocode*, los *bots* cuentan con una cuota de energía limitada (que puede utilizar o no) y un radar para detectar al enemigo. Dicha energía se reduce cada vez que se dispara, cuando se recibe un disparo o un golpe. Además permite ser ejecutado sin utilizar entono gráfico, con una gran variedad de comandos y parámetros, así como almacenar y reproducir con posterioridad las partidas.

Fig. 1. Ejecución de *Robocode*.

[1] Dpto. de Informática e Ingeniería Industrial, Universitat de Lleida. e-mail:{jordi.llados, ismael.arroyo}@udl.cat, {fcores, jlerida, sisco} @diei.udl.cat

La Fig. 1 muestra un ejemplo de una batalla *Robocode* en donde intervienen ocho *bots*, con su correspondiente campo de visión y los disparos de los mismos.

B. *CoDiP2P*

CoDiP2P [6] es una arquitectura P2P de cómputo distribuido implementada por el grupo GCD (Grupo de Computación Distribuida) de la Universidad de Lleida, junto con la empresa Indra.

La topología de este entorno es jerárquica, en forma de árbol *B-ario*, donde cada nodo tiene *B* hijos. En la Fig. 2 se visualiza la topología de *CoDiP2P*. Cabe destacar que su topología se divide en niveles y a su vez, cada nivel en áreas, que son agrupaciones lógicas de peers.

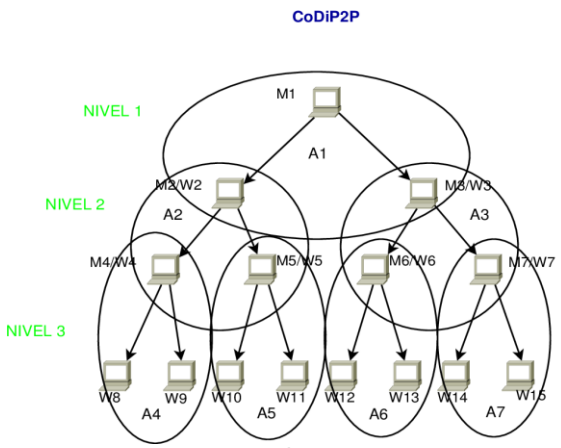


Fig. 2. Topología de *CoDiP2P*.

Cada peer puede realizar tres tipos de roles: *Manager*, *Worker y Master*. El rol *Manager* controla y gestiona a los peers conectados inmediatamente en el nivel inferior mediante un subárbol. El rol de *Worker* es el encargado de ejecutar tareas y enviar información sobre el estado de sus recursos a su Manager. Se denomina *Master* al peer que lanza una aplicación al sistema y que se encarga de gestionar su ejecución.

Un *área* es un espacio lógico formado por un *Manager* más sus *Workers*. Un área puede ejecutar localmente aplicaciones intensivas de cómputo y si es necesario distribuir las tareas que no puede asumir entre las áreas vecinas.

Las funcionalidades básicas de *CoDiP2P* [6] son inserción, mantenimiento, salida, planificación, búsqueda de recursos y ejecución de peers.

Los principales objetivos de la plataforma *CoDiP2P* son:

- Escalabilidad y heterogeneidad. Permite la inclusión de un número indefinido de peers sin tener en cuenta su arquitectura.
- Gestión distribuida. Carece de entidades centrales encargadas de la gestión. Los roles son asignados de forma dinámica.
- Tolerancia a fallos. La gran volatilidad de los peers implica que se tenga que reasignar de forma dinámica las funciones del sistema a los peers para asegurar robustez.

Cabe destacar que la manera de usar la plataforma es completamente transparente al usuario, solo hay que enviar contra *CoDiP2P* los cálculos que se quieran realizar. La plataforma automáticamente mandará a una serie de *Workers* que procesen estas tareas y cuando se hayan completado, la plataforma devolverá el resultado al máster solicitante. En la Fig. 3 se muestra, a modo de ejemplo una visualización de la interface gráfica de la plataforma CoDiP2P.

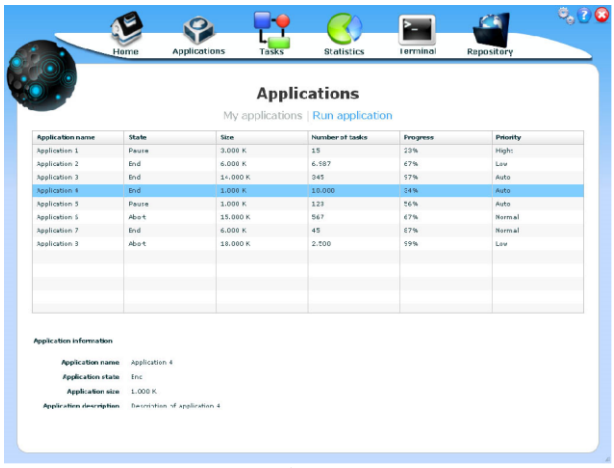


Fig. 3. Interfaz de *CoDiP2P*.

III. ROBODIST

En esta sección presentaremos la plataforma *RoboDist*. Como se puede observar en la Fig. 4, esta plataforma está formada por tres componentes: la lógica del juego, *CoDiP2P* y *Robocode*.

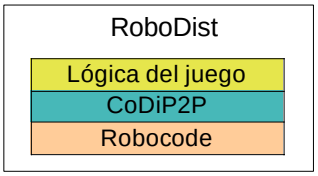


Fig. 4. Estructura de la plataforma *RoboDist*.

La finalidad de esta estructura es poder gestionar un juego distribuido a través de internet, ofreciendo la posibilidad de entablar múltiples partidas con otros jugadores mediante distintas instancias del juego de forma distribuida. Se aprovecha la capacidad de cómputo que ofrece *CoDiP2P* para resolver las batallas mediante la ejecución de tareas *Robocode*.

A. *Lógica del juego*

El objetivo final del juego consiste en vencer a todos los oponentes sobre un mapa dividido en distintas zonas.

Inicialmente, cada jugador aparece en su propio territorio y deberá expandirse para dominar el mapa; para ello deberá ir combatiendo con los *bots* enemigos y conseguir derrotar al resto de jugadores.

Este juego comparte ciertas similitudes con el juego de mesa *Risk*, pero con la diferencia que no está basado en turnos, sino en tiempo real.

B. *Diseño*

Acorde con el objetivo descrito anteriormente, los retos a alcanzar en el diseño de la plataforma *RoboDist* son los siguientes:

- Escalabilidad. No limitar el crecimiento en cuanto a usuarios, tamaño del mapa dentro de una partida y número de partidas concurrentes,

garantizando además una comunicación eficiente.

- Gestión distribuida. Toda la gestión de partidas se distribuye entre los usuarios de la plataforma, sin necesidad de entidades centrales dedicadas.
- Facilidad de uso. El usuario únicamente debe manejar una interfaz simple e intuitiva, ya que de forma transparente el sistema será capaz de autogestionar la no disponibilidad de puertos, el acceso a través de NAT o cortafuegos.
- Requisitos mínimos. Cualquier máquina, por antigua que sea, será capaz de usar la plataforma distribuyendo las tareas más pesadas sobre otros peers de la red.
- Tolerancia a fallos. Mantener la funcionalidad de la partida a pesar de la caída de los clientes/peers, incluso si el que falla es el que está trabajando como servidor de la partida.

C. Arquitectura

La Fig. 5 muestra un esquema general de la arquitectura global de la plataforma de juego. Inicialmente, los clientes se conectan al sistema haciendo login contra el servidor de entrada. Esta conexión se establece mediante un modelo cliente – servidor. Una vez autenticados, los clientes se encuentran en una sala global de espera (Fig. 5a), donde deben agruparse en distintas salas de juego para poder entablar una partida.

Cuando *n* clientes se encuentran reunidos en una sala de juego, ésta puede ser iniciada (Fig. 5b). Arrancar la sala de juego implica inicialmente asignar ciertos roles a los clientes que forman parte de ésta, ya que un cliente puede actuar como Servidor de Partida (SP) o Servidor Replicado de Partida (SRP), además de jugar la partida.

Una vez establecidos los roles empezará la partida sobre la plataforma.

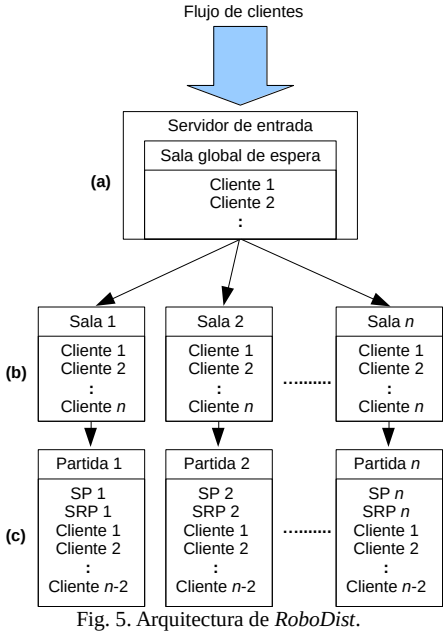


Fig. 5. Arquitectura de *RoboDist*.

Acorde con lo descrito, la arquitectura *RoboDist* está formada por dos componentes, claramente diferenciados, como se muestra en la Fig. 6: Servidor de Entrada y los Clientes (jugadores).

El servidor de entrada dota al sistema de una base de datos de jugadores para el almacenamiento de su progreso, además actúa como punto de entrada al sistema P2P y gestiona las comunicaciones con los clientes de forma segura. El cliente ofrece al usuario una interfaz para poder interactuar con el servidor de entrada y así utilizar sus servicios, además de unirse como peer para ofrecer cómputo a las partidas en curso.

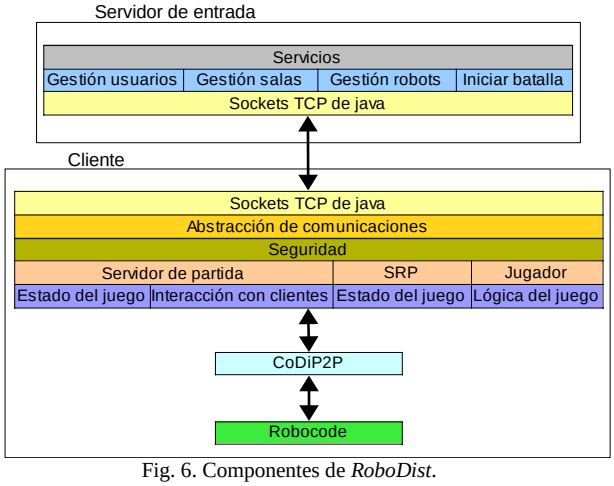


Fig. 6. Componentes de *RoboDist*.

En la Fig. 6 se puede observar que el servidor de entrada ofrece un conjunto de servicios, mediante comunicación de sockets java TCP. La conexión inicial es posible mediante sockets de java, ya que se dispone de la IP y puerto del servidor de entrada, siendo éste visible desde Internet.

En este punto se explicarán cinco aspectos clave de la arquitectura de *RoboDist* y las decisiones de diseño tomadas en cada caso:

C.1 Selección del Servidor de Partida

Una vez *n* clientes se han reunido en una partida y ésta es ejecutada, se debe decidir cuál de estos usuarios es el óptimo para actuar como Servidor de la Partida (SP), dejando en segundo plano al servidor de entrada.

La decisión tomada en este punto es conseguir que el SP sea el peer con la menor latencia respecto a los demás peers de la partida. Para ello se implementa un algoritmo que mediante el envío de pings entre clientes permite obtener un tiempo aproximado de respuesta entre estos, pudiendo de éste modo escoger el peer óptimo, con respecto al parámetro latencia, al cual poder conectarse durante dicha partida.

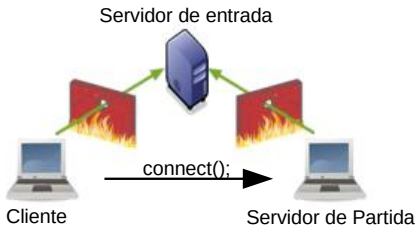
C.2 Acceso a través de cortafuegos

Decididos los roles que asumirán los peers de la partida, la siguiente cuestión es cómo se abordará la comunicación entre clientes. Las restricciones se deben a que un usuario común no debe realizar ninguna configuración de puertos en el router o poner excepciones en su cortafuegos. Por tanto, los sockets básicos de java no son efectivos en este punto.

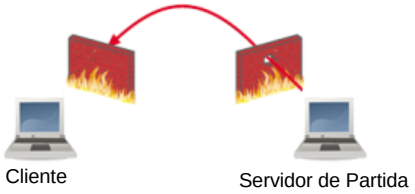
El método más indicado de los estudiados es el *hole punching* [7], utilizado en servicios de voz como *Skype [8]*. Este permite establecer una comunicación entre dos peers usando un tercer peer que tiene una conexión con ambos.

Los cortafuegos suponen un gran problema cuando se requiere de una conexión directa entre dos ordenadores mediante Internet. El truco consiste en persuadir al cortafuegos conforme ya hay una conexión establecida. Esto se consigue mediante un servidor al cual ambos están conectados. El proceso se puede desglosar en los siguientes tres pasos:

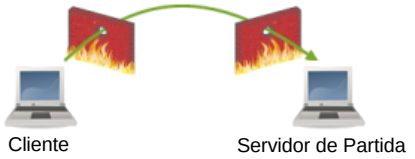
1. *Intento de conexión con el servidor de partida.* El cliente quiere establecer una comunicación con el SP (Fig. 8a). El servidor de entrada conoce las direcciones IP de ambos, así como el puerto que van a usar. La información del cliente es transferida del servidor de entrada al SP.
2. *Abrir agujero en el cortafuegos.* Como se muestra en la Fig. 8b, el SP abre un agujero en su propio cortafuegos y envía un paquete al cliente. Este paquete es descartado por el cortafuegos del cliente, pero el SP no lo sabe. Además cree que todo lo que llegue por este agujero será legítimo, siendo la respuesta al paquete enviado.
3. *Conexión completada a través del cortafuegos.* Finalmente, el servidor de entrada envía la información del SP al cliente, el cual intenta una conexión a la dirección del SP y ésta es aceptada (Fig. 8c).



(a) Conexión inicial con el Servidor de la Partida



(b) Intento conexión mediante la apertura de un agujero en el cortafuegos.



(c) Conexión final al Servidor de la Partida a través del agujero del cortafuegos.

Fig. 8. Técnica "*Hole Punching*" para conectar dos clientes a través de un cortafuegos

C.3 Seguridad

Uno de los objetivos primordiales de *RoboDist* es garantizar la seguridad del sistema, haciendo que el usuario se sienta seguro, usando un entorno robusto y fiable. Con este fin, se contempla la seguridad tanto a nivel de usuario, como de aplicación.

RoboDist dispone de servicios de validación e identificación de usuarios del sistema. Así mismo utiliza cifrado de datos para proteger las comunicaciones que se realizan. El uso de estas estrategias permite comprobar la fiabilidad de los datos recibidos, así como de las comunicaciones, obstaculizando la interceptación de información en la red.

C.4 Tolerancia a fallos

La implementación de mecanismos de tolerancia a fallos se basa en el rol del Servidor Replicado de la Partida (SRP) [9].

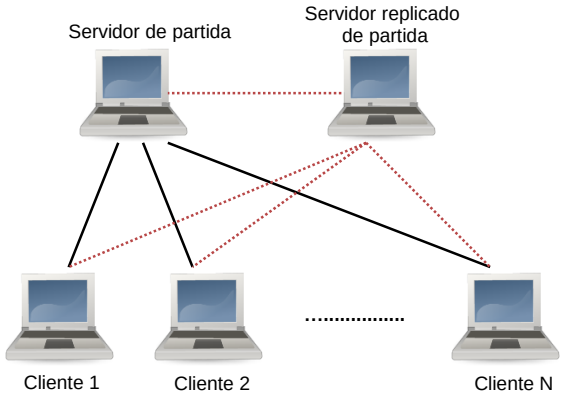


Fig. 9. Arquitectura de la partida.

La Fig. 9 muestra la conexión entre los clientes, el SP y el SRP al inicio de una partida en *RoboDist*. Este conexionado permite que el SRP sea capaz de reemplazar al SP en el caso que éste se desconecte del juego. Por este motivo, tanto el SP como el resto de jugadores de la zona envían el estado del juego al SRP (líneas rojas discontinuas). De este modo, el SRP tiene el estado del juego constantemente actualizado, estando preparado en cualquier momento para sustituir de forma transparente al SP en caso de fallo. El SRP asume que el SP ha caído siempre que no reciba ningún mensaje de éste en un intervalo de 30'. El SRP permite reducir drásticamente la probabilidad de caída de la partida, ya que ésta pasa a ser la probabilidad de caída de SP * SRP.

Para decidir que peer actuará como SRP se hace uso del mismo algoritmo usado para decidir el SP, la diferencia es que en este caso se escoge el segundo mejor peer de la partida según la latencia. Además se debe prever que el SRP podría desconectarse sin ser el servidor activo. En dicho caso, otro cliente de la partida ha de ser capaz de asumir el rol de SRP en cualquier momento, el cual sería el siguiente peer de la partida con la segunda mejor latencia respecto de los demás.

C.5 Distribución de batallas

RoboDist hace uso de una red de cómputo P2P para la ejecución de batallas. Esto permite evitar problemas con la falta de requisitos de algunos clientes del sistema, ya que si un peer no es capaz de ejecutar la tarea de *Robocode*, otro lo hará.

En el momento que un usuario entra en el sistema, ofrece sus capacidades de cómputo al propio sistema. La plataforma tiene establecidos permisos para poder

ejecutar sobre *CoDiP2P*, de modo que únicamente el Servidor de la Partida será capaz de enviar tareas.

El proceso a seguir por el SP frente a una petición de batalla, mostrado en la Fig. 10, es el siguiente:

1. Jugador A ataca a Jugador B con *bot Ta* (Fig. 10a).
2. Jugador B recibe la alerta (Fig. 10b) y responde eligiendo el *bot* a usar *Tb* (Fig. 10c).
3. El servidor de la partida tiene la información del ataque y genera la tarea *Robocode*, enviando en esta los *bots Ta* y *Tb* (Fig. 10d).
4. Transparentemente, el Peer C recibe la tarea y la ejecuta (Fig. 10e), devolviendo el resultado y la grabación de la batalla al servidor de la partida (Fig. 10f).
5. El servidor de partida recibe el resultado y la grabación de movimientos (Fig. 10g) y los reenvía a los jugadores A y B (Fig. 10h).

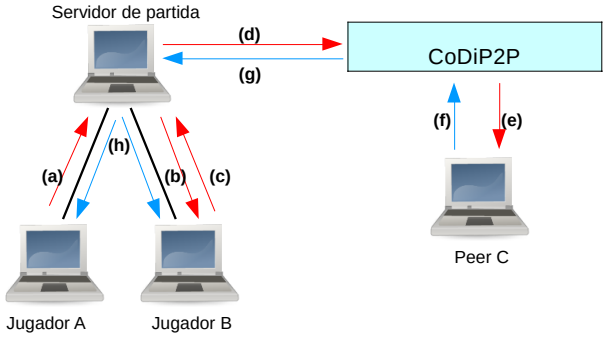


Fig. 10. Ejemplo de batalla entre dos jugadores.

Este proceso es posible gracias al uso de CoDiP2P y la alta capacidad de configuración de Robocode, los cuales se pueden ejecutar de forma transparente al usuario.

IV. Experimentación

El principal objetivo de esta experimentación es analizar los requisitos computacionales y de escalabilidad de la plataforma *RoboDist*, en función del número de jugadores y el tamaño de las batallas entre *Bots*. Como resultado de esta experimentación seremos capaces de establecer el número máximo de clientes que puede soportar una partida. Para ello se han lanzado varias partidas en una sola máquina (i7 a 2,2GHz con 4 GB de memoria RAM y utilizando Virtual Box para configurar el uso de distintas CPUs y solamente 1024MB de RAM) y se ha analizado el consumo de CPU, RAM y el tamaño del fichero donde se registran todos los movimientos de los *bots* durante la partida.

Se ha tomado como referencia para llevar a cabo esta experimentación un robot llamado *DIAMOND* [10], por ser un robot de tamaño medio de entre los publicados en Internet [11]. Con este robot se han realizado varias batallas de 10 rondas, cada una, aumentando el número de *bots* en cada batalla.

A. Requisitos de Cómputo

La Fig. 11 muestra el porcentaje de CPU consumido con respecto al número de *bots*, participantes en una batalla, y CPU activadas. En general se puede observar como el número de *bots* que están luchando en una misma batalla simultáneamente no influye en el consumo de CPU.

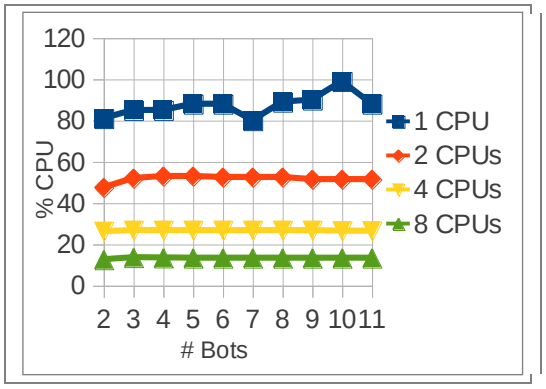


Fig. 11. Consumo de CPU según #Bots y #Procesadores.

Hay que destacar que en un sistema con 1 CPU el consumo se mantiene por encima del 80%, incluso con pocos *bots* en la partida. Esto significa que en un ordenador con pocos recursos (1 CPU), la ejecución de nuestro sistema podría ralentizar o incluso saturar sus recursos computacionales, perjudicando enormemente al usuario local. En estos casos es cuando más se aprecia la necesidad de utilizar el cómputo distribuido.

B. Requisitos de Memoria

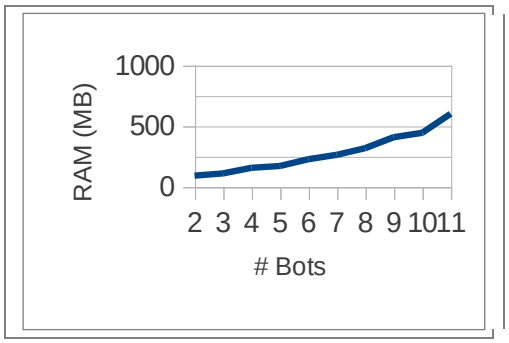


Fig. 12. Consumo de RAM según #Bots.

En lo que refiere a la memoria RAM (Fig. 12), el consumo, aunque va creciendo de forma casi lineal según el número de *bots*, no es muy alto. Se observa que con 11 *bots* tenemos un consumo de 600 MB.

C. Requisitos de red

Analizando el patrón de comunicaciones de la plataforma, se ha podido constatar que los mensajes más pesados se corresponden con la transmisión entre los

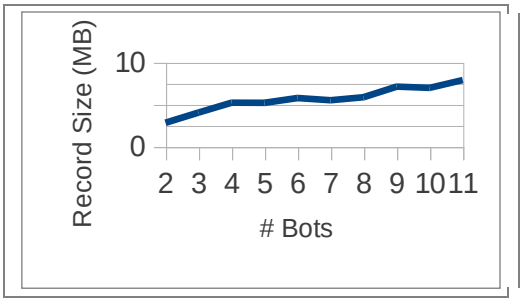
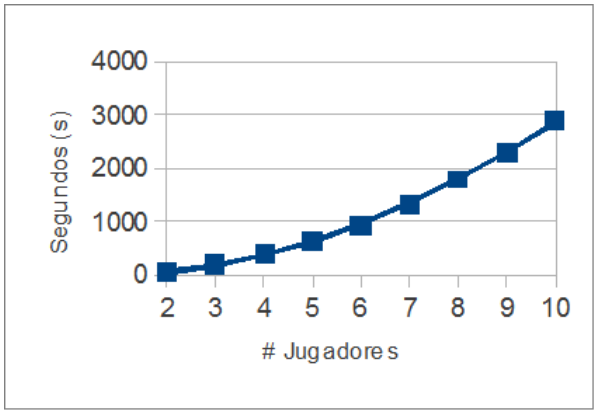


Fig. 13. Tamaño del fichero de grabaciones.



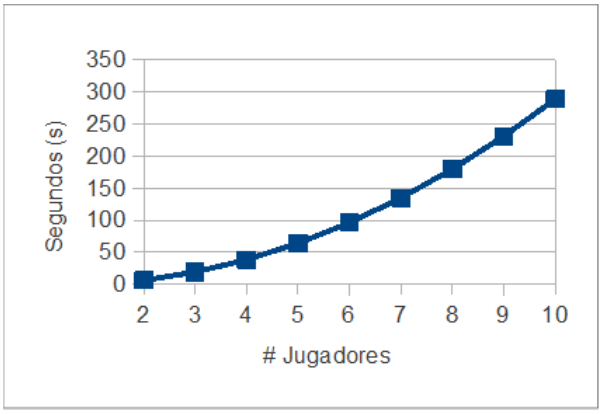


Fig. 14. Tiempo requerido para el envió del archivo de grabaciones según el # jugadores por partida. Izq: 10 rondas/batalla. Der: 1 ronda/batalla.

clientes y el servidor SP del archivo de grabación de la batalla. El fichero de grabaciones es un archivo reproducible que guarda la posición, movimiento y acciones que hace cada bot en sus turnos de batalla (una entrada cada 15ms). Por lo tanto es lógico que cuanto más tarde una batalla en finalizar, más grande será este archivo. Cabe destacar que el archivo analizado está comprimido, reduciendo aproximadamente 4 veces su volumen real.

Este archivo permite que un usuario pueda reproducir una batalla; por lo tanto, se ha de enviar al usuario cada vez que éste lo requiera, pudiendo tardar este proceso unos cuantos segundos o minutos según el ancho de banda de la conexión.

La Fig. 13 muestra el tamaño del archivo con respecto al número de *bots* en una batalla. Observamos como el tamaño del archivo crece linealmente con el número de *bots*, aunque el orden de tamaño del fichero para unos 10 jugadores está por debajo de los 10 MB. Acorde con esto, podemos deducir que el tamaño necesario para almacenar y transmitir las partidas no es un cuello de botella en el rendimiento.

En la Fig.14 se puede observar el tiempo necesario que haría falta para enviar los archivos de grabaciones de todos los jugadores simultáneamente, bajo dos diferentes situaciones: la Fig. 14.Izq. muestra el caso de que cada batalla de *bots* comporte 10 diferentes rondas, mientras que la Fig. 14.Der. se corresponde con el caso de una ronda por batalla. Cabe decir que en este segundo caso se añade un factor de aleatoriedad mayor por cada batalla disputada y además reducimos drásticamente el tamaño del archivo de grabaciones. Hemos supuesto un tamaño medio del archivo de 4MB, para el caso de 10 rondas por batalla. Asimismo se ha utilizado una línea ADSL de subida de 512kbps. En cada batalla lucharán 2 jugadores, por lo que habrá que enviar el archivo de la batalla a ambos. También suponemos que cada jugador hace una batalla con todos los otros jugadores, por lo que habrán $n*(n-1) / 2$ batallas, donde n es el número de jugadores.

Considerando que el tiempo máximo que un jugador está dispuesto a esperar el retorno del fichero de grabaciones sea de 200 segundos, en el caso de la Fig. 14.Izq obtendríamos que el número de jugadores máximo en una partida seria de 3, mientras que en el caso mostrado en la Fig. 14.Der sería de 9 jugadores.

V. Conclusiones

En el presente artículo se describe un entorno de Juego distribuido en Red, denominado *RoboDist*, el objetivo del cual es incentivar las competencias de programación entre estudiantes de Ingeniería.

RoboDist se compone por tres capas: la primera sería la lógica del juego, la segunda integra la plataforma Robocode, la cual permite la programación de los bots de los jugadores, y finalmente la capa CoDiP2P, que interacciona con la red y permite distribuir la ejecución de las batallas entre los peers que integran el sistema.

Actualmente se ha implementado un primer prototipo de *RoboDist* y sobre el mismo se han analizado los requisitos computacionales y de escalabilidad de la plataforma en función del número de jugadores y el tamaño de las batallas entre *bots*.

Agradecimientos

El presente trabajo ha sido financiado mediante el proyecto CICYT TIN2011-28689-C02-02.

Referencias

[1] http://www.usfirst.org/roboticsprograms/frc?id=966 - FIRST Robotics Programs.
[2] Cliburn, D.C. Experiences with the LEGO Mindstorms throughout the Undergraduate Computer Science Curriculum. 36th Annual Frontiers in Education Conference, pp.1-6. 2006.
[3] Sung, K. Computer games and traditional CS courses. Commun. ACM 52(12). pp.74-78. 2009.
[4] O'Kelly J., Gibson J.P. Robocode & Problem-based learning: a Non-descriptive Approach to teaching Programming. SIGCSE Bull 38 (3), pp.217-221. 2006.
[5] Chen-Chung Liu, Yuan-Bang Cheng, Chia-Wen Huang. The Effect of Simulation Games on the Learning of Computational Problem Solving. Computers & Education, pp 1907-1918. 2011.
[6] D. Castellà, J. Rius, I . Barri, F. Giné and F. Solsona. CoDiP2P: a New P2P Architecture for Distributed Computing. In Conference on Parallel, Distributed and Network Based Processing (PDP 2009), pages 323-329, 2009
[7] Peer-to-Peer Communication Across Network Address Translators - http://www.brynosaurus.com/pub/net/p2.
[8] *Skype* – http://www.skype.com/intl/es/home.
[9] I. Barri. Modelado y Planificación de Aplicaciones de Juegos Masivos Multijugador en Red en Entornos Distribuidos. Tesis Doctoral de la Universitat de Lleida, 2011.
[10] Diamond *bot* - http://robowiki.net/wiki/Diamond.
[11] *Bots* - http://robowiki.net/wiki/Category:Bots.