UNIVERSIDAD
REY JUAN CARLOS

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA INFORMÁTICA

Curso Académico 2011/2012

Proyecto de Fin de Carrera

DESARROLLO DE UN JUEGO DE ESTRATEGIA EN LÍNEA BASADO EN LA PROGRAMACIÓN DE INTELIGENCIAS ARTIFICIALES

Autor: Jairo Canales Alfonso

Tutores: José María Recio Peláez

Resumen

El proyecto consta de dos partes: por un lado se encuentra un juego de estrategia para computador y por otro lado un sistema distribuido disponible en línea. Este último sistema da soporte al juego, alojando simuladores de partidas en su núcleo y añadiendo nueva funcionalidad.

La parte del proyecto que aporta más originalidad es el sistema de juego que incluye. Mientras en la mayoría de juegos de estrategia bélica en tiempo real los usuarios escogen en todo momento cuales son las acciones a realizar por sus soldados, el juego que se implementa en este proyecto consiste en programar las inteligencias de los soldados. Podría decirse que este juego es un simulador de batallas, y el único cometido del usuario es preparar las decisiones de sus soldados antes de que se dispute la batalla.

Respecto a la segunda parte del proyecto, se implementa un sistema distribuido escalable que permite dar soporte al sistema de juego. Además de alojar los nodos que simulan las batallas, permite a los usuarios comprar nuevas unidades, objetos y armas, escoger contra quién disputa las batallas, y observar los resultados de otras batallas. Igualmente, podrá visualizar las batallas de otros jugadores con la misión de analizar y mejorar los sistemas de decisión de sus soldados.

El cometido final del usuario es ascender posiciones en el sistema de ranking, en el cual se ha trabajado activamente para establecer la mayor justicia posible al evaluar la calidad de los participantes.

Para realizar todo el sistema se han utilizado tecnologías JavaEE, bases de datos SQL y NoSQL (no relacionales). Para la comunicación entre los sistemas distribuidos se ha usado Java RMI (Invocación Remota de Métodos) y se ofrece una API REST (sobre el protocolo HTTP) para el acceso al sistema distribuido.

Sobre el autor

Jairo Canales Alfonso

Alumno de la Universidad Rey Juan Carlos en el campus de Móstoles (Madrid). Estudió Ingeniería Informática entre los años 2007 y 2012 en la Escuela Técnica Superior de Ingeniería Informática.

Entre sus intereses se encuentra el diseño de videojuegos tanto en el ámbito técnico como en otros aspectos más creativos.

Email: jairo.canalesalfonso@gmail.com

Twitter: @jairo_canales

Para el desarrollo del proyecto ha contado con la colaboración de José María Recio Peláez como tutor, profesor adjunto del departamento GSYC, que se encuentra en el Campus de Fuenlabrada.

Email: josemrecio@gmail.com

Todo los archivos creados en el proyecto pueden ser accedidos a través de Dropbox, accediendo al siguiente enlace:

http://dl.dropbox.com/u/51409718/ProyectoJairo.rar

(25 de mayo de 2012)

Agradecimientos

En primer lugar quería agradecer este proyecto a mi tutor Chema por haberme dado una libertad casi total a la hora de plantear qué hacer, y sus ayudas ambientadas en el “estado del arte”.

Además de Chema, ha habido muchos profesores de la universidad que me han ayudado en muchos momentos. Quería agradecer a Álvaro y Agustín su voluntad de ayudar en todo momento. También a Holger, quién me orientó a la hora de diseñar cómo se programan las inteligencias artificiales. Y a muchos otros profesores de los que he aprendido muchas habilidades técnicas y sociales durante sus clases. A la universidad.

Quería agradecer a mis compañeros de clase todo lo que he compartido con ellos en las clases pero especialmente fuera de ellas. A Dani, Fernando y Noelia por todos los proyectos que hemos compartido. A Carlos e Ian porque su ayuda en los primeros pasos del proyecto permitió que la idea se convirtiera en realista. A la asociación UACM y todos sus miembros y amigos. A todos mis amigos en general, que han sido mi mejor vía de escape en los peores momentos.

Al Ministerio de Educación, que me ha permitido estudiar durante toda mi carrera universitaria sin tener que pagar una sola matrícula. Además me permitió vivir la experiencia de estudiar un año en la maravillosa ciudad de Skövde (Suecia) con su ayuda a las becas Erasmus.

A todos los desarrolladores de software que contribuyen interesada o desinteresadamente al progreso global. Sin herramientas como Eclipse o Netbeans el proyecto habría sido sustancialmente más largo y aburrido. Gracias a los desarrolladores de las tecnologías que he usado.

A mi padre, que siempre se ha preocupado por sacar todo lo bueno que hay en mi. A mi familia en general porque ha sido capaz de transmitirme su alegría.

Sobretodo a mi madre, porque ha sido la que ha tenido que lidiar con mi estrés y agobio durante todos estos meses.

Índice de contenido

Índice de ilustraciones

Índice de tablas

1 Introducción

Antes de comenzar la explicación, y para evitar que el lector se pierda, se añade una nota principal sobre el contenido del proyecto.

Podría decirse que el proyecto se divide en dos grandes subproyectos: por un lado se diseña e implementa un juego bélico en el que el humano programa la inteligencia de sus unidades y en segundo lugar se desarrolla un sistema distribuido capaz de alojar el juego anterior y añadirle nuevas funcionalidades intentando aplicar los mejores principios de escalabilidad y adaptabilidad.

La estructura de esta introducción comienza por la explicación del juego. Desde un punto de vista crítico podría decirse que la primera parte del proyecto (el juego) dota de identidad y originalidad a la segunda (los sistemas distribuidos). A pesar de ello, el diseño e implementación de los sistemas distribuidos tiene sus puntos fuertes en la capacidad de adaptación a nuevas circunstancias sin tener que re-programar la mayor parte del sistema y la facilidad de añadir nodos de ejecución al núcleo de proceso, propiciando una gran escalabilidad en casi todos los aspectos.

1.1 Introducción al videojuego

El proyecto tiene su eje central en un videojuego bélico que ha sido desarrollado con motivo del mismo. Como en la amplia mayoría de juegos de este estilo, la misión del jugador es aniquilar a las fuerzas de los otros jugadores o realizar objetivos estratégicos como puede ser capturar la bandera, escoltar a una unidad especial o proteger un sector del mapa.

La mayor originalidad de este juego consiste en que las unidades que están en el equipo del jugador no son manejadas por el mismo de manera directa. En vez de ser el jugador el que cada vez que quiere que sus unidades hagan algo primero selecciona a las unidades implicadas, después selecciona la acción y por último selecciona dónde aplicar dicha acción, son las propias unidades las que deciden lo que hacer.

En este sector de juegos bélicos nos podemos encontrar dos tipos de juego que han inspirado el diseño del juego del proyecto. Por un lado se encuentran los juegos de disparos en primera persona (FPS, First Person Shooter). Los FPS pueden servir como inspiración a la hora de diseñar el comportamiento de una unidad. Se podría decir que cuando un jugador desarrolla el comportamiento de un único soldado es como si se imaginara que está

manejando un único soldado. La inteligencia de los soldados es muy reactiva, es decir, los eventos que lee son muy similares a los que se ven durante una partida de un FPS cualquiera (como ejemplo, se recomienda al lector imaginar el videojuego Counter-Strike [1]).

Lista de eventos de un FPS cualquiera:

- Podemos ver unidades
- Escuchar a otras unidades, disparos u otros sonidos
- Atender a mensajes de radio (que sirven para coordinar al grupo entero)
- Ver gran parte del terreno (es decir, podemos identificar que parte del terreno es transitable y que parte no lo es y podemos ver zonas "opacas" que podrían tener enemigos detrás).
- Ver el estado de nuestras armas (cuántas balas cargadas hay, cuántas nos quedan).
- Ver las granadas que tenemos equipadas.

Además de los eventos visibles, tenemos también una serie de acciones posibles:

- Moverse a un sitio
- Girar
- Disparar en una determinada dirección
- Lanzar una granada
- Enviar un mensaje por la radio
- Combinar varias acciones a la vez (por ejemplo, disparar mientras se avanza en una dirección)

Desde un punto de vista matemático se podría especificar el intelecto de las unidades del juego como una función que tiene por entradas todos los eventos visibles por parte del soldado, y que tiene por rango o recorrido el conjunto de posibles acciones. La función se enunciaría como f(entorno) → acción . Por poner un ejemplo: f(veo un enemigo delante) → disparar hacia el enemigo. Localizamos por tanto cual es la misión del jugador: programar una función f tal que analizando las posibles entradas del soldado (lo que sus sentidos perciben) tiene que decidir que acción ejecutar.

Lógicamente cuando el evento es tan simple como que el soldado tiene delante a un

soldado enemigo, la reacción no requiere de demasiado cómputo: sería normal escoger que queremos disparar a ese enemigo. En cualquier caso no todo el comportamiento de los soldados es tan inmediato. Si nos vemos en una situación en la que hay varios enemigos delante, la situación ya no es tan predecible. El soldado podría salir huyendo en dirección opuesta, llamar por radio a más compañeros, lanzar una granada o directamente ponerse a disparar. En este lugar la pericia del jugador como estratega es la que puede marcar la diferencia.

La programación de intelectos por parte del jugador tiene muchas similitudes con varios de los problemas que se dan en la robótica:

- Problema de localización en un entorno: el soldado no tiene ningún mapa memorizado, no sabe donde está y solo puede orientarse por lo que sus sentidos le dicen. Un soldado muy inteligente en este aspecto podría construirse un mapa de manera que pudiera orientarse mejor al volver a pasar por la misma zona.
- Limitación de los sensores: la única manera que tiene un robot de obtener información es mediante sus sensores (en el caso de un soldado del juego, su limitación es el entorno programado y los métodos de la API que se faciliten al programador). En este área los soldados tiene una ventaja respecto a los robots reales, y es que como el entorno es simulado, los sensores nunca fallan.
- El comportamiento de un robot se debería poder explicar por la misma función matemática que un soldado: f(sensores) → acción .

Siguiendo con los tipos de juego que han inspirado a este proyecto podríamos centrarnos en los juegos de estrategia en tiempo real. Si algo caracteriza a los juegos de estrategia en tiempo real (RTS, Real-Time Strategy) es que la visión que tiene el jugador sobre sus unidades es global. Todas forman un ejército, quien gana es el ejército y la pérdida de unidades sueltas es algo probable. El control de las unidades se da con una granuralidad pequeña (podemos manejar a las unidades individualmente), pero sus efectos de manera individual no suelen ser perceptibles y se requiere de acciones en grupo para afectar al flujo de la partida.

El mecanismo que el juego desarrollado implementa para permitir la coordinación entre unidades es la radio. Cada soldado percibe los mensajes de radio que otra unidad envía, pudiendo enviar más mensajes en respuesta. El contenido de los mensajes es adaptable, y a pesar de que existen una serie de mensajes estándar (me han disparado, área vacía, necesito

ayuda), se da libertad al programador para que implemente sus propios mensajes (limitando el tamaño de los mismos).

1.2 Comparación con productos ya existentes

Existen en el mercado varios videojuegos líderes de ventas que han inspirado el proyecto. Al citado FPS Counter-Strike [1], se unen otros FPS como Call of Duty [2] o Battlefield [3]. La saga Brothers in Arms [4] incorpora un sistema de ordenes mediante el cual podemos mandar sobre un pequeño comando de tropas, manejando activamente a dicho comando.

Por otro lado, desde el punto de vista de los juegos de estrategia, el juego desarrollado podría verse como la saga Panzers. En dicha saga además de poder controlar los famosos tanques alemanes, se manejan grupos de unidades especialistas en sus áreas (francotiradores, zapadores, médicos,...). Además, durante el desarrollo de una partida de Panzers [5] hay un punto muy similar con el juego desarrollado: no existen edificios productores ni unidades recolectoras en tu ejercito, de manera que la única misión del jugador es hacer la guerra con las unidades de las que dispone.

Finalmente, el producto más similar al juego desarrollado que existe es Robocode [6]. Dicho juego consiste igualmente en programar el motor de decisiones de una unidad (en el caso de Robocode es un tanque), tratando de destruir al tanque enemigo aunque existen más variantes de juego (2 contra 2, o modo melé entre otros). Las diferencias entre el juego del proyecto y Robocode son varias: Robocode propone un escenario menos realista dónde, generalmente, un tanque se enfrenta contra otro pero los margenes de acción de los tanques son muy grandes: un tanque que va suficientemente rápido puede evitar un disparo. Además, la cooperatividad dentro de un mismo equipo no es un factor clave. Finalmente, podría decirse que la misión global de Robocode es que el ganador sea el mejor programador, mientras que el ganador del juego que se ha desarrollado para este proyecto debería ser el mejor estratega (aunque tenga que programar la estrategia igualmente).

1.3 Introducción al sistema distribuido

Además del sistema de juego, se ha desarrollado una gran estructura software distribuida para aumentar la funcionalidad del juego y permitir su uso de manera Online. Las puntos que han guiado al proyecto han sido: propiciar la mayor escalabilidad posible, permitir una gran capacidad de adaptación a a las necesidades del sistema, permitir la incorporación de nueva funcionalidad y ser seguro frente a caídas inesperadas en servicios secundarios.

Para desarrollarlo se ha usado un enfoque de diseño orientado a objetos. De esta manera la descomposición del sistema se hace en clases. Lo que dota al sistema de su carácter distribuido es la capacidad de colocar las distintas instancias de las clases en distintos nodos, haciendo que cada nodo responda de una funcionalidad específica. La comunicación entre los distintos objetos se da usando RMI, aunque para la localización de los mismos se ha implementado un servidor de nombres (también llamado registro de instancias u objetos) ligero capaz de indicar en que máquina se encuentra cada componente.

Finalmente, las capacidades de responder ante fallos en servicios secundarios y la mejora en la escalabilidad se dan al permitir que una clase tenga distintas instancias y a su vez dichas instancias se puedan alojar en distintos nodos (funcionalidad replicada).

Por otro lado, para permitir una correcta persistencia de la información, se han usado dos tipos de bases de datos: en primer lugar MySQL (base de datos relacional), uno de los sistemas de código libre líderes del mercado (usado en Twitter con algunas modificaciones [7]). Las capacidades de MySQL para ser distribuido y clusterizado quedan probadas en [8]. El segundo tipo de bases de datos usada es no relacional. El uso que se ha dado de dicha base de datos es para guardar archivos de texto (que ocupan del orden de KB o MB). Se ha usado Apache Cassandra, cuyas capacidades para la clusterización y distribución pueden verse en [9].

1.4 Estructura del documento

Una vez comprendida la temática del proyecto, el resto del documento queda estructurado de la siguiente manera:

- Objetivos: Se describe el problema que se intenta abordar con este proyecto, se estudiarán las distintas alternativas y se explicará la tecnología empleada.
- Diseño de los requisitos y la funcionalidad: Debido a que el sistema es ideado por el autor del proyecto, se explica toda la funcionalidad del mismo. Además, se analiza dicha funcionalidad dejando claro qué se implementará en el sistema y cómo queda organizado. Además de analizar las especificaciones propuestas, se tiene en cuenta la manera en la que problemas similares han sido resueltos.
- Descripción informática: En esta sección se explicará toda la realización del sistema propuesto, incluyendo su planificación, diseño, implementación y unas guías de mantenimiento.

- Conclusiones: El contenido de esta sección es un pequeño resumen con los conocimientos y competencias adquiridas. También servirá como análisis de la solución propuesta de una manera global. Finalmente, será en esta sección dónde se incluirá el futuro trabajo propuesto para mejorar el proyecto en los distintos ámbitos que abarca.

1.5 Glosario de términos

Antes de comenzar, podría ser necesario para el lector poner en comunión con el autor algunos términos:

- Por acortar, y debido a que en este preciso instante el proyecto no tiene un nombre comercial, la manera en la que se nombrará a las distintas partes del proyecto será: juego (primera parte de la aplicación) y sistema distribuido (resto del sistema). En algunos pantallazos se usará el nombre "AIWars" para referirse al sistema completo.
- Reporte: representación de una partida ya ocurrida. Contiene toda la información necesaria para reproducir la partida, guardando las posiciones de los soldados en cada turno y las acciones que desempeñan.
- Intelecto: representación del motor de decisiones de una unidad. En la versión actual de la aplicación se necesita programar en Java dicho motor.
- Usuario: persona que usa la aplicación. Un usuario programa los intelectos de sus unidades para que en la simulación de la partida se comporten como él quiere.
- Jugador: igual que usuario.
- Soldado: unidad básica del juego. Requiere un intelecto para saber como actuar. Las acciones que puede realizar son diseñadas en la sección de diseño de la funcionalidad.
- Unidad: aquel objeto que puede ser controlado por el usuario mediante la programación de intelectos. En la primera versión solo hay un tipo de unidades: soldados. Posteriores versiones pueden incluir tanques o helicópteros.
- Localización por IP: funcionalidad que permite saber dónde se encuentra un usuario en función de la IP desde la que se conecte al sistema.
- Interfaz: se va a usar con dos acepciones. Una interfaz puede ser:

  1. Capa más externa del software de un sistema que es utilizada por otros sistemas

para comunicarse con él. Se diseña pensando en la facilidad de interacción con otros sistemas.

2. También referido como interfaz gráfica. Parte visual de una aplicación. Se diseña pensando en la facilidad de uso del usuario.

- Cluster: conjunto de computadores homogéneos que se comportan como si fueran uno. Es decir, varios ordenadores que ejecutan la misma funcionalidad.
- Clusterizable: se dice de un software que es capaz de ejecutarse con totales garantías en un cluster.
- Replica: reproducción o copia. Se dice que la información está replicada cuando existen dos o más copias de la misma información en lugares distintos. Igualmente un servicio estará replicado cuando se pueda ejecutar en dos o más nodos simultáneamente.
- Replicable: que se pueden hacer replicas de ello. Un componente es replicable cuando puede colocarse a la vez en dos o más nodos distintos.
- Servidor de nombres: aplicación software que permite asociar nombres de servicio a las direcciones en las que dichos servicios pueden ser obtenidos.
- Registro de instancias u objetos: al igual que el servidor de nombres, se asocian nombres de servicio a las direcciones dónde se encuentran los objetos que allí se registran.
- Colisionable: clase desarrollada en el motor de juego creada para implementar el motor de colisiones. No solo las unidades son colisionables, también lo son los elementos que delimitan el escenario.

2 Objetivos

2.1 Descripción del problema

El objetivo principal de este proyecto es desarrollar un videojuego completo en línea. A pesar de que la idea del juego es clara (juego en el que inteligencias artificiales compiten), dicha idea debe ser totalmente definida y estudiada. Una vez toda la funcionalidad del sistema está clara, se debe aplicar un análisis sencillo de dicha idea de manera que el proceso de desarrollo sea fácilmente entendible a partir de ahí.

Entre los objetivos principales del alumno existen los siguientes:

- Se usará la distribución de sistemas, de manera que la funcionalidad se reparta físicamente entre distintos equipos.
- Se deberá trabajar activamente en permitir una alta escalabilidad. Imaginando que por azares del destino el sistema es ámpliamente usado, la experiencia del usuario debería ser igual de positiva sin importar cuanta gente hay jugando en el mismo momento.
- La tolerancia a fallos del sistema será una prioridad fundamental.
- Siempre que sea posible y adecuado se buscarán y usarán tecnologías libres ya creadas y probadas que faciliten el desarrollo del sistema.

2.2 Estudio de alternativas

La simpleza del juego hace que no se requiera de ningún motor de juegos como pudieran ser Quake Engine o CryEngine [10].

Para la parte de sistemas distribuidos si se necesitará usar distintas tecnologías. El estudio de alternativas se hará una vez la funcionalidad total del sistemas distribuido haya sido diseñada.

2.3 Metodología empleada

El desarrollo del proyecto va a ejecutarse por una única persona (el autor). Además, el autor del desarrollo es también el creador de la idea. Esto implica varios puntos:

- La interpretación de los requisitos una vez diseñados no tendrá ambigüedades para el desarrollador. Se puede reducir en gran parte el esfuerzo por formalizar los mismos.

- La fase de análisis de los requisitos tampoco será demasiado formal. La conversión entre los requisitos diseñados y el diseño del sistema se basará en gran medida en la experiencia del desarrollador. Se considera que utilizando formalismos se llegaría a una solución muy parecida y mucho más costosa en tiempo.
- Las pruebas del sistema no se centrarán únicamente en la corrección de errores (aunque esto no implica que dichas pruebas no existan). Se busca medir si el sistema es escalable verdaderamente y cuántos usuarios podrían funcionar simultáneamente.
- Los detalles del modelo de proceso se decidirán una vez diseñada la funcionalidad global del sistema.

3 Diseño de los requisitos y la funcionalidad

3.1 El sistema de juego

Puesto que el autor del proyecto ha sido a su vez el creador de la idea original, la amplia mayoría de los requisitos han sido ideados por el mismo. Hay que tener en cuenta que las prioridades pensadas por el autor podrían no ser las idóneas en algún otro sistema de juego, pudiendo ser superfluas o incluso contraproducentes.

El desarrollo del motor de juego tiene las siguientes prioridades:

- Justicia: todos los usuarios tienen que tener las mismas oportunidades. No puede existir ningún hecho que otorgue una ventaja a un usuario respecto de otro más que su propia habilidad. Esto es tajante, el diseño no puede permitir que en ningún caso los soldados de un jugador tengan un mejor trato por parte del sistema que los de otros jugadores (un mejor trato puede implicar una ejecución más rápida de los algoritmos de selección de acción o disparos que aciertan con mayores posibilidades).
- Realismo: el resultado de ejecutar una determinada acción tiene que seguir una lógica entendible y comprensible dentro de un punto de vista humano. Un disparo a una distancia relativamente pequeña debería ser un acierto casi siempre, mientras que uno más lejano puede fallar. El alcance de una granada tiene que ser acotado. Disparar a través de las paredes no debería ser posible. Los soldados deberán tener velocidades relativamente lógicas en función del tiempo.
- Intuitividad: se debe facilitar al usuario la comprensión del sistema. Es de vital importancia que la curva de aprendizaje del sistema para el usuario sea lo más suave posible.
- Jugabilidad: el juego tiene que ser divertido. Se deben evitar las frustraciones generadas por el sistema de juego. Esto deberá ser objeto de análisis puesto que la actividad del usuario durante la partida es nula (recordemos que la actividad del usuario se da antes de la partida al programar a sus soldados).
- Transparencia: se debe evitar toda ocultación de información en el motor de juego.
- Facilidad de modificaciones: se debe permitir añadir al juego nuevo contenido fácilmente: soldados evolucionados, nuevas armas, nuevos tipos de unidades.

- Facilidad de configuración: una partida debería ser fácil de configurar (número de unidades por bando, armas que dichas unidades llevan, configuración del terreno).

3.1.1 Para empezar...

¿Cómo se hace un juego en el que intervienen inteligencias artificiales?

La pregunta no es sencilla y pueden existir varios enfoques. El que se ha tomado en el juego es: se pregunta a cada una de las inteligencias "este es tu entorno, ¿qué quieres hacer?". El resultado de dicha pregunta será una acción. Posteriormente el motor de juego procesa la acción y produce un nuevo estado del juego.

¿Cómo se permite al usuario programar una inteligencia artificial?

Usando el concepto de polimorfismo de programación orientada a objetos, se diseña una clase abstracta "Intelecto", la cual tiene un método "decidir". Dicho método será llamado por el motor de juego cada vez que se requiera la acción de una inteligencia. Si el usuario programa una clase que herede de Intelecto y sobrescriba el comportamiento del método "decidir", el resultado de llamar a dicho método dependerá de la programación del usuario.

Es importante tener en cuenta que el código programado por el usuario deberá ser compilado y enlazado sin necesidad de recompilar el motor del juego.

3.1.2 Filosofía

A continuación se van a analizar las distintas prioridades especificadas anteriormente tratando de vislumbrar posibles implementaciones del sistema:

(P1) Justicia: para cubrir dicho requisito se debería conseguir que el motor de juego ejecutara por cada soldado su algoritmo de decisión el mismo número de veces. Por un lado podríamos tomar un enfoque basado en hilos (threads), de manera que cada soldado se ejecuta en un hilo distinto con el siguiente comportamiento: leer entorno, decidir acción, generar entorno nuevo. El problema de dicho enfoque es que la gran mayoría de implementaciones de hilos que existen en los lenguajes actuales no asegura igualdad de oportunidades, es decir, podría darse el caso de que un hilo se ejecuta significantes veces más que otro. También podría pasar que, teniendo dos soldados frente a frente, el hilo del soldado que primero llegue a ejecutar enviara primero la acción "disparar", produciendo un resultado injusto (los dos deberían verse al mismo tiempo y disparar a la vez).

El enfoque que se ha seguido asegura que todos los soldados se ejecutan el mismo número de veces y relativamente a la vez. Basándose en un sistema de turnos, y estableciendo que E(n) es el entorno global del sistema en un instante n, se tiene el siguiente bucle de ejecución:

Por cada soldado: procesar E(n) y guardar la acción decidida.
Por cada acción decidida: procesarla en el entorno y producir todos los cambios oportunos. Guardar el nuevo entorno como E(n+1).

Con este enfoque se supone un número finito y natural de instantes o turnos para que una partida finalice (teniendo un ganador, varios o ninguno), facilitando la exportación de partidas (se guarda el estado de cada uno de los turnos).

¿Qué ocurre entonces si dos soldados enemigos se ven simultáneamente y deciden disparar? ¿Quién mata a quién? Si la justicia se asegura ambos soldados deberían resultar muertos. El enfoque seguido para conseguirlo es separar las acciones de las consecuencias de las mismas, de manera que una acción puede tener distintas consecuencias (un soldado que corre de un lugar a otro produce dos consecuencias: un movimiento y un sonido). El procesamiento del momento en que los dos soldados se ven quedaría de la siguiente manera:

Soldado1 decide disparar a Soldado2 → **Acción A**
Soldado2 decide disparar a Soldado1 → **Acción B**
Acción A es procesada → **Consecuencias A1, A2**
Acción B es procesada → **Consecuencias B1, B2**
Consecuencia A1 es procesada → Se produce un **disparo desde la posición de Soldado1 hasta la posición de Soldado2.** Acierta. Soldado2 muere. Dicha consecuencia no afecta al resto de las consecuencias que ya estaban en la cola.
Consecuencia B1 es procesada → Se produce un **disparo desde la posición de Soldado2 hasta la posición de Soldado1.** Acierta. Soldado1 muere.
Consecuencia A2 es procesada → Se produce un **sonido de disparo en la posición de Soldado1.**
Consecuencia B2 es procesada → Se produce un **sonido de disparo en la posición de Soldado2.**

Nota: para garantizar una cierta integridad, los disparos y ataques se ejecutan antes que cualquier otra acción.

(P2) Realismo: cuando una acción se convierte en consecuencia, dicha acción es procesada para producir resultados semi-aleatorios. Así, por ejemplo, una AccionDisparo desde P1 hasta P2, producirá: ConsecuenciaDisparo desde P1 hasta P2', siendo P2' una posición similar a P2

con un margen de error que dependerá de la distancia a P1 y del arma usado para disparar.

(P3) Intuitividad: al margen de lo sencillo que sea imaginar los resultados de una determinada acción, se debe crear una pequeña guía de lectura muy recomendada que explique todas las posibles acciones a tomar, así como las consecuencias que produce en el entorno.

(P4) Jugabilidad: llegados a este punto es normal que el lector siga dudando si el juego del que trata el proyecto es un juego como tal. En cualquier caso, el sistema debe tener un comportamiento que reduzca la frustración del usuario al usarlo. Muchos videojuegos actuales son causa de frustración al producir resultados que no dan al usuario ninguna oportunidad de escapar (un francotirador que mata a todo el que atraviesa una calle -Call of Duty o Counter-Strike-, un defensa que nunca cubre al delantero correctamente y hace que te metan gol -Fifa12-, jugar contra la máquina y que esta te ataque mucho antes de tu tener un ejercito listo -Warcraft III-).

Al margen del apartado realista, se debe tener en cuenta que dar segundas oportunidades al jugador puede reducir su frustración. Es por esto, que la resistencia de los soldados es grande: un soldado puede aguantar varios disparos sin morir, incluso una granada relativamente cercana. En cualquier caso, y para no ir en contra del principio de realismo, dichas modificaciones no serán demasiado severas, y una granada que explota a los pies de un soldado es mortal, al igual que un disparo de escopeta a muy poca distancia.

Finalmente, algo que sirve para aumentar la jugabilidad es enseñar al usuario a jugar. Además de las guías de las que se habla en el apartado de intuitividad, deberán existir enemigos de prueba que sirvan al usuario para testear su progreso. Un sistema en el que los usuarios pudieran exponer sus dudas y resolver las de los demás también podría ser acertado (se podría crear un foro de dudas por ejemplo).

(P5) Transparencia: todo el motor del juego será publicado bajo licencias que permitan a cualquier persona acceder al código fuente. Además de proporcionar una cierta seguridad al usuario de que existe igualdad de condiciones, tiene la ventaja de que los usuarios que analicen el código pueden reportar los errores que encuentren y sugerir modificaciones.

(P6) Facilidad de modificaciones: cada unidad deberá tener una amplia lista de características que afecten a su comportamiento. La lista de características que se ha estudiado es: salud, salud máxima, tamaño, velocidad al andar, al correr y al girar, distancia a la que puede oír, distancia a la que puede ver con nitidez, distancia a la que percibe el terreno,

ángulo de visión, precisión base con armas, daño al atacar cuerpo a cuerpo, defensa en el cuerpo a cuerpo, ante disparos y proyectiles, distancia a la que puede atacar cuerpo a cuerpo. Los valores por defecto se corresponderán a los de un soldado "normal", pudiendo ser modificados fácilmente para añadir otras unidades terrestres. Por ejemplo se podrían modificar tamaño, velocidad y ángulo de visión de manera que la unidad se comporte como un tanque. Además, cada unidad tendrá una lista de armas, teniendo cada arma unas características propias: alcance, munición total, munición que cabe en un cargador y munición que hay cargada en el momento, balas por ráfaga, precisión del arma y daño que provoca. Ajustando los valores de dichas características sería sencillo añadir cualquier otro tipo de arma.

Haciendo un buen diseño orientado a objetos, debería ser sencillo introducir unidades especiales (helicópteros por ejemplo) introduciendo las menores modificaciones posibles en el resto del motor de juego. Para hacerlo habría que basarse en la propiedad de la herencia.

(P7) Facilidad de configuración: para cada partida que el motor de juego simule se deberá crear un fichero XML con toda la información que el motor necesita. Dicha información incluirá las características de unidades y armas, así como el estado inicial del entorno. El uso de un archivo XML permite modificaciones fáciles de dos tipos: humanas, que se dan cuando un usuario quiere construir el entorno que desea; y hechas por otros sistemas, que se dan cuando otros sistemas configuran al motor de juego para simular una partida en función de la configuración que el sistema externo tenga.

Cuidando todos los principios enumerados anteriormente, se deberá tener en cuenta también que puesto que el usuario va a escribir código que posteriormente será ejecutado en el servidor, dicho código tendrá que tener unas restricciones extra (un usuario no debería, por ejemplo, poder apagar el servidor remoto con una instrucción).

3.1.3 Herramientas

También se debe tener claro que al margen del motor de juego, se deben crear herramientas que permitan probar el correcto funcionamiento de dicho motor. También deberían permitir depurar los intelectos programados.

No todas las herramientas tienen por qué ser aplicaciones. Sería muy útil publicar una guía con un par de capítulos a modo de tutorial. También se debería adjuntar al simulador de juego un documento con instrucciones básicas sobre como usar los archivos XML de configuración

de partidas.

Finalmente, y con la misión de facilitar a los usuarios la programación de intelectos, se debería crear un editor de comportamientos que no requiera programar código fuente. Dicha herramienta podría ser un editor con sistema “Drag and Drop” (arrastrar y soltar) que permita desarrollar y entender fácilmente los algoritmos de decisión. El editor “Drag and Drop” no hace más que representar un diagrama de flujo en el cual partimos siempre de un mismo nodo “inicio” aunque puede acabar en varios nodos “fin”, que en nuestro caso representa la acción que se decidiría en dicha situación. Podemos ver a continuación algunos ejemplos de diagramas de decisión:

Ejemplo 1:

Se expone un algoritmo básico. Cuando el soldado ve a un enemigo lo intenta disparar. En el caso de que no tenga balas intentará recargar. Si no ve a nadie girará. El comportamiento que el soldado mostrará es equivalente a un radar armado.

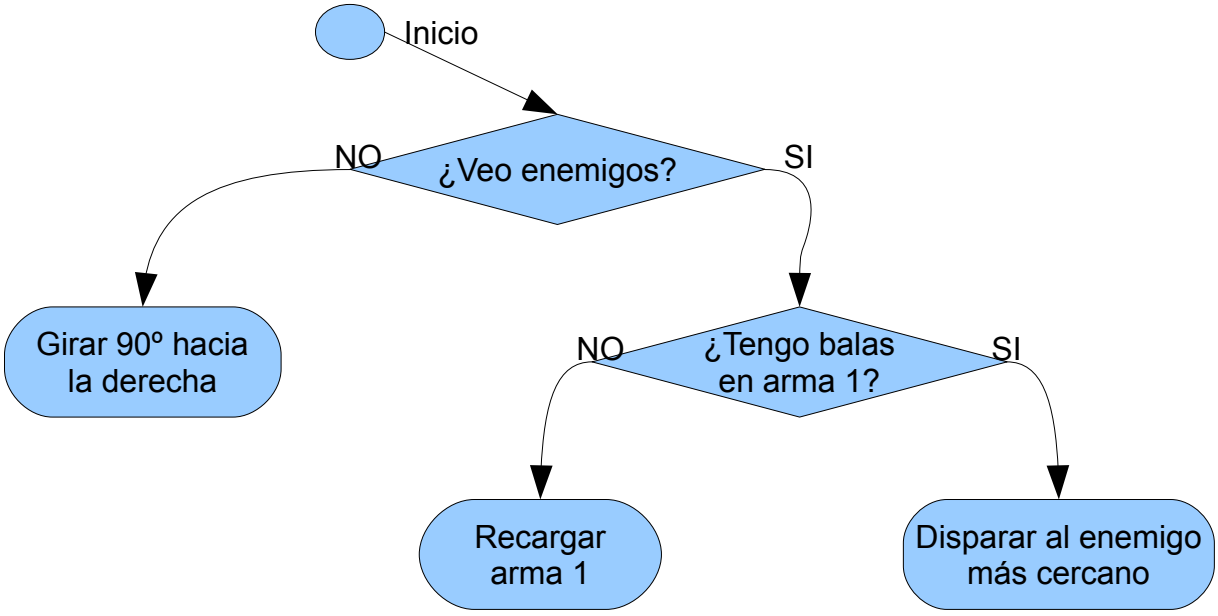

Ilustración 1: Algoritmo de decisión simple

Ejemplo 2:

Este algoritmo tiene una mejora respecto al anterior. El soldado tiene memoria sobre si ha visto enemigos anteriormente. De esta manera si el soldado deja repentinamente de ver al enemigo que estaba viendo, en vez de girar inmediatamente esperará un turno para hacerlo. Visto de esta manera no parece aportar una gran ventaja, pero muestra el potencial de usar la

memoria del soldado.

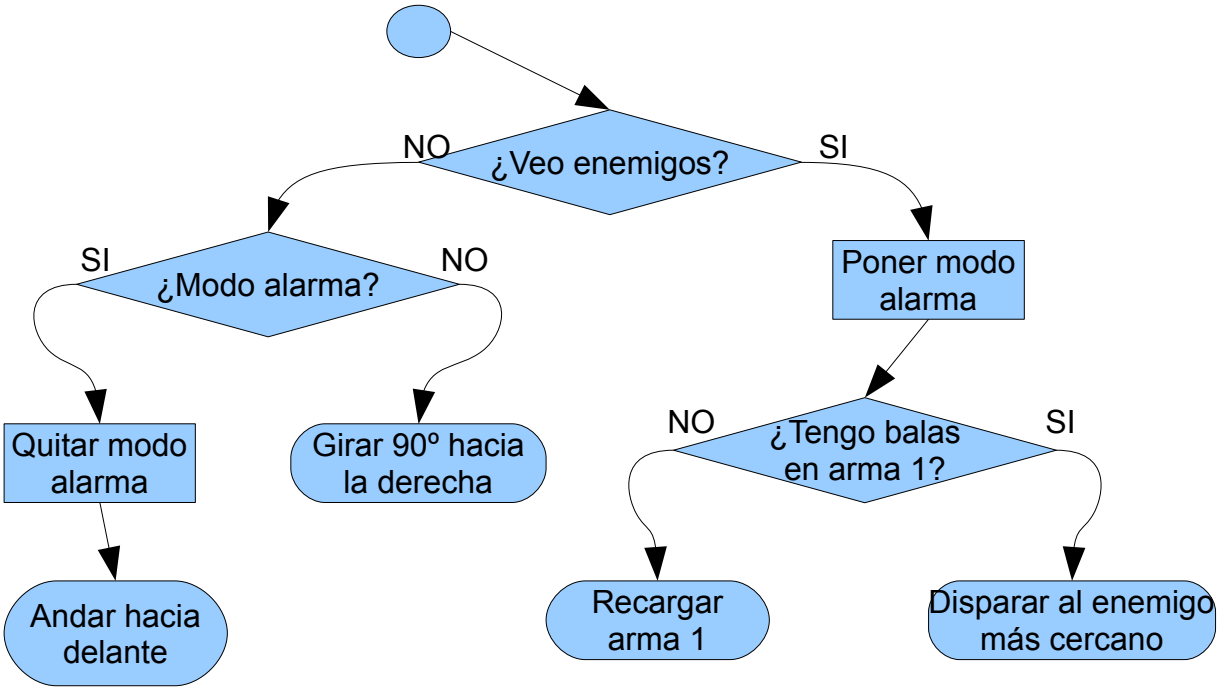

Ilustración 2: Algoritmo de decisión con memoria

Ejemplo 3:

El último algoritmo que se expone representa el comportamiento de un soldado zombie. Este soldado no importa si tiene armas o no, se comporta como un zombie. Cuando ve enemigos corre hacia ellos y los ataca. Si no ve a nadie y escucha un ruido se gira hacia él. Finalmente si ni ve, ni escucha a nadie, se comportará de manera errática. Andando y girando aleatoriamente.

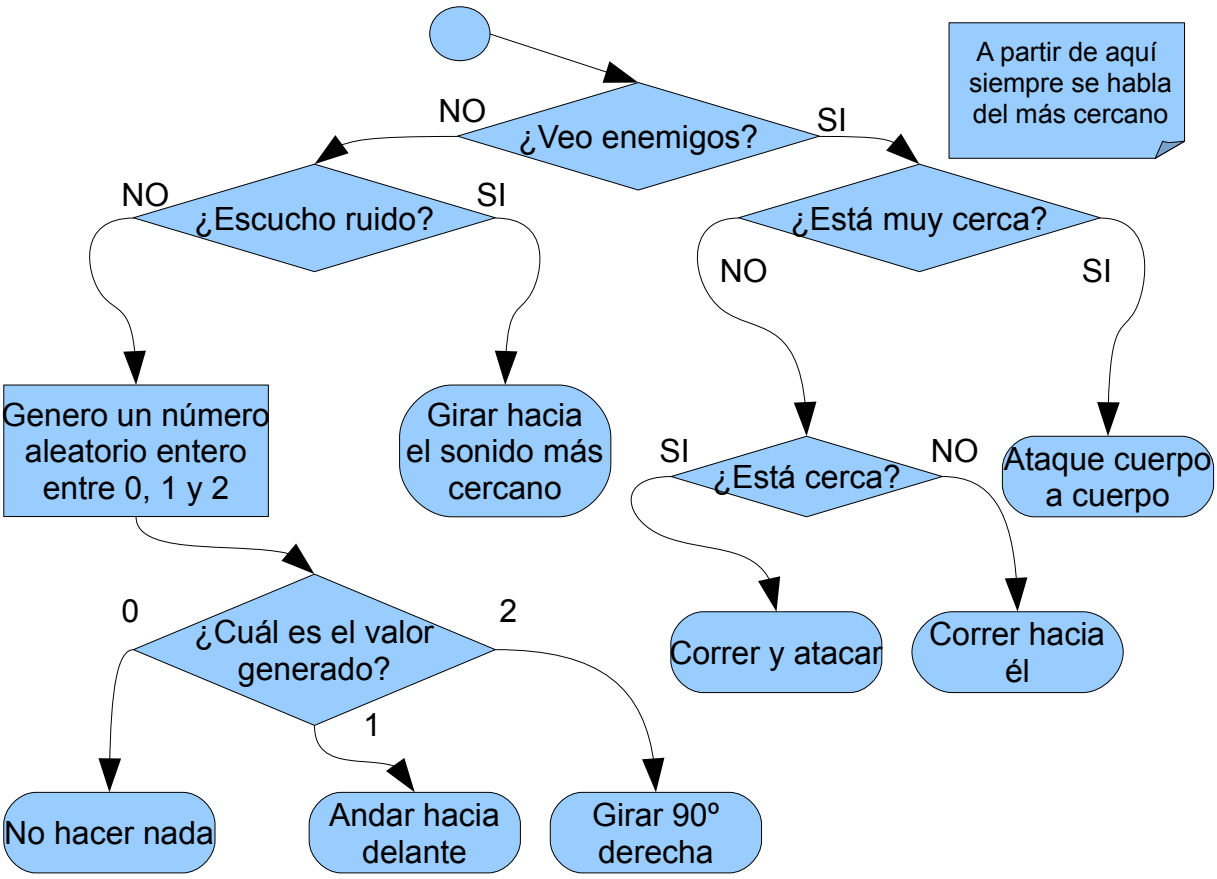

Ilustración 3: Algoritmo de decisión complejo: Zombie

La idea de usar un editor de este estilo para programar ha sido usada en muchos otros proyectos. En especial, los modelos de robots Lego NXT pueden ser programados por un software basado en gráficos que permite diseñar completamente el comportamiento del robot, aunque en su caso también se tiene en cuenta una idea de "secuencia" (en un diagrama se puede ver que hacer en cada momento y no tanto la toma instantánea de decisiones). En la referencia [11] pueden encontrarse los distintos métodos para programar los robots LEGO NXT .

3.2 El sistema distribuido

3.2.1 Adaptación del juego a su uso Online

El primer y máximo requisito que debe tener el sistema distribuido (visto esta vez como si fuera un único componente) es ser capaz de comunicarse e integrar el motor del simulador de juego. Existirán nodos dentro del sistema distribuido que requerirán el uso del simulador de juego. Dichos nodos tendrán que hablar el mismo lenguaje XML de configuración de partidas que el simulador interpreta.

3.2.2 Características añadidas

Pero el sistema distribuido no se limita a contener el motor. El hecho de colocar el motor Online no tendría ningún sentido si no se añadiera ninguna funcionalidad nueva. Teniendo en cuenta el diseño del motor de juego, una función que cobra mucho sentido es permitir al usuario configurar sus propias tropas. En los juegos sociales actuales uno de los requisitos que permiten a los jugadores volver a jugar a un determinado juego es la capacidad de personalización.

Hay que notar un punto fundamental: el tipo de usuario que va a jugar a este juego no será un jugador medio, y en ningún caso será un jugador casual. Se podría decir con bastante seguridad que en la primera versión de la aplicación, al menos el 90% de los usuarios tendrán conocimientos medios/altos de programación.

En cuanto a la funcionalidad añadida, existen novedades de dos tipos: competitivas y de configuración o personalización.

Añadidos de configuración:

- Se debe permitir al jugador comprar nuevos soldados. Su plantilla de soldados será escasa inicialmente y la calidad de los soldados iniciales muy mejorable. La calidad del soldado y el dinero que cuesta deberían ir de la mano.
- Al comprar un soldado este estará totalmente desequipado. El jugador debería poder comprar nuevos objetos que añadirle al soldado. Por ejemplo, un soldado con chaleco antibalas debería recibir menos daño cuando un disparo le da.
- Al comprar un soldado este estará totalmente desarmado. Será misión del jugador comprar las armas que considere necesarias para abastecer y configurar a sus soldados. El tipo de arma que lleven los soldados afecta mucho estratégicamente al desarrollo de la partida.
- El jugador deberá comprar la munición que utiliza con las armas. Cada arma solo puede utilizar un tipo de munición y cada tipo de munición solo puede usarse en un arma. Un soldado con escasa munición, si no es capaz de adaptar su comportamiento, probablemente se quede sin balas cuando las necesite.
- Todos los objetos y armas y munición son intercambiables. Se podrá equipar a un soldado para una partida con un determinado arma, y a la siguiente ponérselo a otro soldado. Podremos jugar una partida contra un jugador con armas de corto alcance

mientras que la siguiente partida la juguemos con rifles de larga distancia. La elección del equipamiento es un apartado más de la estrategia del usuario.

- Cada soldado tendrá un intelecto asignado. Varios soldados pueden compartir el mismo intelecto pero no es necesario.

Añadidos competitivos:

- Todos los usuarios tienen igual acceso a los distintos objetos que hay en régimen de competitividad: existen N unidades de un determinado objeto. Las N primeras compras sobre dicho arma funcionarán, el jugador que llega tarde no podrá comprar el objeto.
- Ampliando el punto anterior, los usuarios recibirán dinero ficticio al registrarse. Para obtener más dinero deberán participar en batallas contra otros jugadores. La cantidad de dinero recibido al participar será modulado en función de cómo sus soldados se han desenvuelto en el combate.
- La finalidad del juego es ser el mejor del mundo. Se implementará un sistema de ranking que permita a los usuarios comprobar en que posición se encuentran y a que distancia están de sus competidores.
- Tratando de aumentar la competitividad, se implementarán sub-rankings nacionales.
- A la hora de determinar el país del que viene un determinado usuario desde el sistema, se implementará un servicio de localización basado en la dirección IP de la que provenga el usuario.
- Cuando el usuario pretenda jugar una partida accederá a una especie de sala común dónde todos los jugadores que pretendan jugar estarán esperando. Desde dicha sala se podrán ver las partidas que hay creadas y que aún no han comenzado. Además de esa información se verá el país del creador de la misma (útil para el ranking nacional) y cuantos jugadores hay esperando.
- Cuando un usuario crea una partida podrá escoger la configuración de la misma (número máximo de usuarios, número máximo de soldados por usuario, tipo de generación del terreno,...).
- La visualización de una partida una vez lanzada también es muy importante. Según se está simulando la partida, está será almacenada en un sistema de gestión de

"Reportes".

- Cualquier usuario podrá visualizar cualquier partida que se simule. De esta manera un usuario puede ver como se comportan los intelectos de un determinado usuario en partida, pero nunca puede ver el código fuente tal cual.

3.2.3 La creación de una API REST

Un punto importante que describe técnicamente el sistema global es cómo se accede a él. En el caso del sistema desarrollado tenemos una serie de restricciones a priori que nos van a facilitar la toma de decisiones:

- Se busca un sistema que pueda ser usado por varios usuarios de manera simultanea.
- El sistema debería ser independiente de la plataforma que usa el usuario que accede a él.
- La implementación de una nueva interfaz (en este sentido referida a las interacciones con el usuario) no debería restringir el lenguaje que se usa para la misma.
- Un usuario experto debería poder ejecutar sus propios scripts para acceder a la funcionalidad tal cual el sistema es diseñado y no como la interfaz le permite. El protocolo del sistema a nivel de aplicación debe quedar claro y explicarse de manera que todos los usuarios puedan acceder a él. Si es público, se facilita una igualdad de condiciones máxima: el usuario que no acceda al sistema de la manera que desea será porque no lo necesita.
- El sistema debe incluir algún mecanismo que controle el acceso de los jugadores. Habrá información que pueda ser dada a cualquier usuario, mientras que otra información requiera autorización para acceder a la misma (un usuario puede ver la configuración de sus tropas, cualquier otro usuario no puede ver). Cuanto más explícito sea, más fácil de depurar. Todas las llamadas al sistema que requieran autorización deberán ser documentadas.

Si analizamos los 5 puntos anteriores, existen dos alternativas que cumplen todos los puntos: usar servicios web basados en llamadas remotas a procedimientos (RPC) o basados en representación de estados (REST). No es la misión de este proyecto comparar o describir exhaustivamente ambos enfoques, pero nos vamos a quedar con los puntos que motivan la

decisión final:

- De entre los posibles RPC, los más conocidos que se definen como multiplataforma son basados en CORBA (que usa su propio lenguaje de interfaces) o llamadas a servicios remotos (usando una interfaz WSDL).
- En REST se usa el protocolo HTTP para las distintas interacciones con el servidor. Se podría decir que usar REST es, por norma general, más ligero en el sentido de que las distintas peticiones ocupan menos, aunque hay que tener en cuenta que esto depende de los tipos a representar (ver punto siguiente).
- En el sistema WSDL se permite la definición de tipos, al igual que en IDL (el lenguaje de interfaces de CORBA). En REST el sentido de los datos comunicados no es explícito y su codificación e interpretación son cosa del programador. Se podría decir que REST requiere más documentación que un sistema basado en WSDL o IDL, puesto que parte de la información que se debe documentar viene incluida en la propia interfaz, de la que REST carece.
- Para usar una API REST se necesita un lenguaje que tenga librerías de comunicación con HTTP. Además de que la gran mayoría de lenguajes tienen librerías que ya implementan dicho protocolo, la implementación "casera" de la parte necesaria de HTTP para usar un servidor REST es bastante sencilla y se puede hacer sobre cualquier implementación de sockets. CORBA y SOAP requieren de una implementación más difícil, mayor tiempo de estudio y, en general, un proceso más largo de desarrollo.

Teniendo en cuenta los puntos anteriores, el que ha decantado la balanza ha sido el último de los mismos. La sencillez para entender y depurar un protocolo REST así como el hecho de que se base en simple HTTP nos permiten un desarrollo fácil, sin tener apenas preocupaciones desde el punto de vista del servidor.

El tipo de comunicaciones necesario es muy orientado a que el cliente haga peticiones pequeñas y el servidor responda con cantidades de información más grande. Podemos intuir que el gran esfuerzo que se tendrá que hacer a la hora de implementar las llamadas al sistema será al decodificar la información que el servidor envíe. La naturaleza de dicha información es perfecta para usar el lenguaje XML para codificar los datos. La otra alternativa principal era basarnos en el lenguaje JSON, pero no parece demasiado apropiado para enviar cantidades de información relativamente grandes como puedan ser la plantilla de un determinado jugador o

los objetos que se encuentran en una tienda.

3.2.4 Arquitectura

Sin entrar en mucho detalle, podemos resumir la arquitectura del sistema en 4 niveles distintos, aunque solo 3 se alojaran en el servidor. La cuarta capa se encuentra en el cliente, y es la librería que usa el mismo para interaccionar con el sistema servidor.

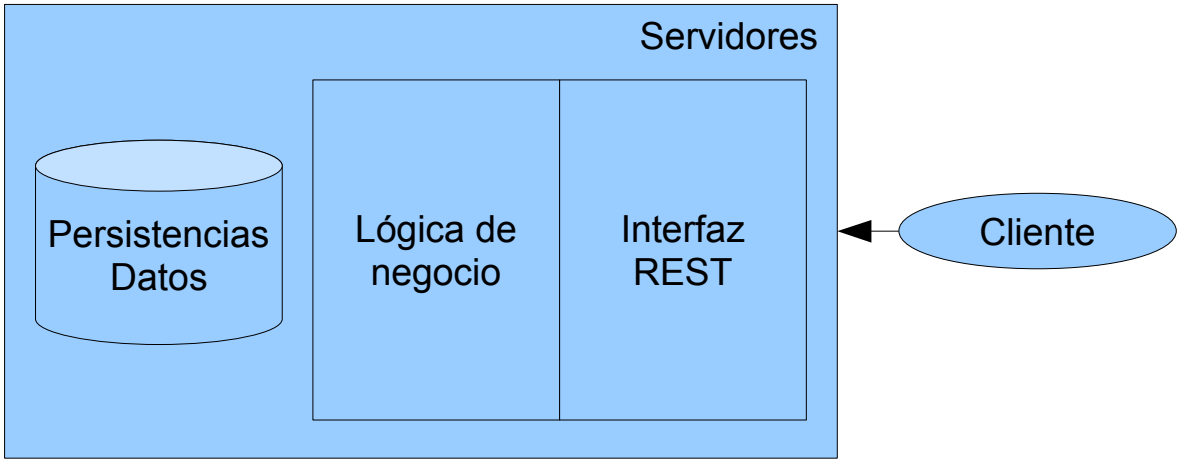

Ilustración 4: Pre-diseño de la arquitectura

Podemos ver, de izquierda a derecha los niveles de la siguiente manera:

1. Persistencias de datos. Aquí englobamos bases de datos y archivos.
2. Lógica de negocio. Dónde se encontrará todo el motor de la aplicación (tiendas, armerías, simulador, salas de partidas,...)
3. Interfaz REST. Hace de barrera o cortafuegos con el resto del sistema. Para acceder a cualquier funcionalidad del sistema hay que atravesar esta capa.
4. Librería que usa la interfaz REST y presentación de los datos que se obtienen a través de la librería.

3.2.5 Tecnologías

En cualquier ingeniería existe un principio que debe estar siempre presente: no reinventar la rueda. El tiempo que se empeña en investigar e implementar una nueva funcionalidad puede ser innecesario si otra persona ya lo ha desarrollado (y sus licencias nos permiten usarlo). El hecho de que algo ya esté hecho no implica que el coste de integración y depuración desaparezca. Voy a enunciar el principio por el que me voy a regir para decidir entre implementar funcionalidad o usar tecnologías que ya estén desarrolladas. Se usarán

tecnologías externas (desarrolladas por otras personas) cuando:

1. La tecnología implemente toda la funcionalidad requerida.
2. El proceso de integración y depuración no sea más costoso que el desarrollo de tecnología "casera".

Se establece una excepción a la regla anterior: cuando la ganancia de usar tecnología ya experimentada sea suficiente respecto de usar tecnología "casera". Por ejemplo, en un servidor HTTP es preferible usar un sistema que tenga millones de usuarios, que haya sido probado y a su vez implemente optimizaciones antes que desarrollar un servidor HTTP casero que pueda fallar al procesar un determinado tipo de peticiones o no responda cuando tiene a más de 20 clientes simultáneos.

En este proyecto se necesita tecnología externa para aportar la siguiente funcionalidad:

- Base de datos: se necesita poder guardar persistentemente una cantidad de datos relativamente pequeña: usuarios y contraseñas y estadísticas básicas de una partida ya disputada. Para hacerlo se puede usar cualquier base de datos basada en SQL.
- Persistencia de ficheros: se necesita poder guardar cantidades de datos relativamente grandes (archivos de KB – MB). El motivo para no poder guardarlos tal cual como ficheros es que si todos los ficheros se encuentran en el mismo computador pueden producir un cuello de botella. Si hay varios computadores encargados de dicha labor puede ser problemático asegurar el correcto funcionamiento del sistema (cuando replicar un archivo, que ocurre si un nodo se cae, …). Existen bases de datos NoSQL (no relacionales) que hacen perfectamente la labor de guardar estas cantidades de datos de una manera fácilmente accesible. Al igual que pedimos inequívocamente un archivo, se puede hacer lo mismo con una clave. Las sentencias "dame el archivo /archivos/blablabla.xml" y "dame el valor de la clave blablabla" podrían tener el mismo resultado.
- Localización: implementar la localización de un usuario mediante su IP es un trabajo tanto o más extenso que este proyecto completo. Hay dos restricciones que hay que tener en cuenta: el sistema que se integre debe alojarse en la máquina servidora (depender de un servicio externo puede empeorar el rendimiento) y debe saber encontrar el país en el que el usuario se encuentra con un margen de error mínimo. Si

fuera capaz de encontrar la ciudad o coordenadas geográficas del usuario ya sería idóneo.

- Alojamiento de servicios REST: es necesario usar tecnología que nos aísle del protocolo HTTP lo más posible. Si además nos aísla totalmente de la parte de sockets y los recursos reciben llamadas de una manera que nos permita abstraer las llamadas fuera de ese nivel de aplicación, mejor.
- Servicios de cifrado SSL: en algunos momentos, se podría requerir cifrar el tráfico para garantizar la seguridad del sistema y de las acciones de sus usuarios. Cuanto más transparente sea esta labor para el programador, mejor.

La resolución sobre qué tecnologías han sido finalmente usadas en el proyecto para cubrir las funcionalidades descritas se hará en la sección siguiente de la memoria: Descripción Informática (Capítulo 4).

3.2.6 Alta escalabilidad

La escalabilidad de un sistema es la capacidad que tiene el mismo para adaptarse a un incremento en su uso sin perjudicar a la calidad del servicio. Lo que se busca en el sistema desarrollado es maximizar esta capacidad de manera que, sea cual sea la cantidad de usuarios del juego, el sistema sea capaz de proporcionar al usuario una buena experiencia de juego.

Existen famosos casos de fallos en la escalabilidad de un sistema. Uno de los más conocidos se ha dado en Twitter, sitio web en el que, dado su espectacular crecimiento, el servicio era interrumpido frecuentemente con su famosa ballena voladora.

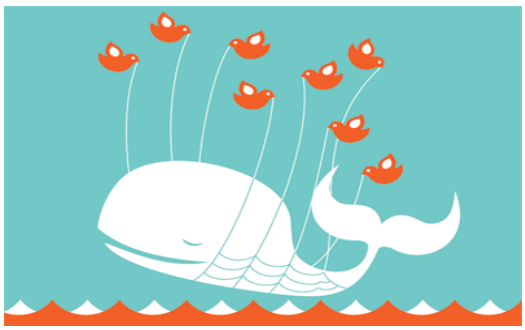

Ilustración 5: La ballena de Twitter

Este mismo problema, aunque con menos usuarios, se da también con algunos juegos de Facebook. La cantidad de jugadores puede aumentar rápidamente y si este hecho no ha sido previsto, el resultado puede ser un juego que inicialmente era entretenido convertido en un

juego que apenas reacciona correctamente a las acciones del usuario y que tiene tiempos de respuesta abismales.

El último gran ejemplo se ha dado con la salida del esperadísimo Diablo III. Una gran cantidad de jugadores ha experimentado problemas al conectarse a los servidores del juego. La solución por parte de Blizzard (la compañía desarrolladora) ha sido incluir más servidores en el sistema [12].

Para implementar la máxima escalabilidad posible se van a replicar muchos de los subsistemas. Un sistema replicado se comporta de la siguiente manera: para que el sistema falle deben fallar todas las réplicas, mientras que bastará con que una réplica pueda dar el resultado para que el sistema continúe disponible. En nuestro caso, a pesar de que no todos los subsistemas estarán inicialmente replicados, se debe dar soporte a una replicación total de los mismos.

Existe una manera muy sencilla de entender como la replicación ayuda a la escalabilidad: si tenemos 3 nodos distintos que computan el mismo servicio y cada nodo tiene una capacidad de ejecutar 10 acciones por segundo, los 3 nodos unidos serán capaces de ejecutar 30 acciones por segundo. A pesar de que la capacidad del sistema no aumenta de una manera totalmente lineal (cuantos más nodos se añaden al sistema, mayor es el coste de coordinarlos), es algo muy positivo para el sistema por otros motivos que se explican en el siguiente punto.

3.2.7 Alta tasa de supervivencia (survivability)

El término tasa de supervivencia (relacionado con el concepto en inglés “survivability”) tiene múltiples acepciones como se puede ver en este análisis [13]. La definición en la que esta sección se apoya tiene dos principios:

1) La habilidad de una red de mantener o restaurar un nivel de rendimiento aceptable ante condiciones de fallo.
2) La prevención de potenciales cortes de servicio usando técnicas preventivas.

(Traducción libre de [14])

La naturaleza del sistema no requiere una tasa continua de datos ni latencias bajas. El parámetro crítico a optimizar es la supervivencia del sistema, es decir, aumentar la tolerancia de fallos del sistema y minimizar las consecuencias de los mismos. Si la coherencia del sistema en algún momento no es garantizada, el propio sistema debería ser capaz de garantizar

que no se corrompe el último estado correcto.

Si el subsistema que permite comprar armas está caído, el usuario no debería ser capaz de comprar armas alterando el último estado guardado. A su vez, si hubiera varios nodos que computan la compra de objetos y uno de ellos falla, debería permitirse que las compras siguieran efectuándose. En este último caso, el nodo que falla debería ser desconectado antes de afectar a la coherencia del sistema.

Se ha de buscar una manera cómoda y efectiva para implementar la distribución del sistema, cuanto más eficiente mejor, y que permita la adicción y supresión de nodos en tiempo de ejecución sin penalizaciones en la coherencia del sistema.

3.2.8 Bajo acoplamiento y alta facilidad de cambio del sistema

El acoplamiento de un sistema se define como el nivel de dependencia entre los distintos componentes software del mismo. La relación entre bajo acoplamiento y alta facilidad de cambio es directa: si la facilidad de cambio de un software es la capacidad del sistema de ser cambiado o desarrollado nuevamente con poco trabajo, parece claro que reduciendo las dependencias entre los distintos componentes ya existentes en el sistema, es más sencillo hacer modificaciones o sustituciones de componentes. La razón para esto es que un cambio en un componente puede provocar cambios en el resto de componentes que lo usan, y si el acoplamiento es bajo habrá menos componentes que lo usen.

Lo idóneo en este aspecto sería que un componente pudiera ser reemplazado sin afectar al resto del sistema y permitiendo las interacciones que la versión anterior del componente permitía.

Ejemplo práctico:

> Existe un sistema que tiene un método “localizar”, que recibe una dirección IP y devuelve el país en que esa IP se encuentra registrada. La versión 1 implementa esto de una manera directa, con una base de datos de que relaciona IP y país. El sistema se modifica y en una segunda versión lo que hace es conectarse a un servicio externo que tiene una interfaz más compleja que permite averiguar la ciudad de la IP. Si la versión 2 implementa correctamente el método que tenía la versión 1, el comportamiento del resto de módulos del sistema no debería ser modificado.

Existe una abstracción dentro de la programación orientada a objetos que permite llevar a cabo esta aproximación: el concepto de interfaz, que define únicamente los métodos que las clases deberían tener, y no la manera en la que dichos métodos son ejecutados. Si se mantiene una interfaz constante en el componente la adaptabilidad del sistema aumenta de manera sustancial.

3.2.9 Seguridad

La seguridad en un sistema distribuido siempre es importante. Lo es doblemente si el sistema da un servicio al público general. En el caso de este proyecto vamos a diferenciar dos apartados donde la seguridad se debe asegurar:

1. Seguridad contra ataques al sistema (tratar de obtener tabla de usuarios del sistema, introducir armas en un usuario, alterar la lista de estadísticas,...).
2. Seguridad contra el fraude de los usuarios (intentar ver los intelectos de otros usuarios, o ejecutar acciones como si fueran ellos los que las hacen).

Existe en este proyecto además un punto clave que hay que tener en cuenta: en los servidores del sistema hay que compilar y ejecutar código fuente que ha desarrollado el usuario. Se hace obligatorio conseguir que la ejecución de dicho código sea controlada.

Ejemplo 1:

> Un usuario podría introducir código en el intelecto que le permitiera abrir un socket que se conecte con su propio ordenador y a partir de ahí enviar datos a su máquina sobre las especificaciones del sistema servidor.

Ejemplo 2:

> Un usuario podría programar en el servidor un script que haciendo uso de las librerías del sistema servidor consiguiera apagar el equipo.

Estos dos ejemplos no deberían permitirse. Un usuario que trata de hacer estas cosas debería ser sancionado por dichas acciones.

3.3 Versión demo

A partir de las especificaciones anteriores se creará una aplicación de prueba que implemente las llamadas al sistema distribuido. Esta aplicación deberá demostrar de alguna

manera la funcionalidad del sistema.

Es importante notar que la aplicación será una especie de demostración de funcionalidad. La interfaz visual de la aplicación no debería ser juzgada muy exhaustivamente.

3.3.1 Creación de una versión de navegador

Para demostrar la potencia del sistema y la viabilidad de hacer aplicaciones sencillas que lo usen, la versión de demostración deberá correr en un navegador web, intentando no tener que instalar ningún tipo de librería en el caso de ser posible.

3.3.2 Documentación

Por muchas de las ideas especificadas durante la introducción al sistema como son justicia y transparencia (Página 12), las interfaces con el sistema deben ser públicas y deben permitir a cualquier usuario hacer su propia versión de la aplicación.

3.4 Lenguaje de programación

Llegados a este punto, se ha hablado de la gran mayoría de requisitos que el sistema debe tener. Existen tres requisitos clave que el lenguaje debe tener para que nos sea válido:

(REQ1) El lenguaje debe ser muy reflexivo. Desde el propio lenguaje debería ser posible acceder a las propiedades del propio código. Por ejemplo, cuando aplicación es capaz de explorar su código para saber cuantos atributos tiene una clase.

(REQ2) El lenguaje debe ser dinámico. Mientras una aplicación se está ejecutando debería ser posible cargar módulos extra e instanciar clases que no se habían cargado al inicio.

(REQ3) El lenguaje debe facilitar la distribución de funcionalidad (mediante invocaciones remotas por ejemplo). La gran mayoría de lenguajes implementan librerías para comunicación remota, pero la dificultad de esta labor según el lenguaje escogido varía.

Con los requisitos ya detallados, existen dos lenguajes que los cumplen a la perfección: C# y Java. Ambos son lenguajes muy reflexivos, dinámicos y poseen middlewares específicos para la comunicación remota (Remoting y RMI respectivamente).

La elección por tanto se hace con un único criterio: las experiencias pasadas del desarrollador. Mientras que durante la formación universitaria ha programado una gran variedad de aplicaciones en el lenguaje Java (incluso algunas usando RMI), el desarrollador

apenas ha usado el lenguaje C#. La influencia que la experiencia pasada del desarrollador tiene en el proceso facilita un menor tiempo de desarrollo y evita en gran medida la frustración del lenguaje, es decir, cuando se conoce el modelo conceptual de una funcionalidad pero no se sabe plasmar en código fuente.

Respecto al lenguaje Java se puede encontrar más información en [15].

3.5 Modelo de proceso del proyecto

El modelo de proceso de un proyecto informático es muy importante [16]. Para un proyecto pequeño, como la mayoría de los que se realizan durante la formación universitaria, no suele marcar demasiado las diferencias. Para un proyecto grande en cambio puede ser la diferencia entre ser capaz de terminar el proyecto en buenas condiciones o terminarlo fuera de plazo (o incluso no terminarlo).

En el caso del proyecto del que este documento trata (juego y sistema distribuido), y a pesar de que solo exista un diseñador y un desarrollador, el modelo de proceso es fundamental debido a la gran envergadura del sistema global.

Además del tamaño del sistema, hay un punto extra a tener en cuenta, y es la falta de experiencia del desarrollador del proyecto con las distintas tecnologías de las que va a hacer uso.

Por los distintos motivos expuestos anteriormente, y como se detallará correctamente en el punto posterior (en la página 31), el modelo de proceso tanto de los sistemas distribuidos como del motor de juego deberá ser incremental e iterativo. En algunos casos se podrá realizar el desarrollo en paralelo de distintos componentes, mientras que en otros el desarrollo tendrá que ser secuencial.

Finalmente, se planificarán las pruebas que se consideren necesarias e interesantes al terminar cada uno de los módulos y basándose en las interfaces descritas en cada uno de los subsistemas.

4 Descripción informática

Una vez superada y comprendida la sección de objetivos, el lector debería tener en mente una idea global de lo que se pide del sistema. En esta sección de descripción informática se explicará cómo se han desarrollado los objetivos anteriormente explicados. Se recomienda al lector que retroceda a la sección del diseño correspondiente en el caso de no entender alguno de los puntos de la solución propuesta.

4.1 Planificación del proyecto

La planificación de este proyecto es un tanto peculiar. Puesto que el autor del mismo además de diseñador también asume el rol de creador, los requisitos del proyecto dependen también de él. Se entiende entonces que la solución propuesta por el autor es constantemente evaluada y revisada para adecuarse a los requisitos.

Otro punto particular del proyecto es que se divide en dos grandes fases que pueden ser hechas por separado sin ningún problema, pero que debido al requisito de existir un único desarrollador, se han secuencializado.

El modelo de proceso seguido es secuencial en lo que al sistema completo se refiere, pero cada uno de los dos subsistemas tienen un modelo iterativo e incremental.

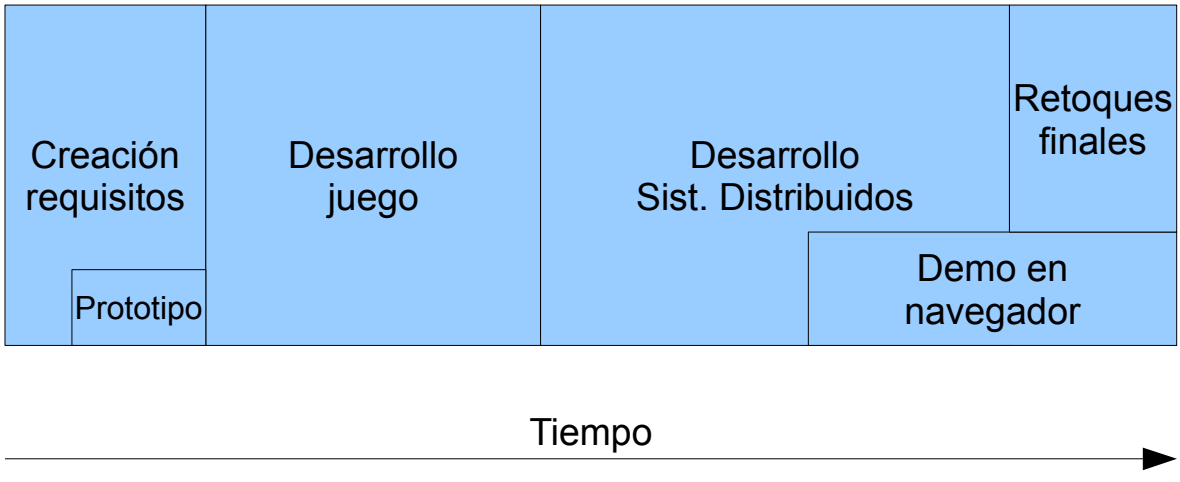

Ilustración 6: Modelo de proceso del sistema completo

Ahora se van a describir las planificaciones de los dos proyectos de manera individual:

1) Planificación del juego (modelo iterativo e incremental)

1. Puesto que el software es relativamente novedoso, inicialmente se planifica un pequeño prototipo para comprobar el concepto. Dicho prototipo debe contar con escasas acciones. Es importante que el prototipo se realice de manera rápida y no tiene en la eficiencia una prioridad.
2. Tras el desarrollo de dicho prototipo se comenzará el desarrollo del sistema global. En primer lugar se debe elaborar un diseño básico de lo que será la arquitectura del sistema.
3. Después se realizarán unos esqueletos básicos que permitan programar el sistema sin errores de compilación y a su vez aporten adaptabilidad al sistema.
4. Se elaborará una lista ordenada de acciones, consecuencias, clases y métodos del motor según lo necesarios que sean para el sistema. La idea de la lista es priorizar los esfuerzos del programador de manera que, usando un símil con la arquitectura se empiece la casa por los cimientos.
5. Se programan revisiones cada 4 días al inicio y posteriormente cada semana. La misión de estas revisiones es revisar que la implementación sigue a un ritmo continuo. Durante la ejecución de las revisiones se debe ejecutar el motor al completo comprobando que todo lo implementado casa correctamente. Nota: esto no implica que durante el desarrollo del punto anterior no se hagan pruebas, es solo que dichas pruebas se realizan sobre los componentes añadidos y no sobre todo el sistema.
6. Una vez se haya cubierto toda la funcionalidad deseada, el sistema deberá ser probado por otros usuarios. El objetivo de estas pruebas es localizar cuál es el contenido que debe estar presente en los manuales o tutoriales que se distribuyan.
7. Deberá elaborarse la documentación para los usuarios.
8. Deberá juzgarse dicha documentación (en caso de no ser correcta, se ampliará y volverá a juzgarse).
9. Puesto que el motor total del sistema será público, se elaborarán unos documentos sencillos que permitan a un programador medio entender el funcionamiento del motor de juego.

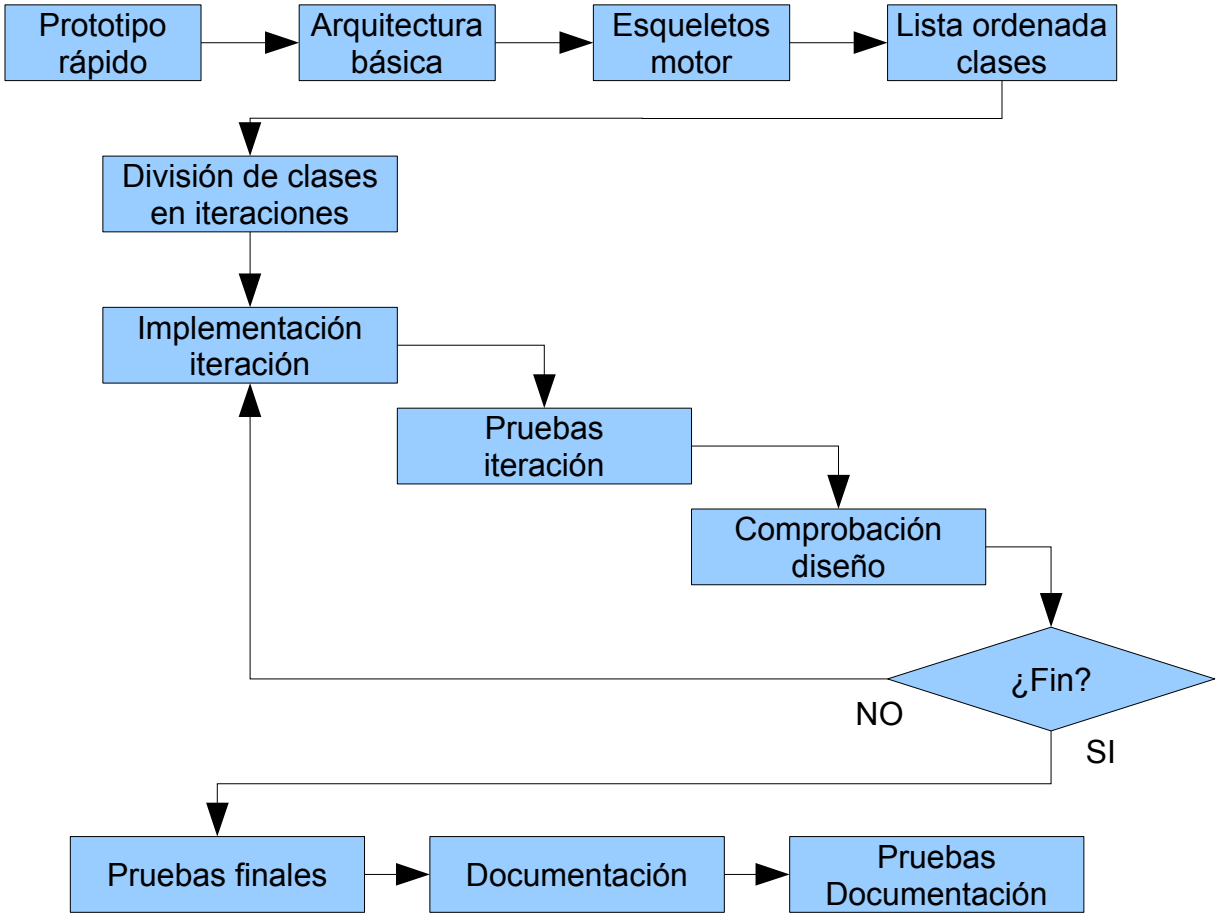

Ilustración 7: Modelo de proceso del juego

2) Planificación del sistema distribuido (modelo iterativo e incremental)

1. Puesto que para comenzar las pruebas del sistema distribuido habrá ciertas clases que se necesiten casi siempre, se debe trabajar inicialmente en una descomposición óptima de toda la funcionalidad en componentes distintos. Además se ordenará la programación de dichos componentes.
2. Para facilitar el desarrollo de todas las clases, las interfaces de los sistemas se desarrollan en este punto, de manera que cuando un sistema requiera de otro, sabe de antemano como va a usarlo. Igualmente también sabe lo que otros sistemas van a requerir de él.
3. Junto a las interfaces, se genera una arquitectura básica del sistema. Con cada iteración esta arquitectura será mejorada y ampliada.
4. Se elaborará una lista que asigne cada componente a una iteración. Varios componentes podrían encontrarse en la misma iteración.
5. Se desarrollarán los distintos componentes. Han de ser probados individualmente

para asegurar que funcionan de manera correcta antes de ser puestos en contacto con el resto.

6. Al concluir el desarrollo de todos los componentes de la iteración, todos ellos se adaptan al sistema global. Se comprueba que el sistema distribuido correspondiente a la iteración funciona correctamente. En este punto se revisa de nuevo la planificación de las iteraciones. La adaptabilidad del proceso debe ser máxima.
7. Se incrementa la aplicación de demostración con las llamadas oportunas a la nueva funcionalidad.
8. Se revisa que los requisitos inicialmente diseñados se van cumpliendo. Se revisa la arquitectura del sistema y se actualiza en caso de creerse necesario.
9. Al terminar todas las iteraciones se elabora la documentación de la API REST desarrollada.

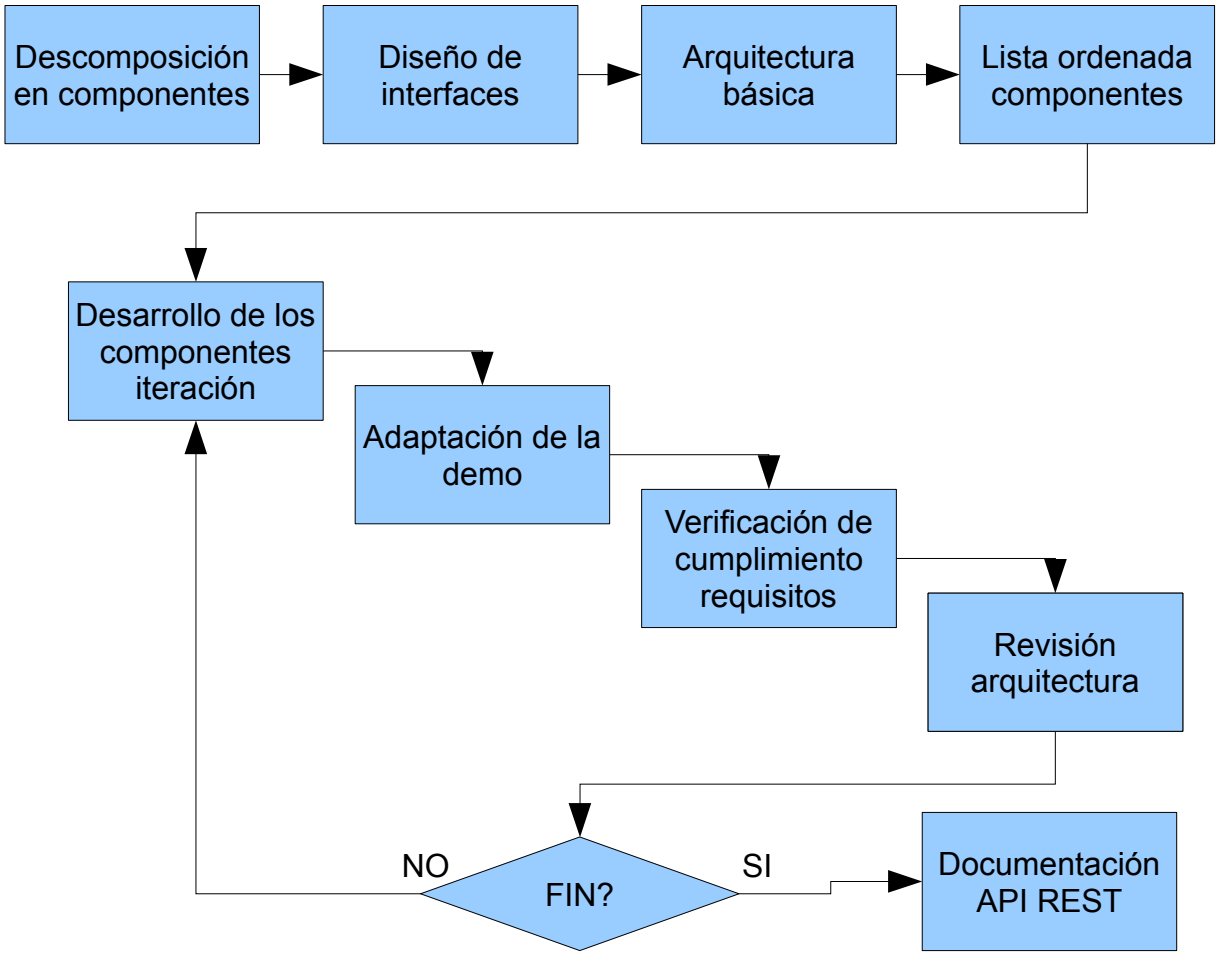

Ilustración 8: Modelo de proceso del sistema distribuido

Teniendo en cuenta que cada componente será relativamente pequeño, el modelo de proceso de cada uno de los componentes será muy simple. Típicamente será un modelo iterativo en el que se desarrolla y prueba funcionalidad varias veces antes de dar el desarrollo del componente por terminado.

Es importante que el nodo quede aislado de los demás mediante funciones stub (funciones temporales usadas únicamente para permitir una correcta compilación) en sus primeras iteraciones. Será al final del desarrollo cuando ya se intente conectar al resto de sistemas distribuidos.

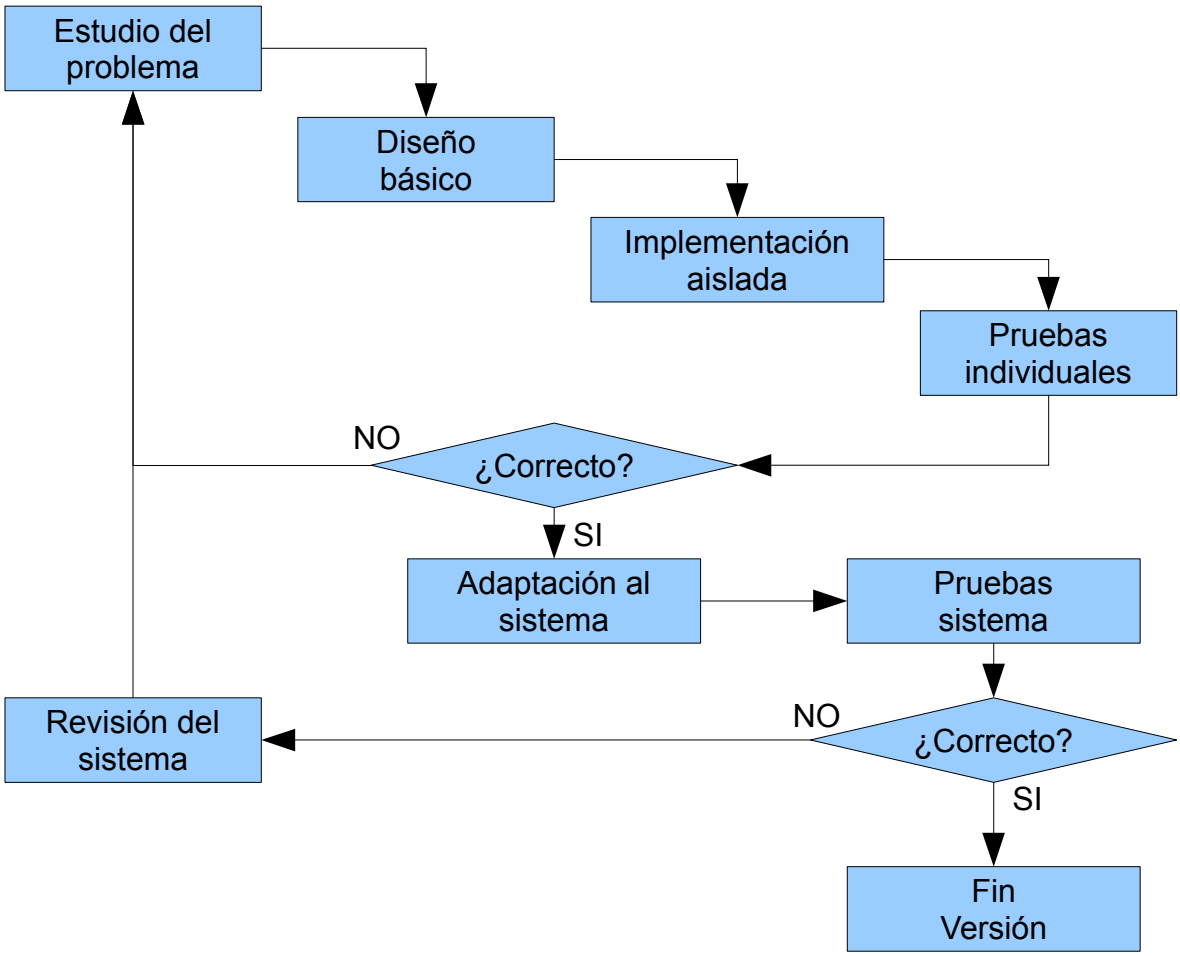

Ilustración 9: Modelo de proceso de cada componente del sistema distribuido

4.2 Diseño del sistema de juego

Una vez terminado el prototipo, se pasó a diseñar el sistema de juego. Es importante notar que entre el diseño inicial y la implementación final hubo muchas modificaciones. Por más bueno que sea el diseño, hasta que no se empieza a implementar, no se detectan métodos que

faltan, métodos invisibles, caminos más sencillos, etc.

La explicación que se va a dar en esta sección se ciñe a la implementación final. Primero se van a explicar las clases más importantes y después se expondrán algunos diagramas sobre cómo esas clases se relacionan.

Para obtener acceso al repositorio de código que contiene la implementación de versiones preliminares del diseño, contacten con el autor (ver Sobre el autor).

4.2.1 Clases importantes

Clase Posición: representa un conjunto de coordenadas. En esta implementación tiene 2 dimensiones.

Clase Angulo: representa un ángulo geométrico.

Clase abstracta Colisionable: representa a una cosa que puede interrumpir la visión de una unidad o un disparo. Caja y Entidad heredan de ella.

Clase Caja: representa una forma geométrica en el mapa. Dicha forma geométrica es intransitable. Marca las paredes del mapa.

Clase Estado: representa una tabla de propiedades. No fija ni el nombre de las propiedades ni el tipo de las mismas. Será usado por Soldado, Arma y Arrojable.

Clase Entidad: representa a un objeto.

Clase Soldado: hereda de Entidad. Representa a un objeto con intelecto. Tiene un Estado con unas características por defecto que pueden ser modificadas. Las características fueron estudiadas en la sección de diseño de funcionalidad (página 14).

Clase EntidadArrojable: hereda de Soldado. Es un truco usado para representar a los explosivos que tardan rondas en explotar. Se les pone un intelecto especial que provoca una AccionExplosion al cabo de N turnos de espera.

Clase Arma: representa a un arma con sus propiedades específicas. Las características fueron estudiadas en la sección de diseño de funcionalidad (página 15).

Clase Arrojable: representa a un arrojable con sus propiedades. Es importante notar que en el momento en que un Arrojable (por ejemplo una granada) es lanzado, se crea una EntidadArrojable en el motor de juego y deja de depender del jugador.

Las clases anteriores se pueden resumir en el siguiente diagrama de relación:

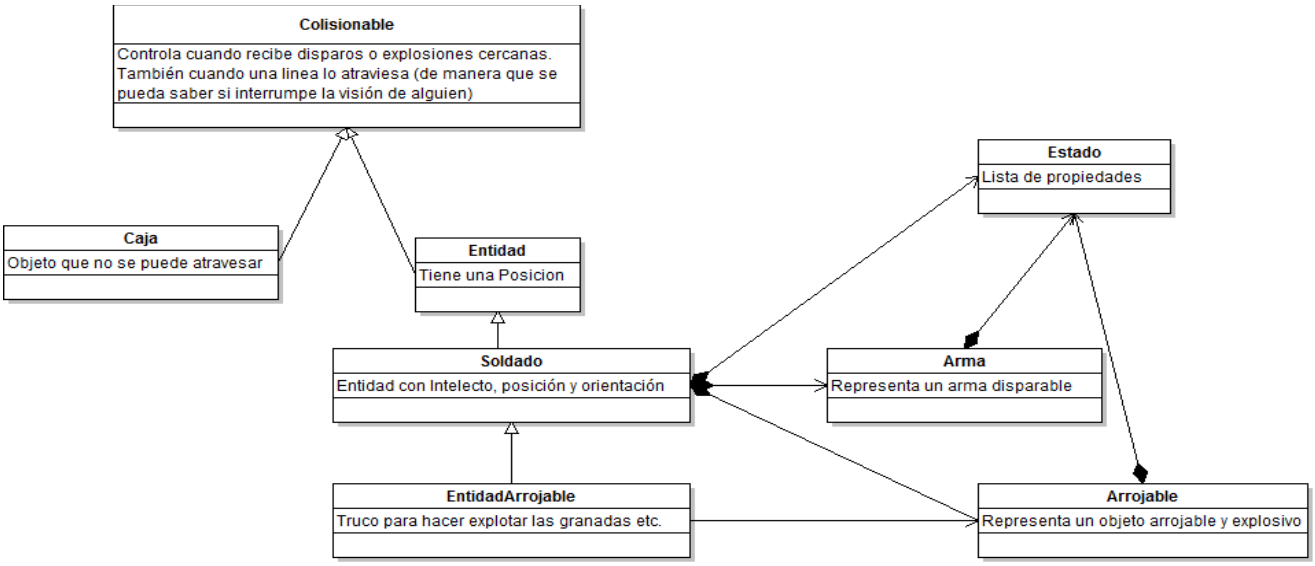

Ilustración 10: Diagrama UML de los objetos colisionables

Clase abstracta Intelecto: representa conceptualmente el motor de decisiones de un soldado. Tiene un método "decidir" que es llamado una vez por turno. Este método tiene por resultado una instancia que herede de la clase Acción.

Clase abstracta Acción: representa conceptualmente a una acción. Tiene un método abstracto "ejecutar" tal que dicha acción se procesa y se obtienen de ella las distintas consecuencias que produce.

Clases que implementan Acción: AccionDisparar, AccionGirar, AccionMoverse, AccionRecargar,...

Clase abstracta Consecuencia: representa una consecuencia de una acción. Es directamente interpretable por el motor de juego, de modo que lo único que produce cambios en el entorno es las consecuencias. Las acciones por tanto necesitan producir consecuencias para producir cambios.

Clases que implementan Consecuencia: ConsDisparo, ConsMover, ConsGirar, ConsSonido, ConsRecargar,...

Clase Vista: representa el sistema de visión de una unidad. Aglutina todas las entidades que una unidad puede ver en una posición y mirando en una determinada orientación.

Clase Terreno: representa la percepción del terreno que la unidad tiene. El terreno es, en esta implementación, un conjunto de cajas. Es decir, representa que puntos son transitables y

cuales no.

Clase Oído: representa la audición de sonidos por parte de la unidad.

Clase Sonido: contiene una intensidad del sonido (la intensidad de un disparo es mayor que la de el sonido que se emite al andar) y la posición dónde ese sonido se ha producido.

Clase Radio: representa las interacciones por radio del soldado. Es donde recibe los mensajes.

Clase Frecuencia: representa una abstracción dónde se envían mensajes. Una frecuencia puede recibir varios mensajes. Un mensaje solo puede enviarse por una frecuencia.

Clase abstracta Mensaje: representa lo que los soldados se comunican a través de la radio. Existen mensajes predefinidos y otros mensajes con tamaño máximo que pueden ser diseñados por los usuarios.

Las clases enumeradas hasta ahora pueden verse en el siguiente diagrama:

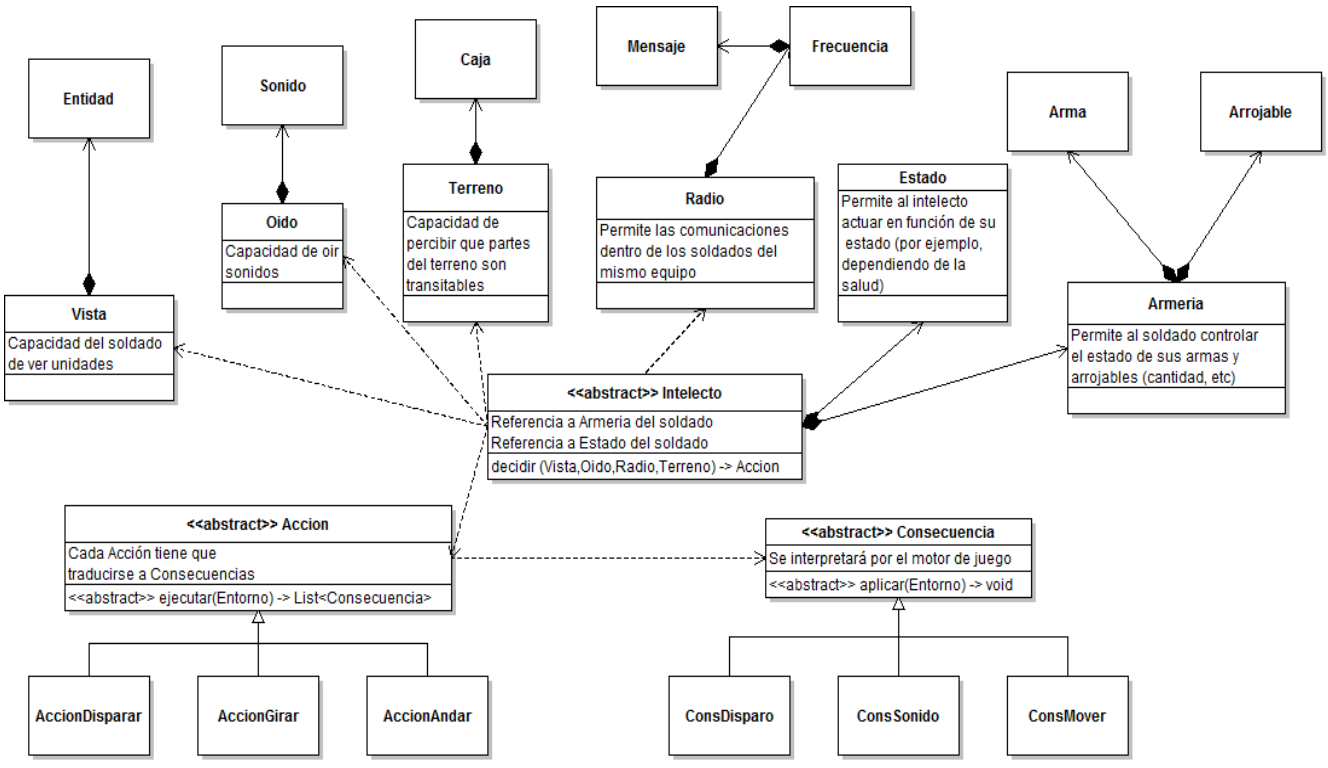

Ilustración 11: Diagrama UML del intelecto y sus interfaces

Clase abstracta Objetivo: representa una condición según la cual el juego termina. Tiene un método abstracto que evalúa en el entorno decidiendo si la partida acaba o no. En caso de terminar determina qué equipo es el ganador.

Clases que implementan Objetivo: LimiteTurnos, AniquilacionEnemigo, BanderaCapturada,...

Interfaz PanelGrafico: permite al motor de juego abstraerse de sobre qué plataforma se está ejecutando a la hora de pintar el estado de la partida.

Interfaz Entorno: es una representación abstracta del motor de juego. El motor de juego es el soporte que el resto de elementos del juego necesitan. Este soporte tendrá métodos para configurar la partida, para ejecutarla y para finalizarla. A continuación se exponen los métodos que contiene:

Métodos de configuración de partida:

```
void insertarCaja(_Caja c);
void insertarEntidad(_Entidad e);
void eliminarEntidad(_Entidad e);
void insertarSoldado(_Soldado s);
Hashtable<String,List<_Soldado>> getTablaSoldados();
void insertarObjetivo(Objetivo objetivo);
```

Métodos de ejecución de partida:

a) Control de colisiones

```
List<ColisionablePos> hayColision(Posicion inicial, Posicion fin);
Colisionable colisionMasCercana(Colisionable c, Posicion
ini,List<ColisionablePos> l);
List<Colisionable> dameColisionesArea(Posicion inicial, float area);
```

b) Control de percepción de los soldados

```
List<_Entidad> dameEntidades(Posicion orig,Angulo izq,Angulo der,float
dist); //Solo devuelve entidades, no soldados
List<_Soldado> dameSoldados(_Soldado orig,Angulo izq,Angulo der,float
dist);
List<Sonido> dameSonidos(_Soldado s, float dist);
List<Mensaje> dameMensajes(Frecuencia frec);
List<_Caja> dameCajas(Posicion orig, float dist, Angulo izq, Angulo
der,float distlarga);
void insertarSonido(_Sonido s, Posicion p, _Entidad soldado);
void insertarMensaje(Mensaje m, Frecuencia f);
List<_Caja> getCajas();
```

c) Motor de juego

```
void moverEntidad(_Entidad ent, Posicion destino);
int getNumTurnos();
```

```
void empezarTurno();
List<Accion> obtenerAcciones();
List<Consecuencia> procesarAcciones(List<Accion> listaAcc);
void procesarConsecuencias(List<Consecuencia> listaCons);
void soldadoMuerto(_Soldado _Soldado,String asesino);
```

d) Control gráfico

```
void pintame(PanelGrafico p);
void pintarEquipo(PanelGrafico p,String equipo);
Posicion getTamano();
```

Métodos de finalización de partida:

```
String finDeJuego();
String explicacionFin();
Hashtable<String, Integer> getIniciales();
Hashtable<String, Integer> getMatados();
```

Finalmente se muestra un diagrama sobre cómo se relaciona con el resto de clases explicadas:

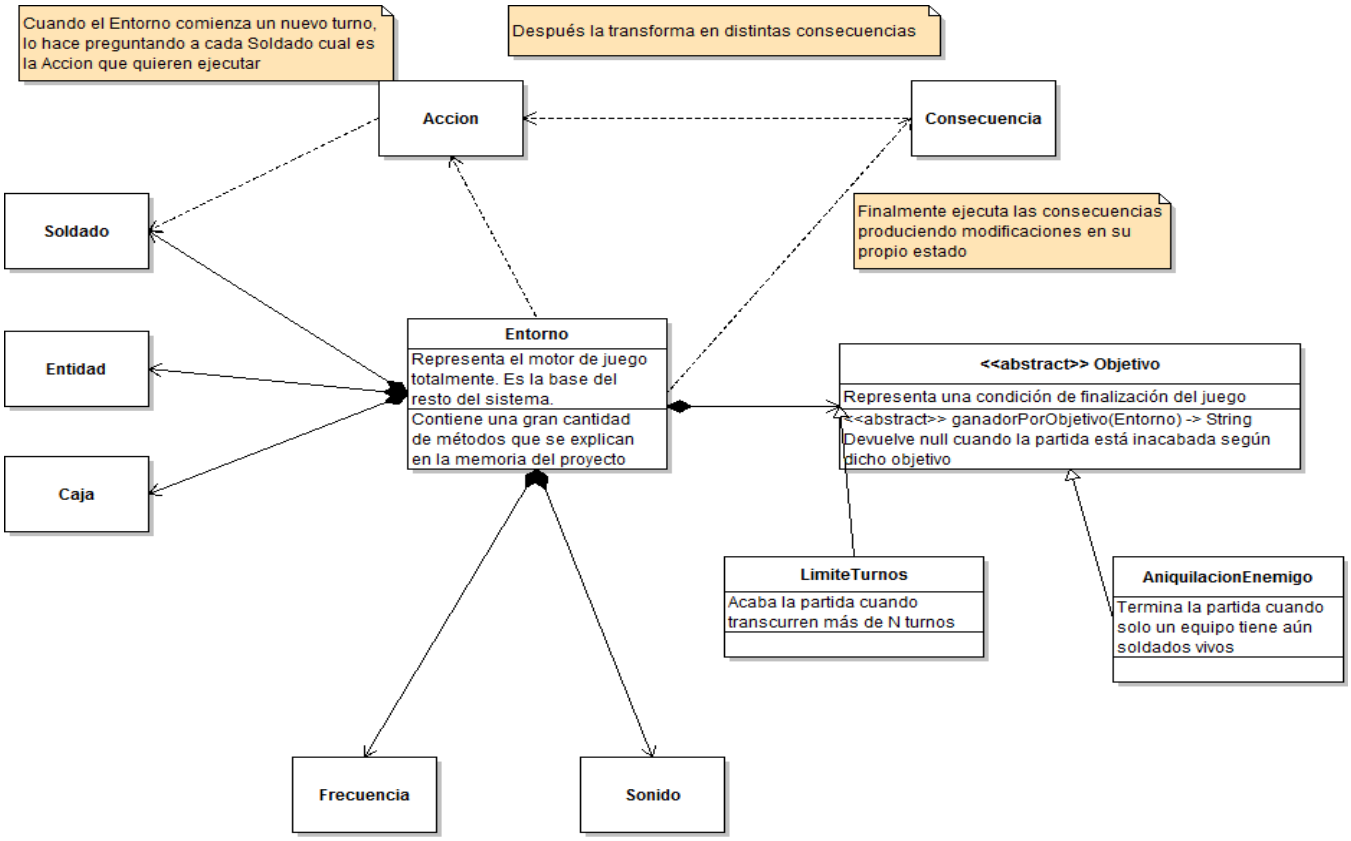

Ilustración 12: Diagrama UML de las interacciones del motor de juego

4.2.2 Uso de patrones de diseño

Para facilitar que otras personas puedan entender el código fuente de la solución se ha intentado usar el mayor número de patrones de diseño posible. Un patrón de diseño es una estructura software cuyo uso está indicado para resolver una situación concreta. Los nombres de los patrones y sus comportamientos han sido extraídos de [17].

Los patrones que se han usado más constantemente han sido varios: los patrones *Builder* y *Factory* (con sus respectivas variantes abstractas que aportan un menor acoplamiento) facilitan la labor de instanciar clases de distintos tipos. Se han usado en la creación de armas y unidades. El patrón *Command* (Comando) es el patrón fundamental del sistema. Es usado en la ejecución de acciones, consecuencias y en la comprobación de las clases Objetivo. Para explicar su utilidad voy a centrarme en las acciones. Se tiene una lista de instancias de acciones (referenciadas con el tipo de la superclase Acción). Habrá acciones de distintos tipos, y cada una de ellas se ejecuta de distinta manera, pero todas tienen en común una cosa: todas se ejecutan. Cuando se recorra la lista intentando ejecutar todas las acciones, cada acción tendrá un resultado distinto.

En la siguiente imagen se puede ver un pequeño ejemplo de uso del patrón. Todas las acciones tienen en común que tienen un método “ejecutar”, que para cada acción se ejecuta de la manera que se puede ver a la derecha de la imagen.

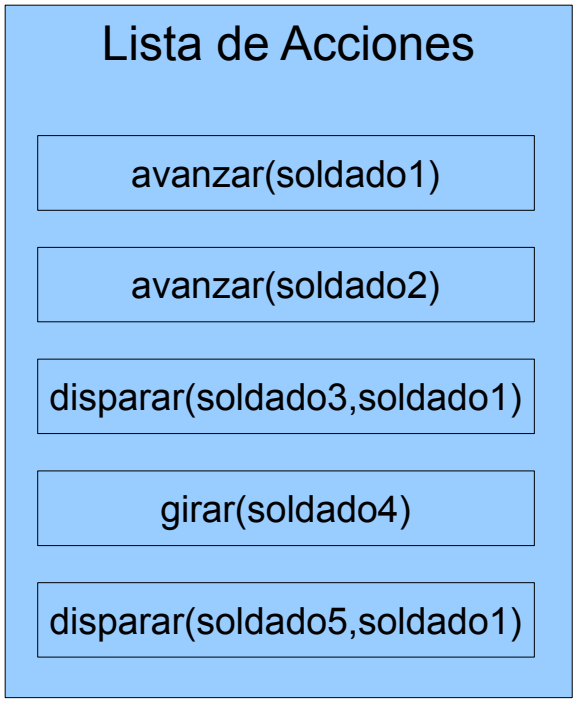

Ilustración 13: Explicación del patrón Comando

Existen en cualquier caso otros patrones que han cobrado mucha importancia en el diseño a pesar de haber sido menos usados:

El patrón *Facade* (Fachada) se usa en el motor de juego. La interfaz Entorno representa una gran fachada que engloba toda la funcionalidad del motor. Los detalles por debajo no interesan a quien necesita usarlo.

El patrón *Bridge* (Puente) se usa dentro del motor para la representación gráfica. Existe una interfaz PanelGrafico genérica que no depende de ninguna plataforma. Contiene los métodos gráficos más básicos (fijar un color, pintar una linea, un circulo, etc.). Dichos métodos son implementados en clases específicas como podrían ser: PanelJavaWindows o PanelAndroid.

Existe un último patrón que no se encuentra en el libro anteriormente citado ([17]). El autor del proyecto lo ha bautizado como el patrón *Visor*. El problema al que este patrón se enfrenta es: queremos que desde unos determinados paquetes no se pueda ver una determinada información pero desde otros sí. En lenguajes como C++ esto se podría solucionar con el concepto de clase amiga, pero Java no lo implementa. La solución es crear una estructura básica que haga de puente para los métodos que nos interesen e impida el paso para los que no.

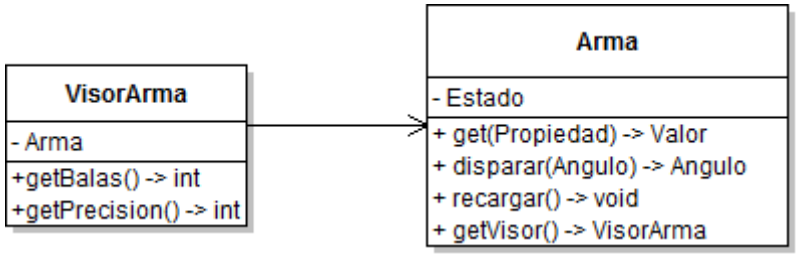

Ilustración 14: Explicación del patrón Visor

Como se puede ver en el diagrama, el comportamiento del visor recuerda bastante al patrón *Adapter* (Adaptador). Las interfaces en este caso son inicialmente incompatibles. El detalle fundamental a tener en cuenta es que desde la clase VisorArma no se puede llegar a obtener una referencia a Arma y por tanto si se tiene una referencia a VisorArma no se puede llegar a ejecutar en ningún caso los métodos disparar ni recargar.

Todas las clases que son recibidas por Intelecto siguen este patrón. Por tanto, se controla qué propiedades puede ver el Intelecto y cuales no. En el caso del visor de soldados, solo tiene métodos para observar el tamaño del soldado, hacia dónde está orientado y su posición. Finalmente se crea también un tipo de lista especial llamado ListaLectora, que actúa como un

visor de una List. De esa manera se restringe al usuario la posibilidad de añadir o borrar elementos de una lista.

4.2.3 Compilación y enlazado

Cuando el usuario quiere jugar programa un Intelecto. Dicho Intelecto tiene que ser compilado y enlazado mientras el juego se ejecuta. No podría ser nada jugable si cada vez que se quisiera jugar una partida hubiera que recompilar toda la aplicación para añadir nuevos intelectos. Por tanto se tiene que instanciar al compilador de Java en tiempo de ejecución. Con una simple instrucción (y enlazando un archivo tools.jar al proyecto) se puede obtener un objeto JavaCompiler.

```
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
```

Dicho objeto permite compilar archivos de texto con una llamada a su método "run" pasándole como parámetros los archivos a compilar. El resultado de dicho método será un entero que nos indicará si la compilación falló o fue correcta. Si fue correcta deberían haberse creado una serie de archivos de extensión class. Ahora solo nos faltaría cargarlos en nuestra aplicación.

El lenguaje Java, al ser un lenguaje con altas capacidades de reflexión, permite fácilmente cargar clases en tiempo de ejecución. Primero se instancia un ClassLoader, se le indica la ruta dónde encontrará las clases dinámicas y finalmente se obtiene un URLClassLoader. El URLClassLoader tiene un método "loadClass(String classname)" que carga una clase. Dicha clase puede ser instanciada con "getInstance()".

El proceso de enlazado dinámico se puede ver en la siguiente función Java:

```
Intelecto getIntelecto(String nombre) {
     ClassLoader parent=Intelecto.class.getClassLoader();
     File dir=new File("IntelectosDescargados");
     URLClassLoader loader = null;
     try {
          loader = new URLClassLoader(new URL[]{dir.toURL()},parent);
     } catch (MalformedURLException e1) {...}

     Class clase = null;
     try {
          clase = loader.loadClass(nombre);
     } catch (ClassNotFoundException e1) {...}

     Intelecto i=null;
```

```
    try{
         i=(Intelecto) clase.newInstance();
    }
    catch (InstantiationException e1) {...}
    catch (IllegalAccessException e1) {…}
    return i;
}
```

A pesar de todo, hay ciertos mecanismos que el usuario no debería utilizar. Uno de ellos es, por ejemplo, usar campos estáticos en su clase Intelecto. Dichos campos podrían ser accedidos por todos los intelectos, y serían una forma sencilla y fraudulenta de comunicación entre soldados. El método usado para detectar dichos atributos estáticos se ejecuta una vez la clase ha sido cargada. En este momento se usa la reflexión de Java para obtener los atributos de la clase, y si alguno es estático se lanza un mensaje de error y se avisa al usuario.

4.2.4 Seguridad

Además de las acciones fraudulentas como el uso de atributos estáticos, el usuario tiene la posibilidad de ejecutar acciones maliciosas. Recordemos que estas clases se compilan y ejecutan con los mismos derechos que lo hace el motor, y esto no es nada bueno. Este comportamiento debe cambiarse de manera que el código del usuario no pueda realizar determinadas acciones (abrir sockets, realizar llamadas al sistema,...).

Java implementa en su núcleo un controlador de permisos que es consultado cada vez que una acción que requiera permisos va a ejecutarse. Lo único que habría que hacer es implementar una clase que herede de SecurityManager y que adopte el comportamiento que se considere necesario para el juego. Después ese SecurityManager se le pasará al sistema de manera que cada vez que se vaya a ejecutar una acción que requiera permisos, será consultado.

De entre todas las llamadas que el SecurityManager es capaz de controlar [18], las que son necesarias para el juego son las siguientes:

- Control de sockets (conexión a otros puntos y escucha)
- Control de hilos (no permitir el acceso a otros hilos que no tengan que ver con el suyo)
- Control del ClassLoader (no permitir la carga dinámica de otras clases)
- Control del sistema de archivos (no permitir borrado, edición o lectura de archivos)
- Control de las llamadas al sistema (serán prohibidas)

- Control del acceso a paquetes (no se podrá acceder a las librerías del sistema)
- Control de ventanas (no se podrá alterar el sistema de ventanas)

Es importante notar que dichas restricciones se aplican solo al código de los usuarios, y no se aplican al código del motor de juego. Por tanto el SecurityManager se activará inmediatamente antes de ejecutar código de los intelectos y se desactivará después.

La implementación de esta clase se adjunta a este proyecto como apéndice en la página 95.

4.2.5 Manejo de XML

Como se explicó en la sección de objetivos, las interfaces del programa en cuanto a la configuración de las partidas se ofrecerán en un sencillo lenguaje XML.

Para fijar el estado inicial de la partida se necesita la siguiente información:

- Posiciones y orientaciones iniciales de los soldados.
- Armas y arrojables que llevan los soldados equipados.
- Nombre del intelecto que cada soldado usa.
- Propiedades que definen a los soldados (tamaño, precisión, velocidad,...).
- Equipos en los que se encuentran los soldados.
- Composición del terreno.

La manera más sencilla de entender como se compone el archivo XML es usando un documento estructural como puedan ser XMLSchema o DTD. Puesto que son lenguajes que requieren el aprendizaje por parte del lector, en vez de usar un documento esquemático se va a usar un ejemplo de archivo y se incluirá como apéndice en la página 100.

Para procesar el archivo XML de configuración se ha usado la librería JDOM. Dicha librería implementa, además de un parser de XML, el lenguaje de consultas XPATH.

Mientras se lee el archivo XML se crea el objeto Entorno. Se van añadiendo soldados, armas, arrojables y cajas en el Entorno gracias a la interfaz que ofrece (los cuales se explican en la página 39).

4.2.6 Herramientas

Se ha desarrollado una versión visual del simulador que permite ver a los soldados como

círculos moviéndose en el mapa. También se puede ver que acción están realizando con un código geométrico. Finalmente, con el programa, se puede controlar para ver únicamente lo que los soldados de uno de los equipos perciben.

Finalmente no se ha considerado necesario implementar la herramienta de edición de intelectos “Drag and Drop” en esta versión del juego. La cantidad de trabajo que requiere es suficientemente grande como para esperar a la siguiente versión para hacerla.

Lo que si se ha adjuntado es varios intelectos de prueba (entre los que destacan un intelecto de “zombie” -página 103- y un intelecto “antizombie” -página 105-) y varios escenarios distintos para probar inicialmente la calidad de nuestros intelectos. También una pequeña guía que permita a los escasos usuarios de las primeras pruebas utilizar la aplicación.

En cuanto a la documentación general del código implementado, se facilitará una copia de lo que se ha explicado en la primera parte de esta sección (Página 36). El autor del proyecto considera que una documentación mucho más exhaustiva conllevaría un trabajo demasiado intenso para lo relativamente poco que aportaría al programador que intenta leerlo. Con las guías que se ofrecen en este documento, el lector debería tener una visión general de como funciona el sistema. Para más información, el código fuente de la aplicación se encuentra disponible en http://dl.dropbox.com/u/51409718/ProyectoJairo.rar y se adjunta como anexo en formato CD (consulten al autor del proyecto en caso de producirse algún fallo, ver Sobre el Autor).

4.3 Diseño del sistema distribuido

La primera labor, una vez comprendida toda la funcionalidad que hay que implementar, es dividir dicha funcionalidad en distintos componentes. Dicha labor se puede hacer a varios niveles.

En el nivel más alto, todo el sistema distribuido se puede ver como un único componente que presta servicio a los usuarios.

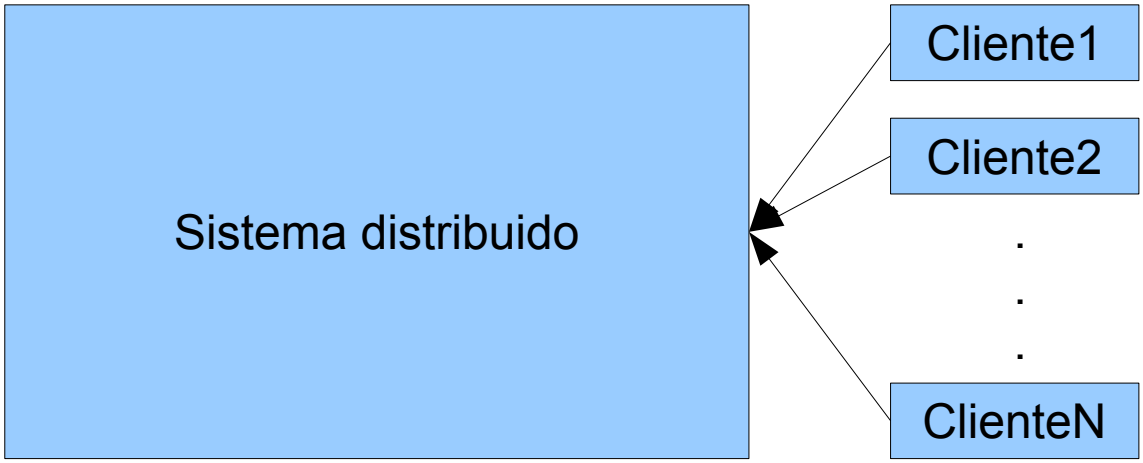

Ilustración 15: Arquitectura a alto nivel

Un nivel por debajo, la descomposición del sistema se hace en 3 capas distintas:

La más externa es la interfaz del sistema. En el caso del sistema del proyecto, la interfaz que se proporciona es una API REST, lo que implica que hay ciertos recursos programados, y habrá un servidor HTTP escuchando las llamadas que se produzcan a las URL de dichos recursos.

La siguiente capa representa a la lógica de negocio. Es en esta capa donde se ejecutan verdaderamente las llamadas que los usuarios hacen. El mayor esfuerzo a la hora de descomponer la funcionalidad del sistema se hará por tanto en esta capa.

La última capa es la de las persistencias. Se ocupa de los datos como tal, tanto de almacenarlos como de servirlos a los servicios que los necesiten. Desde la lógica de negocio se debe evitar recurrir muy a menudo a dicha última capa por motivos de eficiencia.

La arquitectura se puede ver en la siguiente imagen:

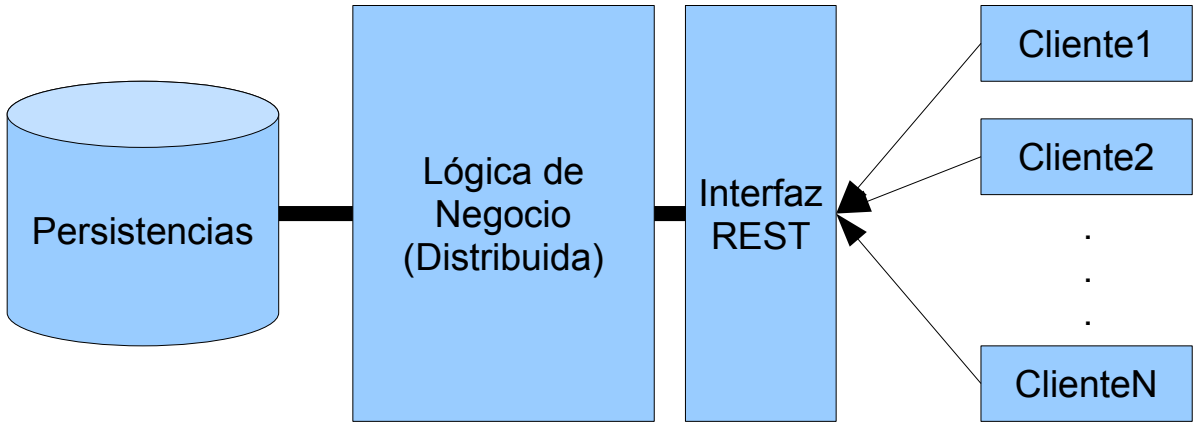

Ilustración 16: Arquitectura a nivel intermedio

Se va a hacer aún una división más de la funcionalidad del sistema. En este caso se va a

tratar de detectar los componentes que podrían implementarse de maneras individuales. El criterio que se busca para determinar cómo hacer la división es: que la funcionalidad del componente sea homogénea (que no exista un componente con funciones que abarquen varios puntos distintos) y que sea completa (que sea capaz de realizar una funcionalidad por sí misma).

Siguiendo el criterio anterior se han identificado los siguientes componentes:

- Autorizaciones: Se ocupa de validar las sesiones de los usuarios. Cuando un usuario se intente conectar, se verificará que su nombre y contraseña son correctos. Una vez verificado se creará una llave de sesión que el usuario tendrá que transmitir cada vez que quiera hacer alguna acción restringida.
- Control de Intelectos: Se ocupa de guardar y servir los intelectos de los jugadores. Un intelecto es, en esta versión, una clase en programada en Java que hereda de la clase Intelecto. Se trata por tanto de un archivo de texto.
- Control de Reportes: Se ocupa de guardar y servir los reportes de las partidas. El reporte de una partida permite reproducir íntegramente lo que ha pasado en esta. Es un fichero de texto con formato XML.
- Localización: Se ocupa de detectar de dónde viene un usuario sabiendo su IP.
- Estadísticas: Se encarga de elaborar un ranking con la información de los resultados que haya en la base de datos.
- Control de Partidas: Se encarga de alojar las distintas salas en las que los usuarios pueden jugar. Permite a los usuarios entrar a una sala, salir de ella, seleccionar con que soldados participar, etc.
- Simulador: Implementa el motor de juego. Simula una partida basándose en la configuración de los soldados que se le transmita. Una vez terminada guarda los resultados y el reporte de la misma para que cualquier usuario pueda verlo.
- Distribuidor de Partidas: Cuando un usuario quiere empezar la partida que se haya preparado en una sala, este distribuidor se encarga de decidir en qué simulador se ejecutará.
- Configuración de Soldados: Permite al usuario comprar unidades, armas y objetos. Igualmente permite asignar a las unidades las armas y objetos comprados.

- Persistencia de Usuarios: Controla el acceso a la base de datos de usuarios. En la base de datos se almacenan usuario y contraseña entre otros datos.
- Persistencia de Partidas: Controla el acceso a la base de datos de partidas. En la base de datos se almacenan los resultados de las partidas disputadas (quién gana, cuántos soldados han muerto, etc).

La siguiente misión era ordenar los componentes en una lista para decidir cuándo se implementaba cada uno de ellos.

La lista quedó de la siguiente manera:

1. Persistencia de Usuarios y Partidas, Autorizaciones y Configuración de Soldados.
2. Localización, Control de Intelectos y Control de Reportes.
3. Control de Partidas
4. Simulador y Distribuidor de Partidas
5. Estadísticas

Además de la implementación de dichos componentes, antes de hacerlo habría que preparar y decidir que tecnología sería usada para hacer de interfaz REST. Se buscaba facilitar la tarea de alojar recursos REST.

Existe una opción que facilita muchísimo el desarrollo y el desplegado de recursos REST. Dicha opción es usar Netbeans junto con Glassfish. Tiene proyectos prediseñados que funcionan desde el principio, y se trabaja fundamentalmente en dotar a dichos proyectos con la lógica que interese para el resto del proyecto. La implementación de recursos REST en Netbeans es facilitada por la librería Jersey.

Existe una gran desventaja al implementar código de esta manera: el programador pierde el control de parte del desarrollo, y eso puede hacer el proceso de pruebas más difícil (al programador puede costarle entender que está ocurriendo en cada momento). También se complica la configuración del entorno, pero la popularidad de Netbeans y Glassfish fuerza a que exista en Internet una gran cantidad de tutoriales que permiten, por ejemplo, actualizar la versión de Glassfish que viene incluida por defecto hasta la última versión.

Será parte del proceso de desarrollo de cada uno de los componentes implementar las interfaces que deseen hacia los clientes, es decir, el desarrollo de componentes también implica el desarrollo de los recursos REST que se necesiten

4.3.1 Interfaz REST

En la siguiente tabla se van a exponer los recursos REST que se han creado y los métodos que soportan.

La interfaz que se expone representa todas las acciones que un usuario puede llevar a cabo sobre el sistema. Las acciones que se marcan como restringidas requieren que el usuario envíe su clave de sesión. Dicha clave de sesión se envía como una cabecera de tipo “Authorization”.

Al margen del nombre del recurso, habría que añadir la URL en la que todos los recursos se encuentran. El montaje que se ha hecho durante este proyecto, la URL queda de la siguiente manera:

https://IP/Soldados-war/resources

Soldados-war es el nombre del proyecto que contiene todos los recursos REST. Dicho proyecto se encuentra entre el código adjunto, en la carpeta Codigo/Sistemas Distribuidos/Netbeans/Soldados-war .

La mejor manera de ver cómo se hacen las llamadas a la API es a través de la aplicación de demostración. El código JavaScript+AJAX que hace las llamadas se encuentra en Codigo/Aplicacion Web/js .

Recurso	Método	Parámetros	Restringida	Explicación
/token	*POST*	*Usuario , Contraseña*	*NO*	*Devuelve el token de sesión*
/usuario/{nombre}	*PUT*	*Contraseña*	*NO*	*Crea un usuario con el nombre y contraseña especificados*
/jugador/{nombre}	*GET*	*-*	*SI*	*Devuelve la información básica del jugador (nombre, dinero, último cargo)*
/jugador/ {nombre}/objetos	*GET*	*-*	*SI*	*Devuelve los objetos que el usuario ya ha adquirido*
/jugador/ {nombre}/armeria	*GET*	*-*	*SI*	*Devuelve las armas y arrojables que el usuario ya ha adquirido*
/jugador/ {nombre}/plantilla	*GET*	*-*	*SI*	*Devuelve los soldados que el usuario tiene así como los equipamientos que llevan*
/jugador/ {nombre}/objetos/ {objeto}	*POST*	*EQUIPAR-{soldado}*	*SI*	*Equipa un objeto en un soldado*
/jugador/ {nombre}/plantilla/ {soldado}/intelecto	*POST*	*{nombreint}*	*SI*	*Fija el nombre del intelecto que el soldado lleva*
/jugador/ {nombre}/plantilla/ {soldado}/objetos/ {objeto}	*POST*	*DESEQUIPAR*	*SI*	*Elimina un objeto del equipamiento del soldado*
/jugador/ {nombre}/armeria/a rmas/{arma}	*POST*	*EQUIPAR-{soldado}*	*SI*	*Equipa en el soldado el arma*
/jugador/ {nombre}/armeria/a rmas/{arma}	*POST*	*MUNICION-{soldado}*	*SI*	*Equipa en un soldado un cargador del arma*
/jugador/ {nombre}/plantilla/	*POST*	*DESEQUIPAR*	*SI*	*Quita a un determinado soldado su arma y todos los*

{soldado}/armeria/ armas/{arma}				*cargadores equipados de la misma*
/jugador/ {nombre}/armeria/a rrojables/ {arrojable}	*POST*	*EQUIPAR-{soldado}*	*SI*	*Equipa un arrojable en el soldado*
/jugador/ {nombre}/plantilla/ {soldado}/armeria/ arrojables/ {arrojable}	*POST*	*DESEQUIPAR*	*SI*	*Quita un arrojable del equipamiento del soldado*
/tienda	*GET*	*-*	*NO*	*Muestra todos los objetos que se encuentran en venta (solo objetos, no hay armas ni soldados)*
/tienda/{objeto}	*POST*	*COMPRAR*	*SI*	*Compra un objeto para el usuario*
/armeria	*GET*	*-*	*NO*	*Muestra todas las armas y arrojables que se encuentran en la armería*
/armeria/armas/ {arma}	*POST*	*COMPRAR*	*SI*	*Compra un arma para el usuario*
/armeria/armas/ {arma}	*POST*	*MUNICION*	*SI*	*Compra un cargador para el usuario*
/ armeria/arrojables/ {arrojable}	*POST*	*COMPRAR*	*SI*	*Compra un arrojable para el usuario*
/mercenarios	*GET*	*-*	*NO*	*Muestra todos los soldados en venta*
/mercenarios/{tipo}	*POST*	*COMPRAR-{nombresolda do}*	*SI*	*Compra un nuevo soldado para el usuario y lo bautiza con el nombre que se fija en el cuerpo del mensaje*
/intelectos/ {jugador}/	*PUT*	*{Código fuente}*	*SI*	*Introduce un nuevo intelecto del usuario en el sistema*

{nombreintelecto}				
/intelectos/ {jugador}/ {nombreintelecto}	*GET*	-	*SI*	*Muestra el código fuente del intelecto*
/intelectos/ {jugador}	*GET*	-	*SI*	*Muestra los nombres de todos los intelectos que el usuario tiene en el sistema*
/reportes/{nombre}	*GET*	-	*NO*	*Muestra el contenido de un reporte guardado*
/partida/{nombre}	*GET*	-	*NO*	*Devuelve la información básica de una partida ya disputada*
/partida/fecha/ {fecha}	*GET*	-	*NO*	*Devuelve los nombres de todas las partidas disputadas un determinado día*
/partida/jug/ {jugador}	*GET*	-	*NO*	*Devuelve los nombres de todas las partidas disputadas por un jugador*
/partida/jug/ {jugador}/ultimas	*GET*	-	*NO*	*Devuelve los nombres de todas las últimas partidas disputadas por un jugador*
/ranking/top	*GET*	-	*NO*	*Devuelve el top 10 de jugadores*
/ranking/jug10/ {jugador}	*GET*	-	*NO*	*Devuelve los puntos que tiene un jugador y otros 9 jugadores más cercanos a él*
/ranking/jug/ {jugador}	*GET*	-	*NO*	*Devuelve los puntos y la posición en la que se encuentra un jugador*
/ranking/pos/ {posicion}	*GET*	-	*NO*	*Devuelve 10 posiciones consecutivas en el ranking empezando por la especificada en el recurso*
/sala	*DELETE*	-	*SI*	*Elimina al jugador de la sala*

				de espera. El jugador será eliminado de la sala de espera igualmente en el caso de que no haga llamadas PUT regularmente (cada menos de 60 segundos)
/sala	*PUT*	*-*	*SI*	*Conecta al jugador en la sala de espera. Necesita hacer esta acción antes de conectarse a ninguna partida*
/sala	*GET*	*-*	*NO*	*Muestra todos los nombres de las salas activas y su información básica (lugar de creación y número de jugadores)*
/sala/{nombre}	*GET*	*-*	*SI*	*Conecta a un jugador a la sala y le da información sobre las condiciones de la partida y los jugadores que ya están conectados*
/sala/{nombre}	*POST*	*UNIR*	*SI*	*Conecta a un determinado jugador a la sala de la partida con el nombre especificado.*
/sala/{nombre}	*POST*	*SALIR*	*SI*	*Desconecta al jugador de la sala de la partida.*
/sala/{nombre}	*POST*	*JUGAR*	*SI*	*Pide iniciar la simulación. Todos los jugadores tienen que estar listos.*
/sala/ {nombre}/soldados	*PUT*	*{lista de nombres de soldados}*	*SI*	*Selecciona los soldados a los que usará en una partida. Una partida no puede comenzar hasta que todos los jugadores hacen esta acción*
/sala/{nombre}	*PUT*	*-*	*SI*	*Crea una partida. El nombre del recurso tiene que ser igual al nombre del jugador*

Tabla 1: Documentación sobre la API REST

4.3.2 Arquitectura de la solución a bajo nivel

En el siguiente diagrama se amplía el nivel de detalle sobre las capas de persistencias y lógica de negocio.

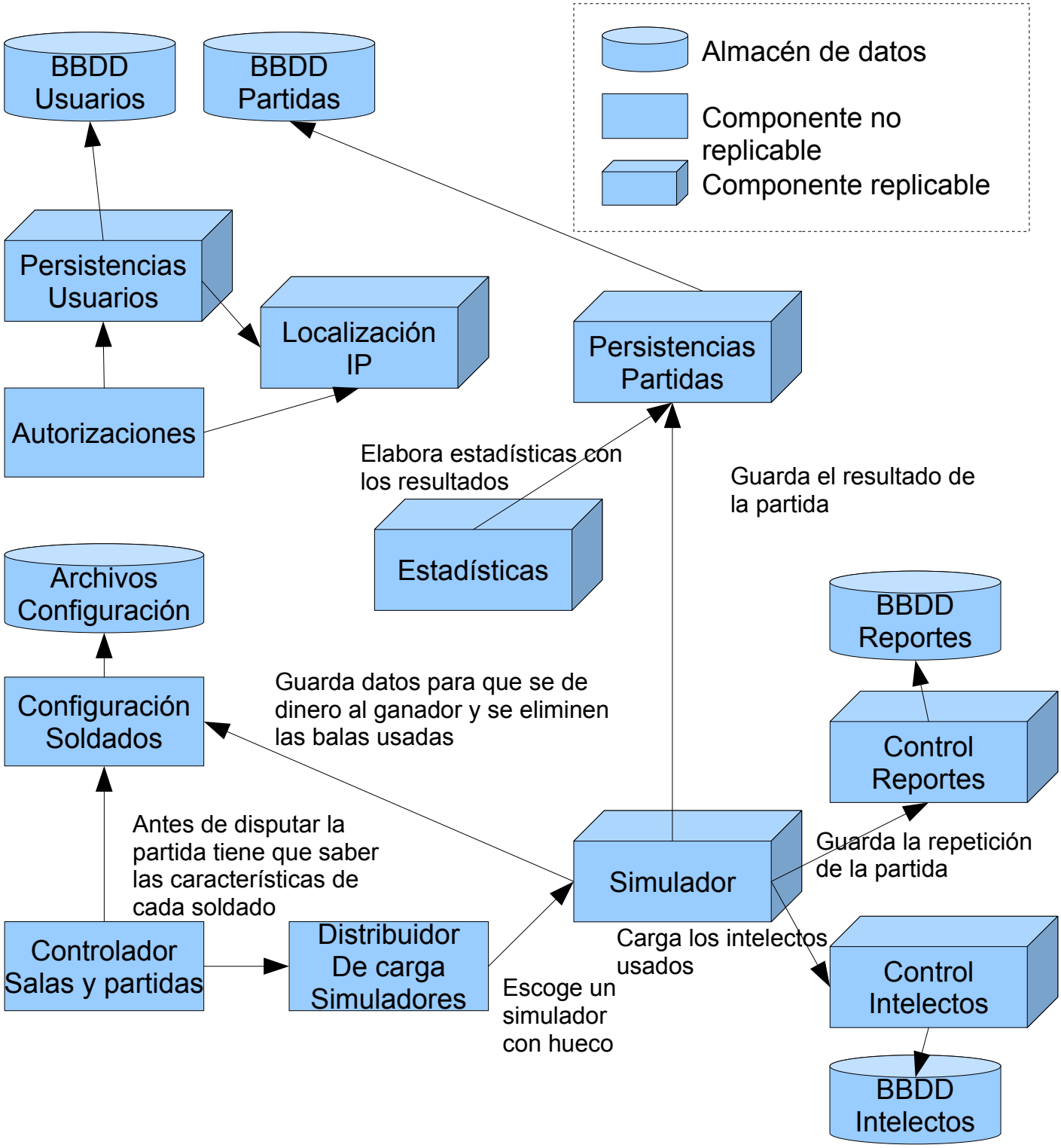

Ilustración 17: Arquitectura de los componentes a bajo nivel

4.3.3 Replicabilidad (EJBs vs RMI)

Se considera un requisito fundamental (ver la sección de objetivos, página 9) del sistema que sus componentes puedan ser replicables. Para aumentar el rendimiento y la tasa de supervivencia, se debe trabajar activamente en hacer un buen diseño dónde los componentes puedan ser replicados.

En este sentido se siguieron dos caminos bastante distintos para implementar la lógica de negocio.

Implementación mediante EJB (Enterprise Java Beans)

La tecnología Enterprise Java Beans (EJB) se encuentra en el conjunto de librerías de Java Enterprise Edition (JEE – JavaEE). El objetivo de esta tecnología es permitir al programador de los sistemas distribuidos abstraerse de muchos de los problemas básicos relacionados con el desarrollo de los mismos (concurrencia, seguridad, transacciones o persistencia entre otros) [19].

La programación de EJBs con Netbeans es muy sencilla. Tiene múltiples asistentes, y al igual que con los recursos REST, permite al programador olvidarse de casi todo lo que no aporta funcionalidad útil para el proyecto. Además está demostrado que las soluciones implementadas mediante EJBs escalan de una manera bastante positiva [20] [21], pero lo hacen de una manera independiente del programador. Es el contenedor de EJBs (del que se ocupa Glassfish) el que es replicado de una manera totalmente simétrica entre los distintos nodos del sistema.

Lo que se busca y necesita para este proyecto es una cierta capacidad de determinar que partes serán replicadas, cuales no y en que condiciones. El diseño enfatiza esas partes, y no se puede tener tanto control sobre los contenedores de EJB.

Es muy importante notar que la programación mediante EJB tiene muchas ventajas: cuando una replica falla se usan las demás, cuando las instancias se saturan se crean más automáticamente, la labor de clusterización es casi automática (totalmente independiente del programador). Existen dos motivos, sin embargo, para no implementar el sistema de esta manera:

- Se limita la potencia del diseño: por más bueno que sea el diseño de un cierto componente, dependerá de cómo el contenedor se comporte. Podría perderse mucho rendimiento si se replica cada parte de su estado interno.

- El administrador del sistema puede tener una sensación de descontrol total. Si algo falla, la depuración de dicho fallo se vuelve mucho más difícil al no tener control sobre él.

Implementación mediante objetos distribuidos que se comunican por RMI

La mejor manera para desarrollar el sistema basándonos en los diseños anteriores es mediante objetos distribuidos. La idea es poder colocar los distintos componentes en los nodos que se crea convenientes. El control por parte del administrador es total. La principal desventaja es que requiere trabajar activamente en la implementación de los principios de clusterización de servicios.

Para implementar esta solución se ha usado el middleware de Java RMI (Remote Method Invocation), que permite declarar una clase como remota, de manera que cuando llamamos a métodos en objetos de dicha clase, los métodos se ejecutarán en el ordenador en que la instancia de la clase se encuentre registrada. La gran ventaja de RMI respecto de otras soluciones viene por la facilidad de aprendizaje. En muy poco tiempo se puede aprender a usar esta tecnología de manera muy eficiente. Por contra su gran desventaja es que es incompatible con otros lenguajes de programación que no sean Java. Es decir, si se quisiera implementar algún componente usando un lenguaje distinto a Java habría que cambiar la implementación de los objetos [22].

La transformación entre EJB y RMI es relativamente rápida, y se puede aprovechar una gran cantidad de código. Dónde hay que hacer hincapié al desarrollar el sistema es en la localización de los distintos objetos. El problema es el siguiente: ¿cómo sabe el servidor a cuál de las instancias tiene que llamar cuando quiera hacer una llamada remota? El mecanismo más sencillo para responder a esta pregunta consiste en programar una especie registro de objetos.

El registro de objetos funciona de la siguiente manera: cuando un objeto remoto nuevo se conecta, se lo comunica al registro de objetos, diciéndole además para qué sirve dicho objeto. Cuando otro objeto necesita de este, se lo pide al registro, el cual responde con la dirección IP y puerto dónde puede contactar al objeto remoto.

La estructura de datos que se implementa es una especie de tabla hash. Las claves son los nombres de los servicios, y los valores son listas de nodos. Un ejemplo de estado de la tabla hash podría ser el siguiente:

Clave	Valor
ControlReportes	{Maq1:9090 , Maq2:9090}
ControlIntelectos	{Maq1:9091}
Autorización	{Maq3:9091}

Tabla 2: Ejemplo de uso del Registro de Instancias

Java RMI implementa su propio registro de objetos remotos, pero se ha querido hacer un registro propio de manera que no se fuerce a los objetos registrados a implementar RMI. Por ejemplo, si en 2 meses se decidiera que hay un módulo que funciona mucho mejor usando Corba, se podría registrar ese nodo en el registro de objetos igualmente. Sería misión de los nodos que lo usan saber que para comunicarse con él no pueden usar RMI sino el protocolo con el que el objeto haya sido configurado.

En la práctica, el registro de objetos se ha implementado como un recurso REST más, el cual acepta un método GET sobre el nombre de un recurso para tener toda la lista de recursos, un método DELETE sobre un recurso específico para borrar una instancia del registro y un PUT para registrar un recurso.

La dirección en la que el registro de objetos se encuentra es fija, y en la versión inicial del sistema, el registro de objetos no es replicable. El registro de objetos no recibe una gran cantidad de tráfico, pero si se cae todo el sistema queda inutilizable, de modo que replicarlo para que nunca se dé esa situación sería una prioridad para futuras versiones.

Para facilitar el uso del registro de objetos cuando se requiere buscar objetos allí registrados se ha programado una librería sencilla. La librería provee la siguiente funcionalidad:

- Cargar las referencias de todos los objetos de un determinado tipo que se han registrado en el registro de objetos.
- Informar cuando un determinado objeto está fallando.
- Para detectar si el fallo es puntual o se trata de un fallo por un estado corrupto por parte del objeto, todas las clases distribuidas implementan un método “estado”. Cuando dicho método es invocado, los objetos son forzados a ejecutar una prueba de estado (en el caso de los gestores de bases de datos se comprueba que tienen conexión con la base de datos, por ejemplo). Si ejecutan la comprobación y es defectuosa, no

responden a la invocación. El invocador sabe entonces que dicho objeto falla y puede ser borrado del registro de objetos para que nadie más use dicha instancia.

La librería solo facilita las cosas del lado del cliente (es decir, cuando se consultan otros objetos remotos). Del lado del servidor se requiere una librería que trabaje con REST (HTTP en su defecto) y que permita hacer una petición PUT en el servidor.

Hay un punto extra que no puede verse en el diagrama de arquitectura pero sin embargo está presente cuando se usan objetos distribuidos con RMI: el patrón fachada se repite sistemáticamente en cada uno de los componentes propuestos. La comunicación que se da entre los distintos objetos se hace en función de las interfaces, y la implementación de cada uno de los componentes podría variar sin tener que modificar los clientes. Es más, se podrían descomponer los componentes en sub-componentes, y mientras la interfaz original sea implementada no habrá ningún problema de compatibilidad. Esta capacidad del diseño aporta toda la adaptabilidad que el sistema necesita.

Para que cada uno de los componentes sean compatibles con el control de replicabilidad, en vez de implementar la interfaz java.rmi.Remote deberán implementar la interfaz "RMITestable" que ha sido desarrollada para forzar a los componentes a incluir el citado método "estado".

Se puede ver un ejemplo de la adaptabilidad del diseño en la siguiente serie de imágenes, donde se explica unas posibles mejoras que se podrían aplicar sobre el componente Autorizaciones sin modificar el resto de componentes que lo usan.

El siguiente diagrama representa la implementación actual del componente Autorizaciones.

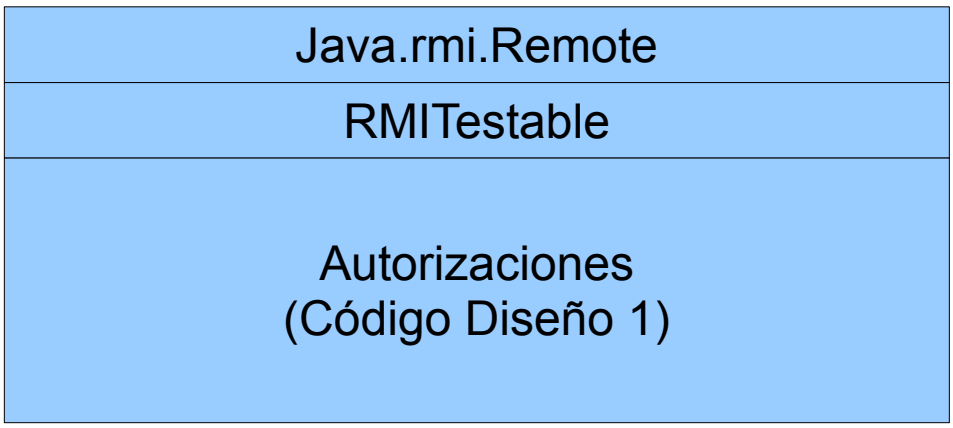

Ilustración 18: Implementación del componente Autorizaciones

El diagrama a continuación representa una posible mejora sobre el diseño actual que usaría una base de datos distribuida para guardar los distintos tokens.

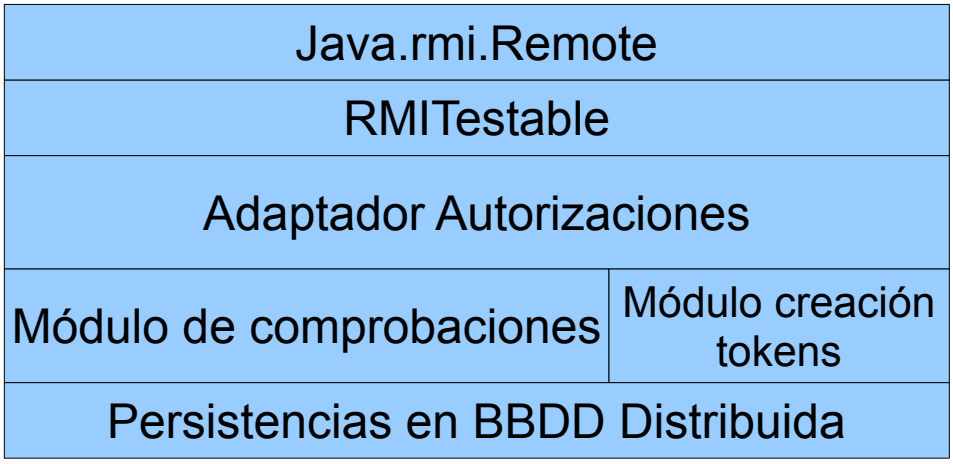

Ilustración 19: Mejora sobre la implementación del componente Autorizaciones

En un último paso, se crearía un siguiente nivel, en el que cada nodo tiene su propia caché, y en caso de no encontrar los datos en ella, se comunica con otros objetos remotos que puedan tener dichos datos.

Ilustración 20: Mejora sobre la mejora anterior

4.3.4 Progresión del diseño

La división de la funcionalidad en componentes ha sido constante en todo el proceso de diseño de la aplicación. En cambio, ha habido muchos cambios a lo largo del proceso en la forma de desplegar los distintos componentes. Las primeras versiones de la implementación confiaban en el enfoque de los EJB.

En las primeras versiones existía un único contenedor WEB y un único contenedor EJB. La replicabilidad se implementaba clonando dichos contenedores, y lo hacía directamente Glassfish.

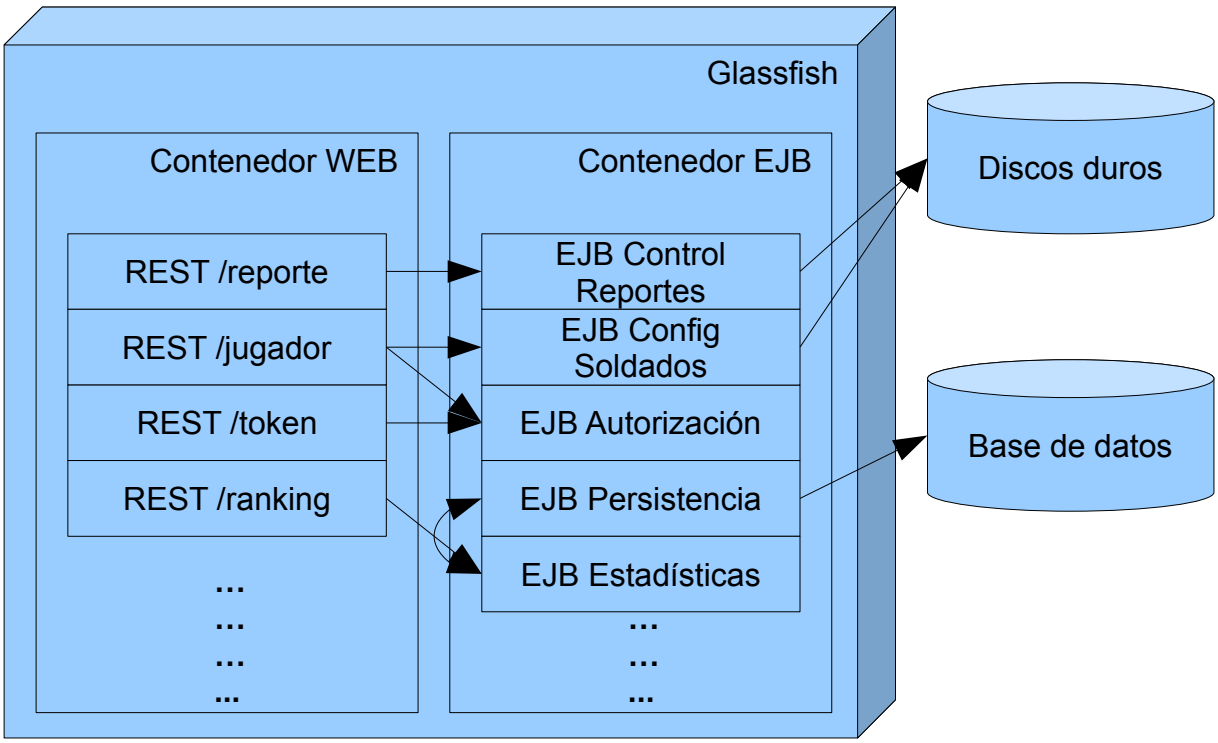

Ilustración 21: Evolución 1 de la arquitectura de componentes

A pesar de todo, los componentes no eran totalmente compatibles con dicho enfoque, ya que muchos de ellos usaban el disco duro para guardar sus datos de manera persistente (y el disco duro es un recurso local de cada nodo).

Los enfoques posteriores confiaban en la separación de los componentes en distintos EJBs, de manera que dichos componentes pudieran ser replicados o no en función de sus necesidades. El problema de dichos enfoques es la dificultad de administración de los distintos contenedores. La configuración de los distintos contenedores especificando cuales son replicables y cuales no es muy difícil a priori. Se complica además la clusterización de nodos en Glassfish. Dicho enfoque no compensa el esfuerzo.

A pesar de todo se elaboraron 3 diseños distintos antes de ser rechazado el enfoque:

El primero separa el cluster de contenedores web de los distintos contenedores EJB (que pueden ser clusterizados o no). Además intenta usar peticiones asíncronas para almacenar cosas en disco duro, de manera que pudieran replicarse los datos mediante algún mecanismo.

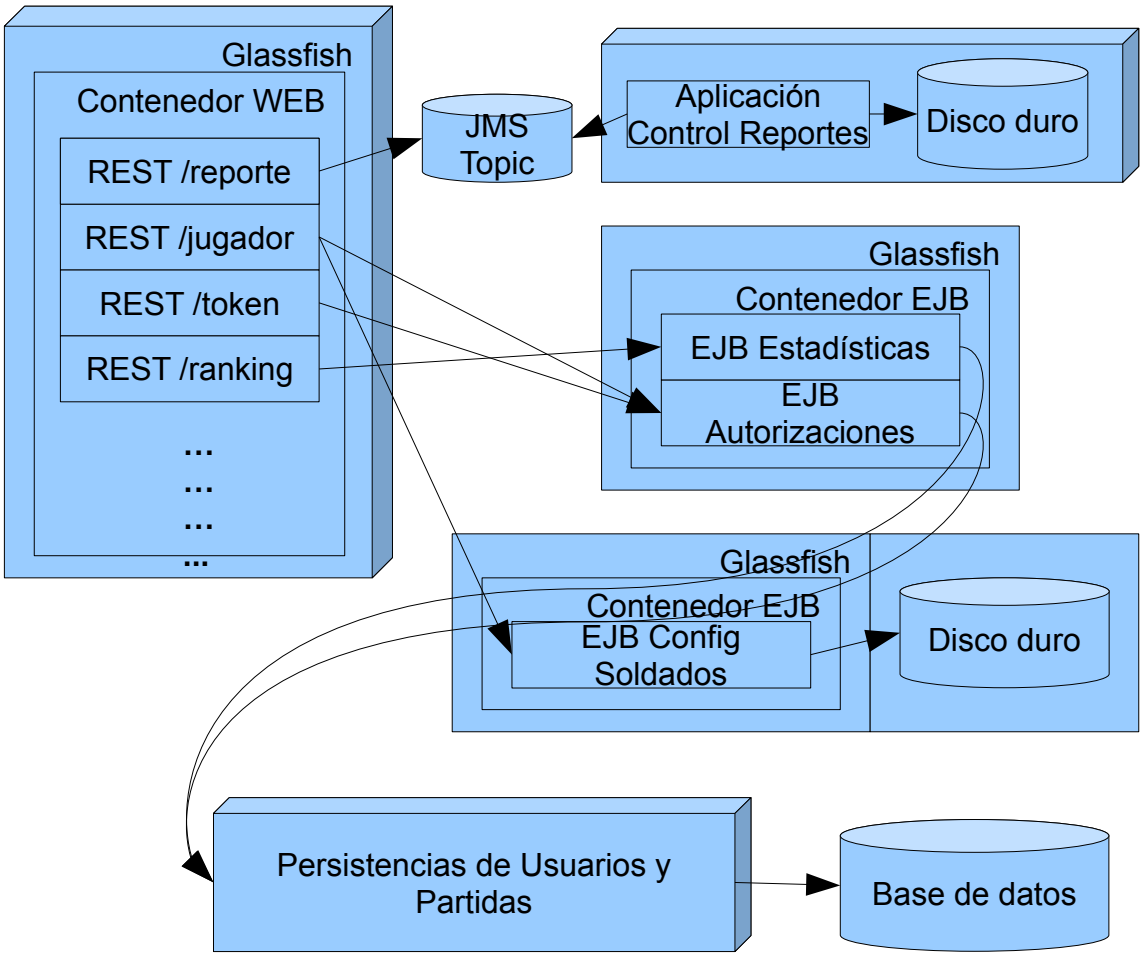

Ilustración 22: Evolución 2 de la arquitectura de componentes

NOTA: en ninguno de los diagramas se incluyen todos los componentes del sistema. Se incluyen unos pocos para permitir a los diagramas ser suficientemente ilustrativos.

El segundo de los enfoques opta por una división distinta: ahora los contenedores WEB y EJB que tienen la misma funcionalidad se alojan en los mismos nodos.

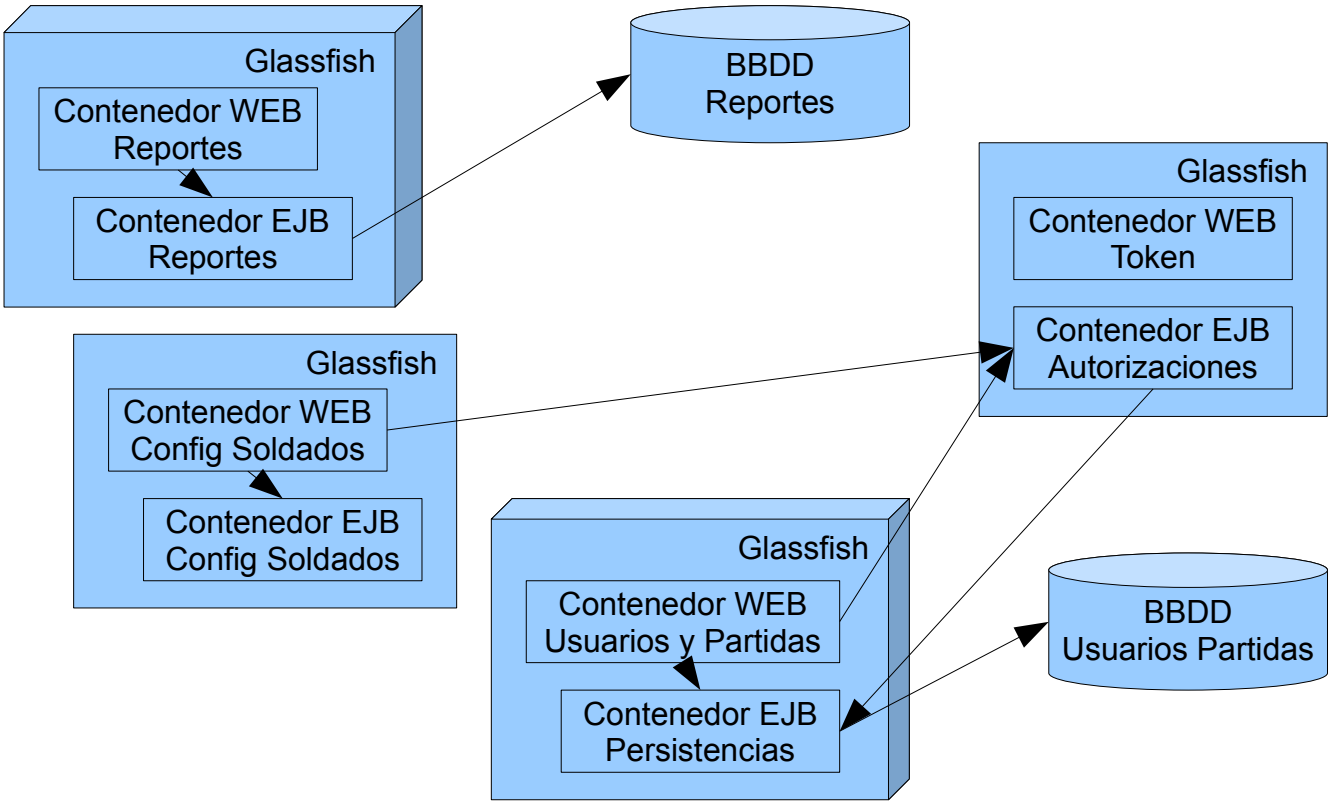

Ilustración 23: Evolución 3 de la arquitectura de componentes

El último de los enfoques no finales mezcla el enfoque de los EJB con aplicaciones no contenidas en los EJB. Para permitir hacerlo implementa un almacén de recursos (muy similar al registro de objetos que finalmente ha sido implementado), el cual centraliza las ubicaciones de todos los componentes externos a los EJBs.

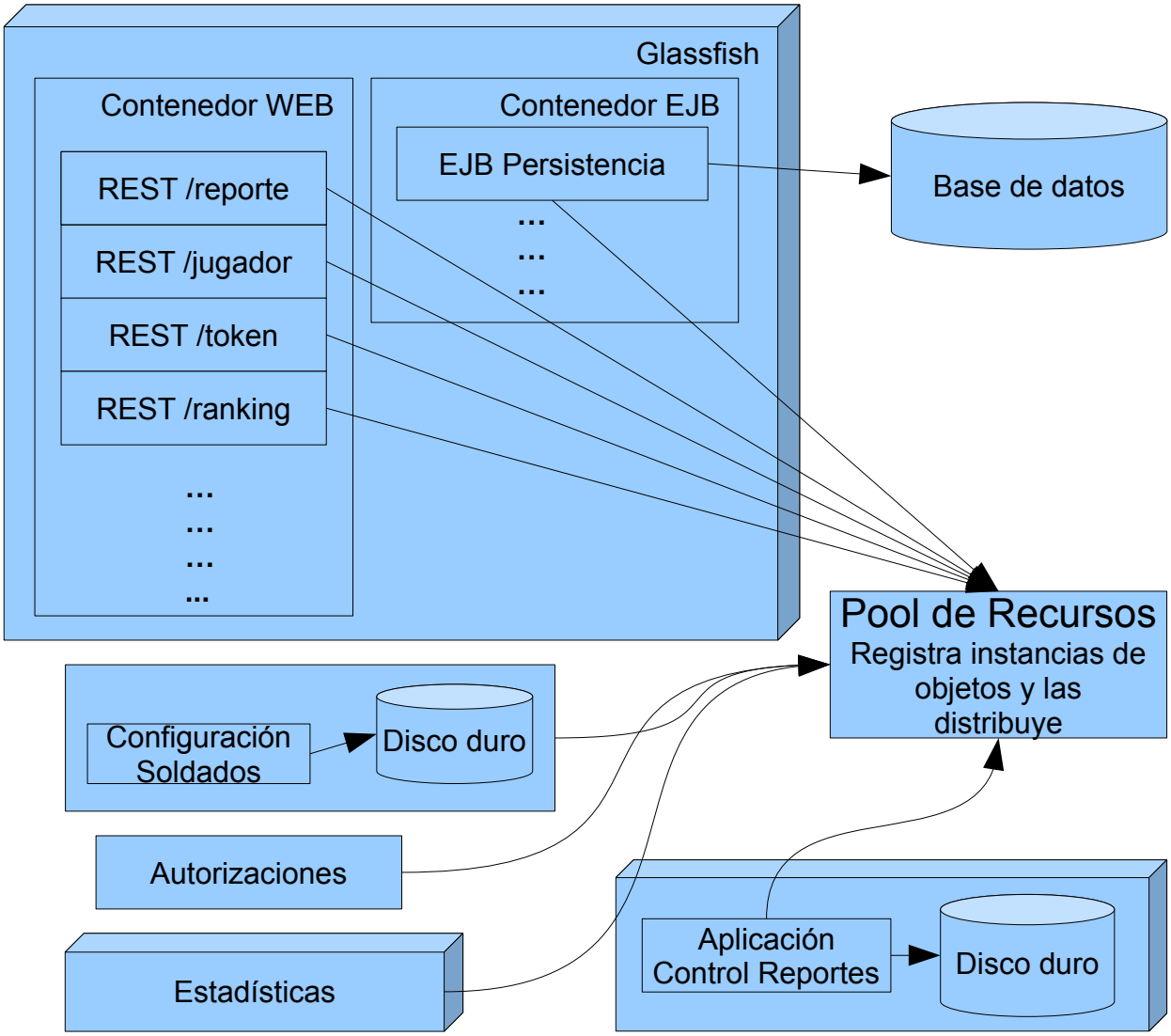

Ilustración 24: Evolución 4 de la arquitectura de componentes

La implementación final quedó explicada al inicio de la sección (página 57), y está totalmente basada en objetos distribuidos que implementan los distintos componentes. Cada objeto se comunica con los demás mediante RMI. Se utilizan contenedores WEB para alojar los recursos REST y el registro de instancias. Dichos contenedores si son clusterizables.

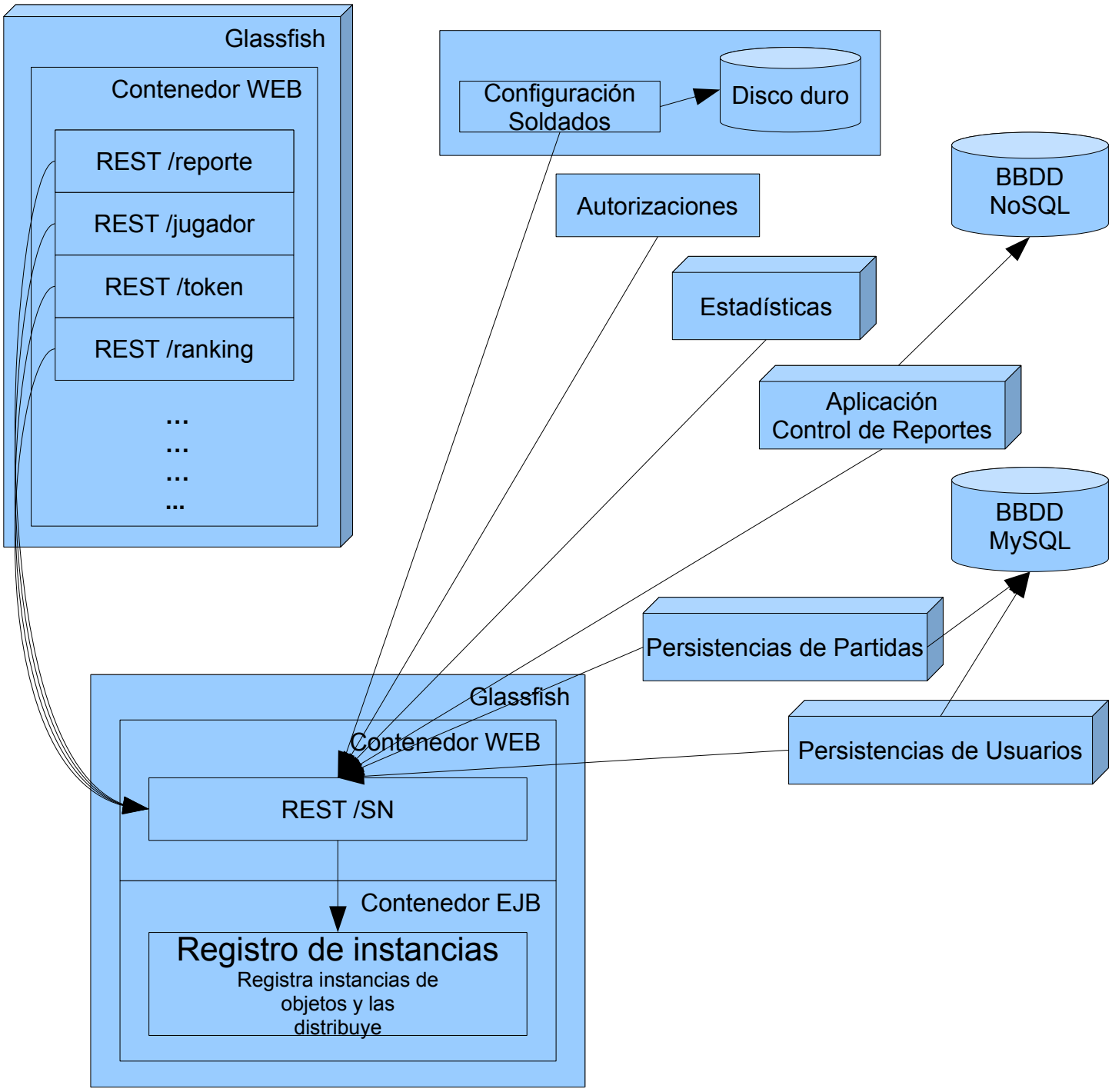

Ilustración 25: Arquitectura final de los componentes

4.3.5 Explicación del desarrollo de cada componente.

Autorizaciones

Se implementa una tabla hash que relaciona claves de sesión con los usuarios. Cuando un usuario inicia sesión se crea una clave nueva en la tabla que apunte al usuario. Existe una llamada a este módulo que pregunta si el token de sesión existe, y en tal caso devuelve el nombre del usuario que validó el token.

El formato del token, además de un número aleatorio grande incluye el país, la ciudad y las coordenadas geográficas que han sido detectadas por el módulo de localización IP al iniciar la sesión.

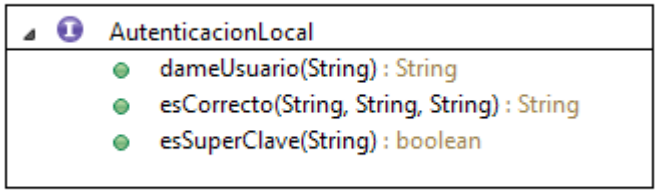

Ilustración 26: UML de la interfaz de Autorizaciones

Control de Intelectos

Se comunica con la base de datos distribuida de Cassandra. Se indexan los intelectos gracias al nombre de usuario y el nombre del intelecto.

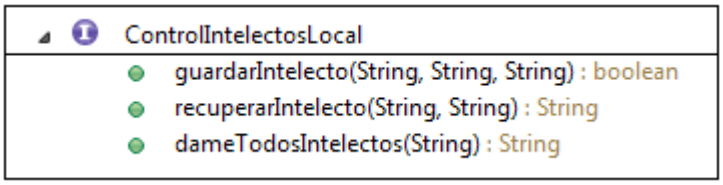

Ilustración 27: UML de la interfaz de Control de Intelectos

Control de Reportes

Se comunica con la base de datos distribuida de Cassandra. Se indexan los reportes mediante el día en que la partida se disputó y el nombre del creador de la misma.

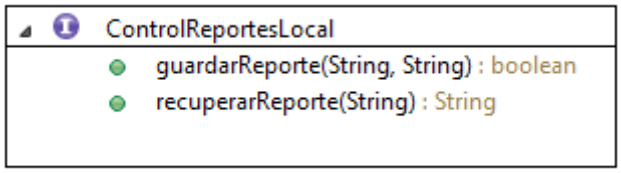

Ilustración 28: UML de la interfaz de Control de Reportes

Localización

Implementa la librería GeoLite City de la compañía Maxmind [23]. La librería está diseñada para llegar en un 85% de los casos a determinar la ciudad en la que la IP buscada se encuentra.

Según la información oficial de la aplicación, la precisión de la misma cuando las IP son españolas alcanza el 72% con una precisión de 25 millas (el 75% en una versión de pago). Se puede ver una tabla con las precisiones en [24].

Se hablará en la sección de futuro trabajo sobre cómo se podría mejorar la funcionalidad de localización de los usuarios.

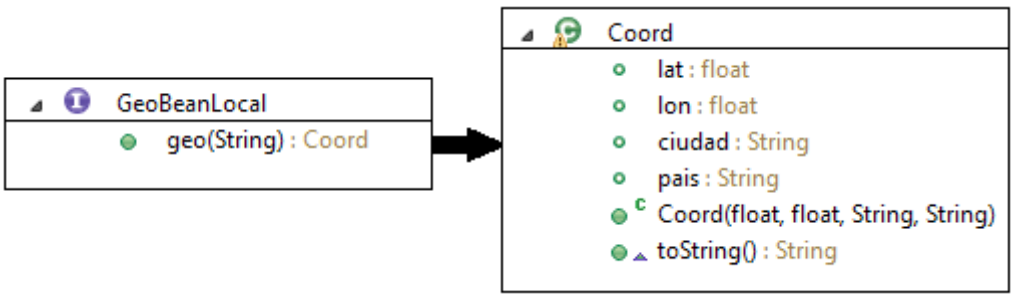

Ilustración 29: UML de la interfaz de Localización IP

Estadísticas

Trata de calcular un ranking de usuarios en función de quién es el mejor. Para hacerlo asigna a cada usuario un número de puntos en función de sus resultados. Se parte de dos principios básicos para evaluar a los usuarios: en primer lugar, cuanto más arriba está en la tabla el jugador, mejor se supone que es. El segundo principio es que si un jugador gana a otro es porque dicho jugador es mejor que el otro. Usando los principios anteriores, se enuncian las siguientes reglas de puntuación de una partida:

El jugador, sin importar su posición en la tabla, tiene que ganar puntos por el hecho de

ganar una partida.

El jugador, sin importar su posición en la tabla, debe recibir puntos por el hecho de participar en una partida. Hay que motivar a todos los jugadores a que jueguen partidas. Lógicamente la cantidad de puntos que reciba no será demasiada.

Los jugadores de la partida recibirán puntos en función del nivel de la partida. Si en una partida juegan los 2 mejores jugadores, deberían obtener más puntos por lo importante de dicha partida.

Si un jugador gana a otro que está mejor situado que él en el ranking, debería obtener como recompensa puntos extra.

Si un jugador grande gana a uno pequeño, deberían limitarse los puntos que gana.

Para poder ver la diferencia entre el ranking anterior y el actual, los rankings son actualizados diariamente con las partidas del día anterior. El ranking del día N dependerá de las partidas del día N-1 y el ranking del día N-1. Si un jugador gana 3 partidas en un mismo día, no será hasta el día siguiente que pueda ver su posición en el ranking mejorada.

Con todas las reglas anteriores se consigue una función matemáticas que asigna a un jugador por una determinada partida un número de puntos. Dicha función es la suma de las 3 funciones siguientes:

$$pBaseGanador(jug) = E + K * (matados(jug) + 1) / (muertos(jug) + 1)$$
$$pBasePerdedor(jug) = K * (matados(jug) + 1) / (muertos(jug) + 1)$$

$$pNivel() = Q * \frac{\sum_{0}^{N-1} ant(i)}{\frac{N}{ant(mejor)}}$$

$$pBonus(jug) = \frac{R * mediaPerdedores}{ant(jug)} \quad si \quad ant(jug) * 1.2 > mediaPerdedores$$

Ilustración 30: Fórmulas matemáticas para el cálculo de puntos en las estadísticas

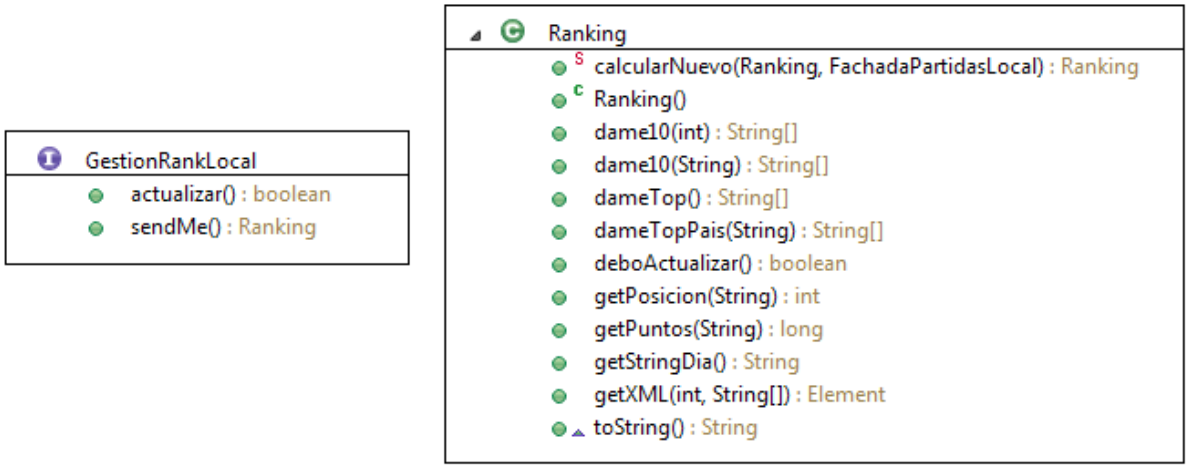

Ilustración 31: UML de la interfaz de Estadísticas

Control de Partidas

Este componente implementa una tabla de salas. Las salas son bautizadas con el nombre del creador. Existen dos métodos fundamentales en el control de partidas en general. El usuario tiene que iniciar sesión en el servidor de partidas para poder jugar. Esto sirve para que al entrar a jugar se puedan ver qué usuarios están dispuestos a jugar en ese momento. Lógicamente, al salir de la sala de espera (cuando están conectados pero sin entrar en ninguna partida), deben notificárselo al servidor. Para que en caso de fallo los jugadores no se queden ilimitadamente en el servidor, cada minuto los jugadores que no han dado señales de vida son desconectados. Pasa igualmente cuando los jugadores están dentro de una sala esperando para jugar.

En la implementación actual este componente no está replicado. Una idea posible sería que en un nodo específico se gestionara la sala de espera mientras que haya otro tipo de nodos que gestionen un cierto número de salas cada uno. De esa manera se aumentaría la escalabilidad.

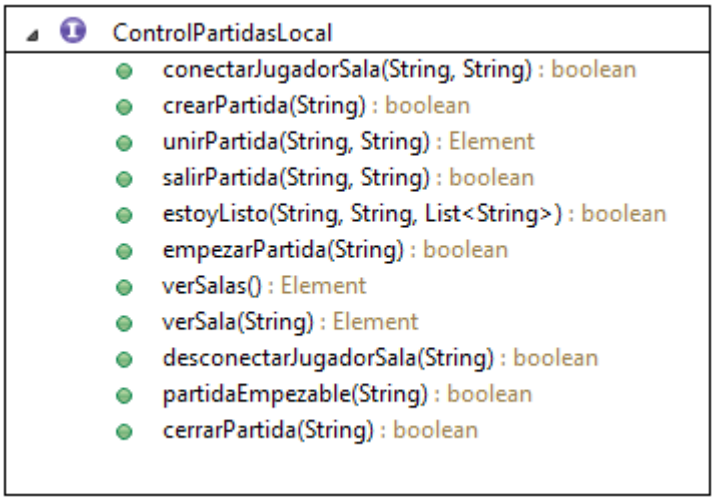

Ilustración 32: UML de la interfaz del Control de Partidas

Simulador

La implementación del simulador está detallada en el proyecto con el nombre en clave de "juego". Se implementa un método en los nodos simuladores que permite saber cuántas partidas se están simulando a la vez y cuántas pueden ser simuladas, lo que facilita el balanceo de carga.

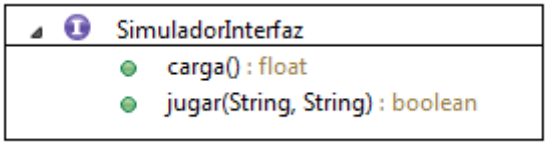

Ilustración 33: UML de la interfaz del Simulador

Distribuidor de Partidas

Basándose en la carga de cada simulador, permite distribuir las partidas entre los distintos simuladores en función de la carga que tenga cada uno.

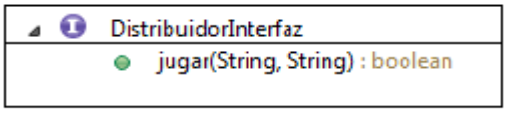

Ilustración 34: UML de la interfaz de Distribuidor de Partidas

Configuración de Soldados

Este módulo trabaja casi íntegramente con el formato XML. Almacena las propiedades de soldados y armas en XML de manera que enviarle los soldados al simulador sea una tarea sencilla.

Ilustración 35: UML de la interfaz de Configuración de Soldados

Persistencia de Usuarios y Persistencia de Partidas

Estos módulos usan JPA (Java Persistence API), que es un conjunto de funciones que facilitan las operaciones más básicas sobre una base de datos. La arquitectura de la base de datos puede verse en el siguiente diagrama:

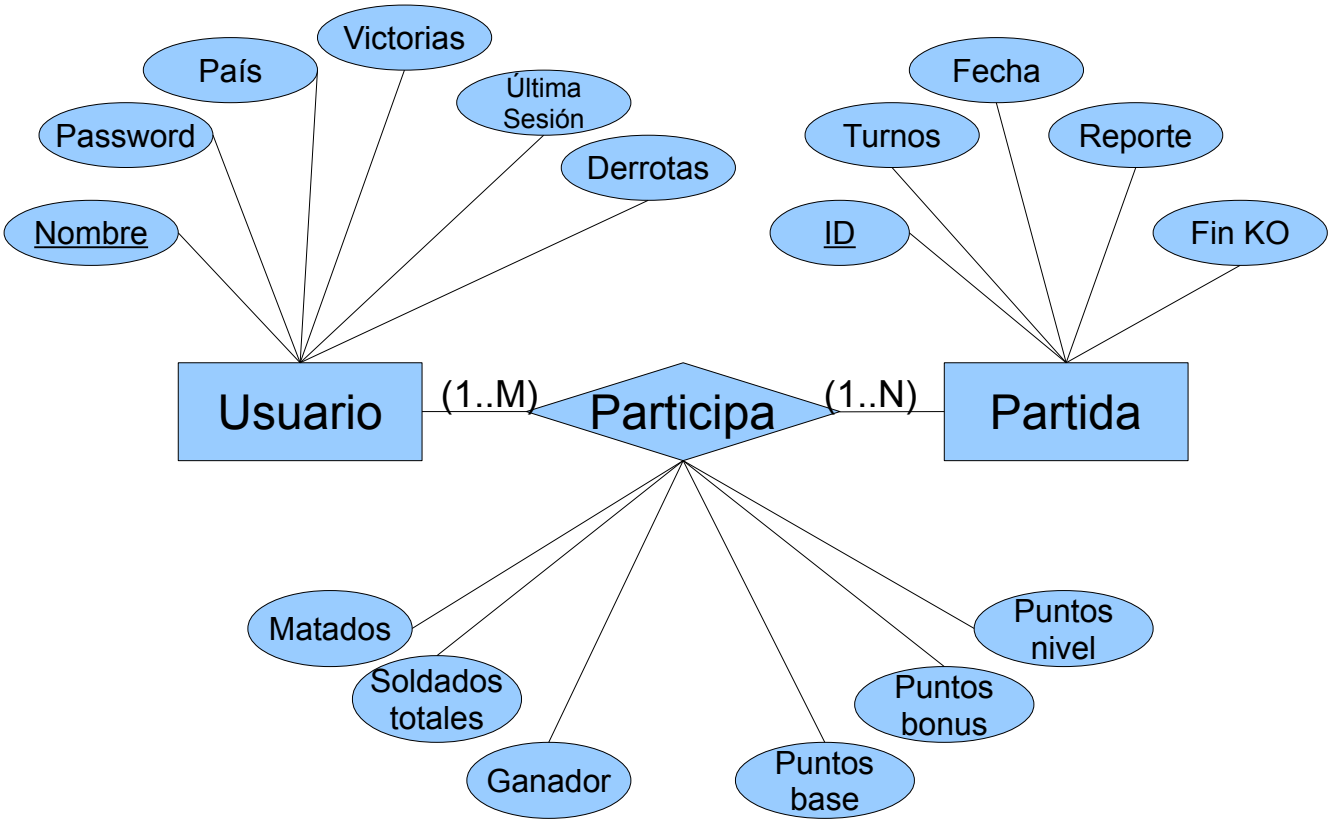

Ilustración 36: Diagrama Entidad-Relación de la base de datos

JPA se apoya en JDBC (Java Database Connection), el cual actúa de interfaz entre Java y cualquier tipo de base de datos relacional. De esta manera la comunicación con la base de datos desde la aplicación nunca necesitaría ser cambiada.

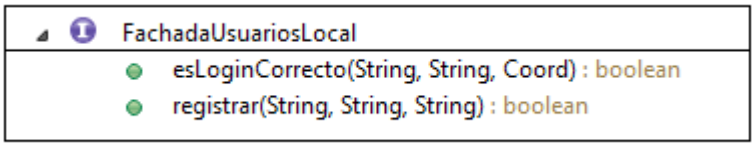

Ilustración 37: UML de la interfaz de Persistencias de Usuarios

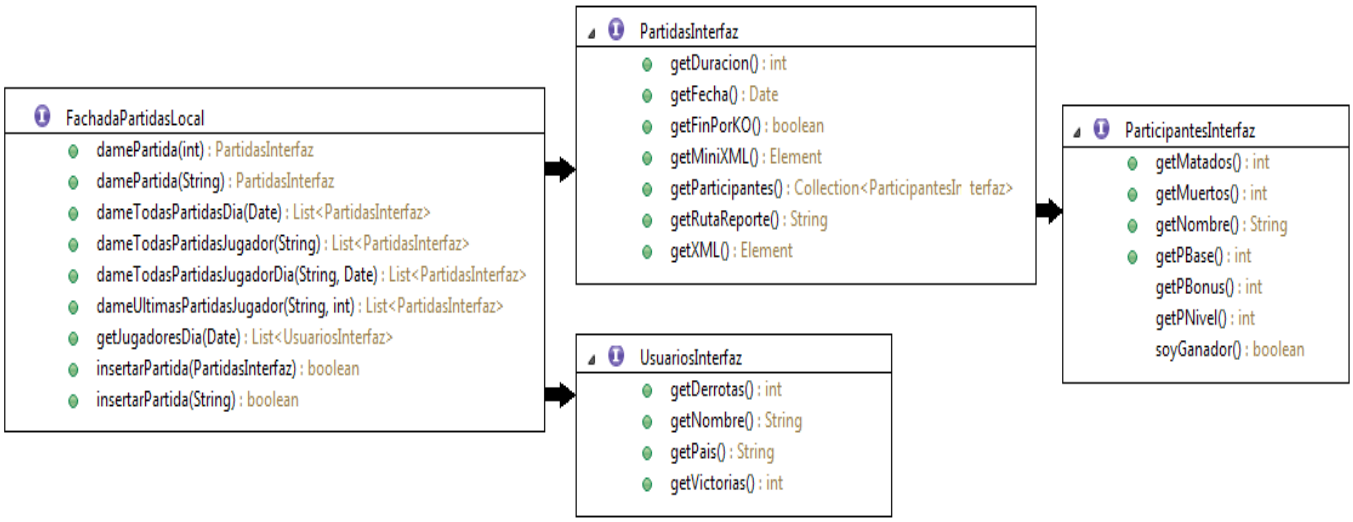

Ilustración 38: UML de la interfaz de Persistencias de Partidas

4.3.6 Seguridad

La seguridad del sistema distribuido es muy importante. El hecho de que el sistema esté disponible en línea lo convierte en un posible y probable foco de ataques de todo tipo. El sistema se protege de las siguientes amenazas:

- Interrupción de servicio: la efectividad de ataque de denegación de servicio podría ser relativamente alta en el caso de realizar consultas válidas (por ejemplo, si se pide al sistema mostrar 100000 veces por segundo la tienda). La solución que queda implementada en este momento está basada en el uso de cachés. Así, las llamadas que consultan el estado del sistema no siempre requieren un alto procesamiento, mientras no se produzcan cambios se podrá dar el mismo resultado siempre.
- Interceptación: para proteger a los usuarios de que sus mensajes sean leídos se confía en la encriptación SSL que se da a los mensajes gracias a Glassfish.
- Modificación: dentro del SSL se usan codificaciones redundantes de manera que si el mensaje ha sido modificado se detectará al ser recibido.
- Fabricación (cuando un usuario se intenta hacer pasar por otro): cada usuario usa una clave de sesión que siempre viaja encriptada mediante SSL. Mientras otro usuario no la tenga no debería ser capaz de hacerse pasar por él. Las claves de sesión son regeneradas cada vez que el usuario inicia sesión.

Además de las amenazas anteriores, se han de extremar las precauciones en la administración de los sistemas. Hay que tener en cuenta que todas las contraseñas por defecto habrán de ser cambiadas. Además las tecnologías deberán estar siempre actualizadas con los últimos parches de seguridad (el sistema operativo también).

La última precaución se da a nivel hardware. Para tratar de imposibilitar los ataques a los distintos componentes del sistema distribuido, se colocan todos ellos dentro de una red privada NAT con un cortafuegos. La idea es que dentro de la subred solo pueda llegar tráfico de una manera: a través del servidor web. Se habilitan únicamente los puertos típicos para HTTP (80) y HTTPS (443). Los puertos de administración de Glassfish (4848) y de conexión con las bases de datos no son accesibles desde fuera de la subred. Esto da un mayor nivel de seguridad a la aplicación, aunque reduce las capacidades de administración desde el exterior.

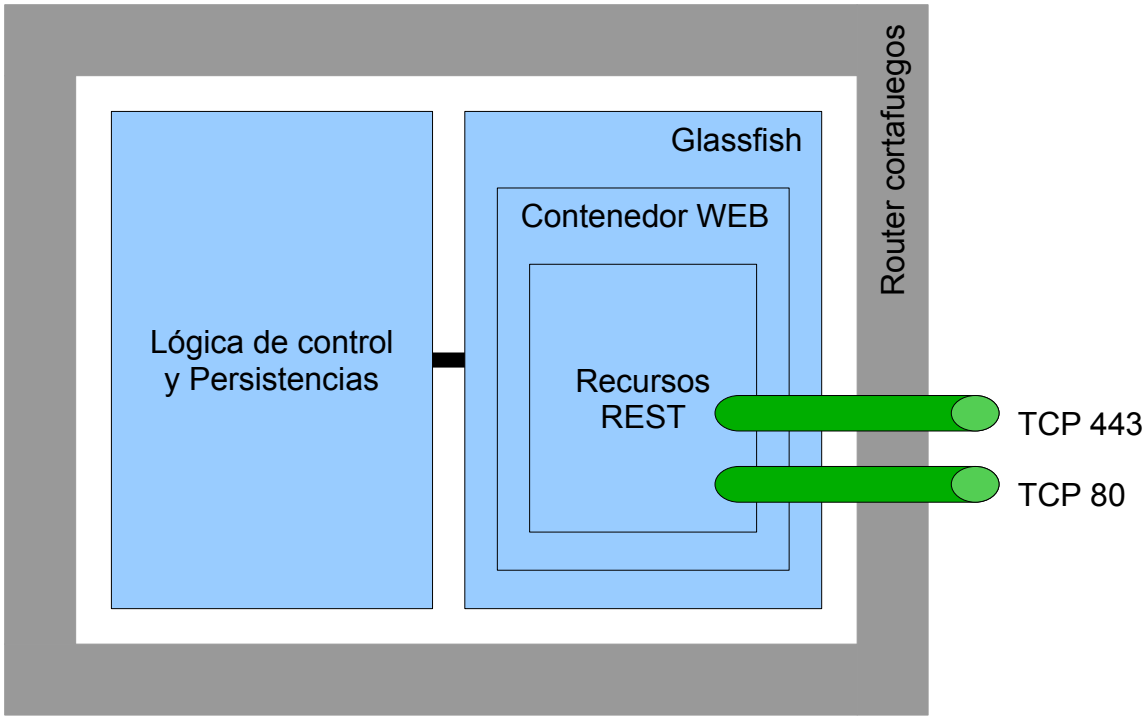

Ilustración 39: Seguridad por cortafuegos

En el caso de que se quisiera montar una arquitectura de red en la que los nodos no se encuentran en redes protegidas por cortafuegos habría que tomar medidas que garanticen la identidad de los componentes de manera que el usuario no programe componentes fraudulentos y desvirtúe el comportamiento del sistema.

Situación posible en ese caso:

Un usuario podría crear un componente de estadísticas y registrarlo en el registro de objetos. Una vez hecho esto, cuando el sistema necesite un ranking, será el componente fraudulento el que responderá.

4.3.7 Mejoras aplicadas

La explicación que se ha dado de los componentes está basada en la última versión de los mismos, pero antes de esta versión se desarrollaron otras versiones provisionales. El caso más importante es el de los componentes Control de Reportes y Control de Intelectos. Ambos tienen el mismo requisito: tienen que ser capaces de servir ficheros de texto y de almacenarlos.

En un primer momento se optó por una solución rápida: crear un componente que tiene una carpeta local en el disco duro y que almacena los distintos archivos. El problema de este enfoque es que si se tiene el mismo componente replicado (por ejemplo dos instancias de Control de Intelectos), cada uno de los componentes tiene un estado distinto.

Para hacer coherente la solución anterior se podría haber seguido un enfoque basado en un nodo organizador (el cual controla en que nodo se encuentra cada archivo) y otros nodos almacenadores (que almacenan los archivos sin más). El problema de este enfoque es que para garantizar que no fallaba si algún nodo dejaba de responder de manera repentina se debería haber trabajado en algún enfoque que garantizara la coherencia entre nodos (replicar la información en tiempos muertos, redistribuir la información cuando algún nodo desapareciera o cuando un nuevo nodo apareciera, etc). Se trata de un problema muy interesante, pero muy largo para ser abordado y existe una gran cantidad de trabajo y estudio por parte de especialistas del ámbito, pero existen soluciones libres, completas y usadas en grandes sistemas que resuelven el problema.

4.3.8 Bases de datos NoSQL

En la aplicación actual se adoptó el enfoque de las bases de datos NoSQL (no relacionales). Una base de datos NoSQL es un sistema de gestión de información que no requiere de tablas con formato fijo para guardar los datos. Entre las ventajas de usar una base de datos NoSQL encontramos que generalmente escalan muy bien horizontalmente. Existen varios tipos de bases de datos NoSQL como pueden ser los pares clave-valor, implementaciones de BigTable o implementaciones basadas en grafos. Las necesidades del proyecto son perfectamente

cubiertas con los pares clave-valor.

Las tecnologías NoSQL están de actualidad entre los sistemas distribuidos por esa capacidad de escalar horizontalmente que se mencionaba y su alto rendimiento. Esto lo consiguen a costa de limitar la funcionalidad de las bases de datos (artículo de opinión sobre NoSQL en [25]).

En este proyecto se ha usado Apache Cassandra para dar servicio a los sistemas de Control de Reportes y Control de Intelectos. Esta implementación de NoSQL incluye una estructura tabular que permite definir propiedades sobre objetos. Las propiedades principales de Cassandra son:

- Es un sistema descentralizado: todos los nodos tienen el mismo rol. Los datos se distribuyen a lo largo del cluster de manera que no hay un único foco de fallo. El sistema por tanto tolera fallos en nodos.
- Altamente configurable: se puede desplegar la base de datos de muchas maneras, pudiendo crearse estructuras complejas como clusters de clusters.
- Alta elasticidad: la adición o supresión de nodos modifica la distribución de los datos anteriores, mejorando la distribución de los mismos entre los recursos activos.

Apache Cassandra es usado actualmente en sistemas como Digg y SoundCloud.

Se implementan dos keyspaces en esta base de datos: Intelectos y Reportes. Cada uno tiene a su vez una familia de columnas (concepto similar a las tablas) homónimo.

La configuración de dichos keyspaces permite que en escritura y lectura se escriba y lea de la mitad más uno de todos los nodos disponibles. Con esto se asegura una total coherencia entre los resultados de las operaciones en los nodos. En el caso de que la mitad de los nodos o más estén caídos, se intenta escribir o leer de un único nodo. Se intenta por tanto cumplir la funcionalidad mínima de la base de datos.

4.3.9 Base de datos MySQL

Por otro lado, otra de las mejoras que se hizo en la aplicación fue la migración del sistema de base de datos de usuarios y partidas hasta MySQL.

MySQL es el sistema de bases de datos más popular, en gran parte por su fácil integración en páginas web dinámicas junto con PHP. Es un software gratuito y de código abierto propiedad de Oracle que se distribuye mediante la licencia GNU v2. Además de a pequeña

escala, esta tecnología se usa también por grandes compañías como Google, Twitter, Facebook, Wikipedia o Youtube (Más información a través de [26]).

Inicialmente se usaba Apache Derby, sistema muy ligero, sencillo y configurado por defecto en Netbeans. El principal motivo para cambiar la tecnología de la base de datos es la escalabilidad. La eficiencia de MySQL está más que probada y es bastante escalable a través de clusters como se explicó en [7] mientras que en Derby la escalabilidad no es una de las prioridades y a pesar de que existen algunas alternativas [27], no parecen tan sencillas y eficientes como la proporcionada por MySQL.

La migración de la base de datos fue un proceso muy sencillo gracias a la interfaz JDBC de Java. Únicamente hubo que instalar el driver específico de MySQL para JDBC y configurar la dirección de conexión con la base de datos, que en el caso que nos atañe es jdbc:mysql://102.168.0.13:3306/usuariospartidas .

4.3.10 Sistema de replicación automática

Para garantizar que el sistema se recupera a la perdida de instancias de componentes, se propone una aplicación extra que se encargaría de monitorizar el estado de los distintos componentes para decidir cuando se necesita añadir nuevas instancias de componentes.

El mecanismo es muy sencillo y se basa en el diseño del registro de instancias, lugar centralizado en el que se encuentran todos los componentes registrados. Si en algún momento hay un componente que no tiene ninguna instancia registrada, eso quiere decir que el sistema necesita un nuevo componente que aporte dicha funcionalidad. La idea es hacer una aplicación que explora el registro de instancias y cuando detecta la falta de algún componente lanza un nuevo proceso que cubre dicha falta.

Una posible modificación sobre este enfoque sería que, basándose en la implementación del método “estado” que todos los componentes tienen, se juzgara si se necesitan más instancias o no. Por ejemplo, si se hiciera que cada componente en su método estado respondiera diciendo cuantas peticiones por minuto ha tenido durante los últimos 5 minutos se podrían detectar componentes que están saturados, y añadir nuevas instancias de los mismos.

4.4 Explicación del montaje final de la solución

Para el montaje provisional del sistema se cuenta con los siguientes 4 equipos:

1) Router Scientific-Atlanta EPR2320R2. Con cortafuegos y soporte para redes NAT.

2) Ordenador de sobremesa. Intel Core2Duo E5500 a 2.5 Ghz x2. 2GB memoria DDR2. Windows XP.

3) Ordenador portátil. Intel Core2Duo T1600 a 1.66Ghz x2. 4GB memoria DDR2. Ubuntu 10.04.

4) Ordenador de sobremesa. Intel Core i5 a 3.3Ghz x4. 8 GB memoria DD3. Windows 7.

El equipo 4 no se usará de manera regular como nodo del sistema, y quedará reservado únicamente a aliviar la carga de los otros dos equipos cuando se detecte mucho tráfico en el sistema.

El equipo 2 será la base del sistema, conteniendo el servidor web de Glassfish y el registro de instancias de objetos. Contendrá también la base de datos MySQL. Será un nodo más en el cluster de la base de datos de Cassandra. Contendrá todos los componentes que no son replicables (Configuración de soldados, Autorizaciones, Control de Partidas y Distribuidor de Partidas).

El equipo 3 contendrá todos los componentes que no contiene el equipo 2, y además formará parte del cluster de Cassandra.

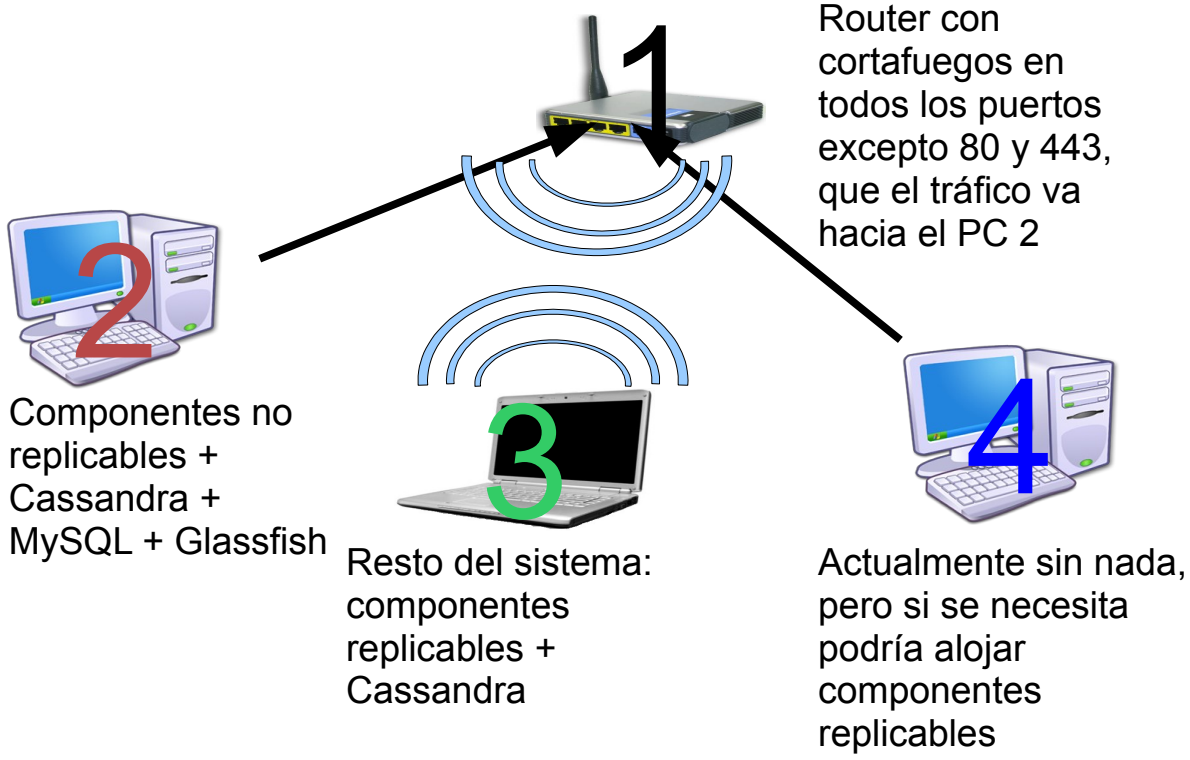

Ilustración 40: Montaje final de la solución

Es importante señalar que para añadir un nodo al sistema solamente se requiere de inicio una máquina virtual de Java. Si se quiere ejecutar el registro de instancias o el servidor web se requiere Glassfish, y si se quiere participar del cluster de MySQL o Cassandra se necesita instalar dichas tecnologías (la primera vez que se unen al cluster los datos son traspasados automáticamente).

En un tiempo medio inferior a 20 minutos se puede tener un nuevo nodo en el sistema funcionando plenamente, y uno de los puntos más interesantes es que los equipos no necesitan ser punteros, puesto que la máquina virtual de Java puede ser ejecutada en equipos de toda clase.

4.5 Aplicación de demostración

Partiendo de la interfaz REST desarrollada y usando JavaScript y AJAX se desarrolló una aplicación de prueba que permite usar gran parte de la funcionalidad del sistema distribuido. Puesto que el sistema distribuido usa en una gran cantidad de sus métodos XML para el intercambio de datos, gran parte del peso de la aplicación de demostración ha tenido que ver con la programación de hojas de traducción XSLT (tecnología que transforma archivos XML en lo que interese al programador, que en el caso de esta aplicación ha sido HTML).

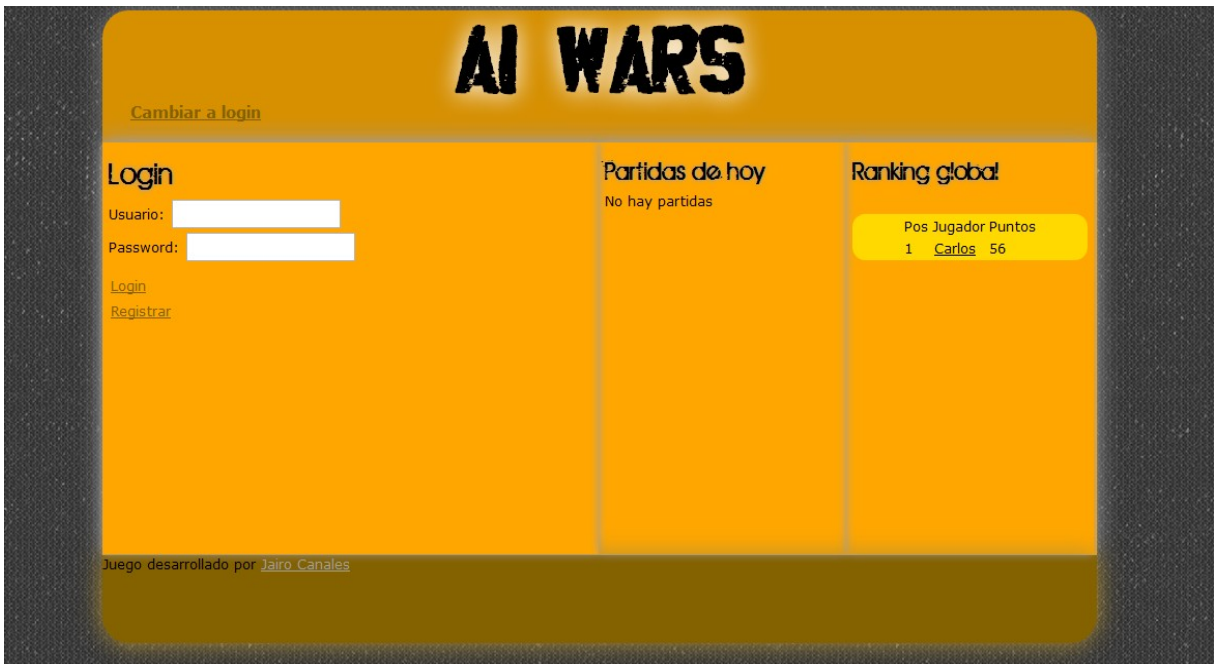

Ilustración 41: Imagen de la aplicación de demostración

Sin darle demasiada importancia a la estética (aunque usa CSS para colocar los contenidos en la página), el usuario puede registrarse, autenticarse, comprar objetos y armas, equipárselos

a los soldados, entrar a las salas de juego, seleccionar los soldados con los que participar en las batallas y disputarlas. También puede acceder al ranking (top10 y el ranking de sus competidores inmediatos). Existen dos funcionalidades que no se han llevado a cabo con JavaScript y se exponen a continuación.

Visualizador de partidas

Se necesitaba mostrar de una manera visual los reportes de las partidas una vez disputadas. A pesar de que con JavaScript se podría analizar el archivo XML del reporte y haber pintado gráficamente soldados en la pantalla, el motor gráfico de la aplicación ya se había desarrollado junto al motor de juego. El coste de aprender a hacer el pintado mediante JavaScript y la implementación del visualizador era mucho mayor que el uso del código Java tal cual fue desarrollado.

Se transformó el motor gráfico a un Java Applet (aplicación software desarrollada en Java que se puede incrustar en una página web). El mayor problema de esto es que se requiere que el navegador en el que la aplicación se ejecuta tenga instalado un plug-in.

Compilador de intelectos

Antes de que el usuario suba sus intelectos se hace necesario que compruebe que estos son correctos. Para hacerlo necesita compilar código Java, lo cual es una labor bastante difícil de ejecutar en JavaScript. También se hace muy difícil de hacer incluso para un Applet (se ha llegado a conseguir ejecutar el compilador en un Applet local, pero nunca ha funcionado al colocarlo en un servidor remoto).

La solución final es distribuir una aplicación Java que incluye un compilador mínimo de intelectos y permite obtenerlos del sistema y añadirlos al mismo.

4.6 Pruebas de rendimiento

Todo software requiere pruebas que evalúen si la ejecución está siendo correcta. Estas pruebas se pueden ejecutar en un único equipo y de manera independiente. Debido a los requisitos y la naturaleza de este proyecto, se necesita otro tipo de pruebas que comprueben que el sistema se comporta como debe en unas condiciones dadas. Las condiciones que deben darse son: múltiples clientes desde sitios remotos intentan acceder a las diversas funciones del sistema simultáneamente. La manera ideal de probar el sistema con esas condiciones es habilitar una beta privada, pero requiere un gran equipo de usuarios, del cual no se ha podido

disponer.

En esta sección se va a comprobar que el diseño y la implementación del sistema han sido correctos en cuanto a que dan soporte a una escalabilidad buena. Naturalmente la complejidad de esta tarea es mucha, y el único mecanismo a disposición del alumno es usar pruebas sintéticas. Dichas pruebas están basadas en clientes del sistema creados específicamente para las mismas. Su comportamiento trata de simular en gran parte el comportamiento de un usuario real del sistema.

Para ejecutar las pruebas se simularán a la vez N usuarios que acceden al sistema, ejecutando las mismas secuencias de acciones simulando a la vez los tiempos de espera que el usuario requeriría para tomar decisiones.

Se establecen 4 categorías en función del valor de N (o hilos en paralelo usados para ejecutar las pruebas). La categoría de bajo rendimiento tiene N=4, la segunda N=16, la tercera N=64 y la última N=256. Se tienen en cuenta las siguientes estadísticas: tiempo medio de respuesta del sistema según petición, tiempo de espera más largo, número de llamadas fallidas.

La simulación de cada usuario dura una media de 15 minutos, y se basa en un ciclo de tareas aleatorio. Cada tarea requiere un tiempo en blanco antes de comenzarse (que simula el tiempo de decisión del usuario). Las acciones que se llevan a cabo son:

1. El usuario se registra (0 segundos)
2. El usuario se autentica (0 segundos)
3. Se descarga su información básica (0 segundos)
4. Se descarga sus objetos, armas y plantilla (0 segundos)
5. Se descarga la tienda, armería y mercenarios (0 segundos)
6. Ve su ranking (0 segundos)
7. Compra 2 soldados (30 segundos – 1:30 minutos por soldado)
8. Compra 2 objetos (30 segundos – 1:30 minutos por objeto)
9. Compra 2 armas (30 segundos – 1:30 minutos por arma)
10. Equipa todos los objetos y armas en los soldados (10 – 20 segundos por equitación)
11. Entra a la sala de partidas (5 – 10 segundos)
12. Crea una partida (55 – 1:50 segundos)
13. Sale de la partida (30 – 60 segundos)
14. Accede a las partidas disputadas en el día (2 – 3 minutos)
15. Accede a la información de una partida en especial (1 – 2 minutos)
16. Accede al reporte de una partida (30 – 60 segundos)
17. Accede al ranking (10 – 20 segundos)
18. Accede a las partidas de un usuario (10 – 20 segundos)
19. Accede al reporte de una partida (30 – 60 segundos)

Tras la ejecución de las pruebas se han procesado los datos obtenidos, y se presentan en la siguiente tabla:

N	Media por llamada	Sumatorio de máximos	Total de fallos
4	53ns	1708ns	0
16	33ns	2036ns	0
64	26ns	2049ns	0
256	28ns	7875ns	0

Tabla 3: Resultados de las pruebas de rendimiento

Observando los resultados, se puede concluir que incluso con 256 usuarios en paralelo el sistema es capaz de responder positivamente y en tiempos muy tolerables.

Se pueden criticar especialmente tres puntos respecto a estas pruebas:

1. No han llegado a mostrar cuántos usuarios son soportados por el montaje actual del sistema. Sería interesante tener pruebas que permitan afirmaciones como “con el montaje actual se soportan 600 usuarios” o “para soportar 3000 usuarios simultáneos se requieren 4 máquinas servidoras”.
2. Se está juzgando únicamente el rendimiento interno del sistema. Las pruebas se ejecutan desde un ordenador de la red local del sistema, por tanto los tiempos totales son bastante pequeños. Habría sido idóneo para estas pruebas haber dispuesto de usuarios (reales o virtuales) situados en puntos repartidos por todo el planeta.
3. El comportamiento de los usuarios ha sido intuido. En ningún caso se ha estudiado el comportamiento de los usuarios reales (los cuales no han existido).

4.7 Verificación de la planificación

En esta sección se exponen las fechas de comienzo y finalización de las distintas tareas que se han llevado a cabo. Muchos de los datos han sido sacados del servidor SVN en el que se han ido guardando los archivos. Otros sobre los que faltaban registros han sido estimados.

Tarea	*Inicio*	*Fin*
Juego	01/01/12	18/02/12
Elaboración del prototipo	01/01/12	08/01/12
Interfaces	08/01/12	10/01/12
Implementación clases 1	10/01/12	14/01/12
Implementación clases 2	14/01/12	18/01/12
Implementación clases 3	18/01/12	28/01/12
Composición + tests 1	28/01/12	04/02/12
Composición + tests 2	04/02/12	11/02/12
Documentación para jugadores	11/02/12	16/02/12
Documentación de la implementación	16/02/12	18/02/12

Tabla 4: Tareas desarrolladas para el juego

Tarea	*Inicio*	*Fin*
Sistemas Distribuidos	18/02/12	20/04/12
Descomposición funcional	18/02/12	19/02/12
Ordenación de componentes	18/02/12	19/02/12
Diseño de interfaces entre sistemas	19/02/12	21/02/12
Diseño de Arquitectura 1	21/02/12	24/02/12
Componentes 1 en Arquitectura 1	24/02/12	10/03/12
Componentes 1 en Arquitectura 2	10/03/12	14/03/12
Componentes 2 en Arquitectura 2	14/03/12	18/03/12
Registro de instancias (Arquitectura 3)	18/03/12	22/03/12
Componentes 1 en Arquitectura 3	22/03/12	23/03/12
Componentes 2 en Arquitectura 3	23/03/12	24/03/12
Componentes 3 en Arquitectura 3	24/03/12	28/03/12
Componentes 4 en Arquitectura 3	28/03/12	01/04/12
Componentes 5 en Arquitectura 3	01/04/12	06/04/12
Pruebas globales del sistema	06/04/12	08/04/12
Adaptación a MySQL (investigación + impl)	08/04/12	10/04/12
Adatapción a NOSQL (investigación + impl)	10/04/12	13/04/12
Librería manejo replicas en Reg Instancias	13/04/12	18/04/12
Diseño e impl de pruebas sintéticas	18/04/12	19/04/12
Ejecución y análisis de pruebas sintéticas	19/04/12	20/04/12

Tabla 5: Tareas desarrolladas para el sistema distribuido

Tarea	*Inicio*	*Fin*
Demostración en navegador	24/03/12	07/04/12
Diseño básico	14/03/12	16/03/12
Desarrollo llamadas básicas genéricas	16/03/12	25/03/12
Adaptación a las llamadas a comps 1, 2 y 3	25/03/12	31/03/12
Adatapción a comps 4 y 5	31/03/12	04/04/12
Maquetación final	04/04/12	07/04/12

Tabla 6: Tareas desarrolladas para la demo

5 Conclusiones

5.1 Logros alcanzados

En este trabajo se han conseguido los siguientes logros:

- Se ha desarrollado un proyecto partiendo desde cero. El diseño de la funcionalidad del juego y del sistema distribuido ha sido importante. Al contrario que en otros proyectos dónde el tutor propone el enunciado del trabajo y el alumno tiene que analizarlo, en este proyecto es el alumno el que ha desarrollado su propio enunciado basándose en una creación propia.
- Se ha desarrollado un juego plenamente funcional, original y novedoso. A pesar de que no es atractivo para un alto porcentaje de los habituales jugadores de videojuegos, podría tener su cabida en un mercado especializado.
- Se ha desarrollado un sistema distribuido muy escalable capaz de mejorar la experiencia del usuario en cuanto a que facilita su uso del juego junto con otras personas.
- Se han integrado múltiples tecnologías cuyo rendimiento y calidad han sido probados en otros proyectos. Se ha tratado de reinventar la rueda lo menos posible para ahorrar tiempo y esfuerzo.
- El alumno ha trabajado en ámbitos y tecnologías muy solicitados en el mercado actual: JavaEE (usando EJBs y JMS), programación web con JavaScript + AJAX, web basada en XML + XSLT y bases de datos NoSQL.

5.1.1 Diseño modular

Uno de los puntos clave que demuestran que el diseño del sistema distribuido fue bien realizado puede verse a través de la experiencia sobre cómo se integró la tecnología NoSQL en el mismo.

Inicialmente, la funcionalidad cubierta con NoSQL se cubría usando el disco duro de los nodos. A pesar de que aún no se había dado respuesta a cuestiones como la coherencia entre los distintos nodos, la solución funcionaba perfectamente cuando los componentes “Control de Reportes” y “Control de Intelectos” no estaban replicados. Era, en cualquier caso, una solución bastante pobre al problema, y rápido se pensó una nueva mejora.

El diseño posterior de los componentes se basa en la integración de los sistemas con una base de datos distribuida NoSQL. El punto principal que se quiere destacar como conclusión es que el cambio total en el comportamiento del componente se dió de manera transparente al resto del sistema. Ni la interfaz del sistema ni los componentes que tenían llamadas hacia "Control de Reportes" y "Control de Intelectos" requirieron de ningún cambio.

Siguiendo la misma filosofía, hay varios componentes que requieren cambios (algunos de ellos requieren una adaptación para ser replicables). Dichos cambios pueden ser implementados sin requerir modificaciones en el resto.

5.2 Mejoras necesarias

A pesar de que el proyecto ha sido terminado y toda la funcionalidad básica se ha implementado, existen muchas tareas sencillas que podrían ejecutarse para mejorarlo sustancialmente (sección 5.2.1). También se requieren infraestructuras (las cuales necesitan de dinero), y se van a explicar en la sección 5.2.2.

5.2.1 Software

El sistema ha quedado en un porcentaje de desarrollo muy alto, tan alto que es plenamente funcional. No obstante, tiene algunas lagunas aún:

- Existen componentes que aún no son replicables. Deberían rediseñarse los módulos que no son replicables para que empiecen a serlo. Este punto no se limita únicamente a los componentes desarrollados mediante RMI: el registro de instancias debería también ser capaz de ser replicable (si falla en este momento el sistema entero deja de funcionar) y la interfaz web de la aplicación debería ser configurada mediante Glassfish para ejecutarse en modo cluster (el tiempo estimado para esta configuración es de menos de una hora).
- No se ha implementado ninguna automatización de control del sistema. El diseño hecho del sistema de replicación automática (página 77) debería implementarse para liberar de trabajo al administrador del sistema.
- La aplicación de demostración no abarca toda la funcionalidad del sistema distribuido. Por ejemplo, el usuario de la aplicación no puede ver el ranking en línea ordenado por países. Además, su diseño es poco atractivo.

5.2.2 Hardware

El montaje final de la solución es bastante casero y provisional. Existe una gran ventaja: el coste de inversión del desplegado del proyecto ha sido exactamente 0 euros puesto que se han aprovechado equipos antiguos. En cualquier caso, para mejorar la solución propuesta se proponen las siguientes modificaciones:

- Inversión en más equipos: la gran ventaja del sistema es su escalabilidad horizontal. Con 2 o 3 equipos en la red no se puede aprovechar totalmente. Los requisitos de dichos equipos no son demasiado altos ya que en principio sería necesario con que fueran capaces de ejecutar una máquina virtual de Java.
- Inversión en instalaciones: puesto que el sistema está montado en casa del alumno, existen dificultades para colocar todos los equipos, el cableado está montado de manera provisional y los problemas energéticos también existen (la factura de la luz aumenta tanto como la temperatura de la casa). Si el sistema generara algún tipo de beneficio esta situación tendría que cambiar.
- Inversión en conexión a la red: dado que el sistema es casero, es la conexión a Internet de la casa la que se usa. El alumno tiene una conexión a Internet asimétrica con 30Mb de bajada y 1Mb de subida. Esta conexión podría no ser suficiente para dar un correcto servicio a los usuarios. Además, el contrato con la compañía ONO debería ser estudiado para comprobar si es posible, como un cliente particular, alojar un servicio público.

5.3 Trabajos futuros

Además de las mejoras que se exponen a lo largo del apartado 5.2, existen varias lineas de trabajo que podrían llevar al progreso del proyecto en general.

5.3.1 Elaboración de un plan de negocio

El sistema desarrollado hasta este punto es una prueba de concepto que se ha hecho con coste 0 (el alumno ha sido el único desarrollador y los equipos ya eran propiedad del mismo). Para continuar desarrollando este proceso sería fundamental elaborar un plan de negocio el cual especificara fuentes de financiación.

El sistema de juego tiene cabida para la obtención de dinero a cambio de la venta de

objetos. Por ejemplo, se podría establecer una tasa de cambio entre dinero del juego y dinero real, de manera que un usuario pudiera comprar objetos o armas con el dinero obtenido a través de la experiencia en combate, o con dinero obtenido con su tarjeta de crédito real. Existe una gran cantidad de juegos en línea gratuitos que usan esta fuente de financiación (Ikariam, Team Fortress 2).

En cualquier caso, no es labor de este proyecto determinar los planes de negocio posteriores al mismo.

5.3.2 Motor gráfico mejorado

Los usuarios de videojuegos generalmente tienen debilidad por los motores gráficos atractivos. En la versión actual la visualización de combates es bastante pobre. Círculos y rectángulos se mezclan sobre una superficie blanca.

Se podría crear un visualizador de partidas mucho más avanzado sin la necesidad de modificar absolutamente nada del motor de juego. Esta parte debería ser contemplada en el plan de negocio.

5.3.3 Análisis de los usuarios y de su trabajo

Para detectar posibles fallos de diseño en la funcionalidad del juego se hace imprescindible estudiar a los usuarios. Voy a explicar la situación con un ejemplo:

> Se descubre al cabo de 2 semanas que un 70% de los usuarios tiene un intelecto que hace lo mismo y resulta una acción ilógica (por ejemplo, corren siempre en círculos porque descubren que en ese caso hay un error de cálculo y no se emiten sonidos). Gracias a la observación de los usuarios se podría llegar a detectar un problema de la aplicación.

Este proceso se simplificaría con las herramientas adecuadas. El sistema de foro que se pensó en el diseño de funcionalidades (página 15) sería perfectamente válido para que los usuarios reportaran estos problemas.

5.3.4 Adicción de nuevas características al motor de juego

El motor de juego está especialmente diseñado para ser capaz de adoptar nueva funcionalidad con los mínimos cambios. La adicción de acciones es casi trivial, y podría

llevarse a cabo en función del estudio de los usuarios anterior: si los usuarios sienten que tienen pocos mecanismos para actuar mejor que sus rivales, se pueden añadir acciones, tipos de armas, tipos de unidades, etc.

5.3.5 Mejora del sistema de localización de usuarios

Puesto que en la versión actual se depende de una librería que no es suficientemente exacta (ver sección 4.3.5), se propone como mejora de la capacidad de localizar dónde se encuentran los usuarios descentralizar dicha capacidad. La gran mayoría de los dispositivos móviles de las últimas generaciones tienen incorporados chips que permiten la localización del dispositivo por GPS. Otra posible opción es basarse en la información que se puede obtener de las redes Wifi vecinas. Si se sabe qué redes Wifi son visibles desde un punto y dónde están dichas redes, se sabe dónde se encuentra el usuario. Este método es usado actualmente por Google.

6 Bibliografía

[1] : Valve (2012), "CounterStrike: Global Offensive" en Counter-Strike Blog. [En línea] Disponible en: http://blog.counter-strike.net/ [accedido el día 23/05/2012]

[2] : Activision (2012), "Call Of Duty" en Call of Duty. [En línea] Disponible en: http://www.callofduty.com/ [accedido el día 23/05/2012]

[3] : EA Games (2012), "Battlefield" en Battlefield. [En línea] Disponible en: http://www.battlefield.com/ [accedido el día 23/05/2012]

[4] : Ubisoft (2012), "Brothers in Arms: Furious 4" en Brothers in Arms. [En línea] Disponible en: http://brothers-in-arms.ubi.com [accedido el día 23/05/2012]

[5] : FX Interactive (2012), "Panzers II" en Panzers II. [En línea] Disponible en: http://www.fxinteractive.com/p093/p093.htm [accedido el día 23/05/2912]

[6] : Robocode Team (2012), "Robocode" en Sourceforge. [En línea] Disponible en: http://robocode.sourceforge.net/ [accedido el día 23/05/2012]

[7] : Twitter Engineering (2012), "MySQL at Twitter" en twitter.com. [En línea] Disponible en: http://engineering.twitter.com/2012/04/mysql-at-twitter.html [accedido el día 23/05/2012]

[8] : Oracle y afiliados (2011), "MySQL Cluster" en MySQL 5.0 Reference Manual. [En línea] Disponible en: http://dev.mysql.com/doc/refman/5.0/es/ndbcluster.html [accedido el día 23/05/2012]

[9] : The Apache Software Foundation (2009), "Welcome to Apache Cassandra" en apache.org. [En línea] Disponible en: http://cassandra.apache.org/ [accedido el día 23/05/2012]

[10] : Wikipedia Foundation (2012), "Artículos en la categoría «Motores de videojuegos»" en Wikipedia (en español). [En línea] Disponible en: http://es.wikipedia.org/wiki/Categor%C3%ADa:Motores_de_videojuegos [accedido el día 23/05/2012]

[11] : Dick Swan (2007), "Programming Solutions for the LEGO Mindstorms NXT - Which approach is best for you?" en Robot Magazine. [En línea] Disponible en: http://find.botmag.com/100701 [accedido el día 23/05/2012]

[12] : César Otero (2012), "Blizzard, obligada a lanzar más servidores hoy para aguantar el tráfico de Diablo III" en Meristation. [En línea] Disponible en:

http://www.meristation.com/es/pc/noticias/blizzard-obligada-a-lanzar-mas-servidores-hoy-para-aguantar-el-trafico-de-diablo-iii/1777361?utm_source=twitterfeed [accedido el día 23/05/2012]

[13] : V. R. Westmark, (2004) "A Definition for Information System Survivability" en Proceedings of the 37th Hawaii International Conference on System Sciences - 2004, páginas -. doi>10.1.1.106.5586

[14] : S. Louca, A. Pitsillides, G. Samaras, (1999) "On network survivability algorithms based on trellis graph transformations" en International Symposium on Computers and Communications -IEEE, páginas 1008 - 1023. doi>10.1.1.5.7228

[15] : Oracle (2012), "Conozca más sobre la tecnología Java" en Java.com. [En línea] Disponible en: http://java.com/es/about/ [accedido el día 23/05/2012]

[16] : K. R. Harisha (2012), "Software Development Process and Its Importance" en www.streetdirectory.com (software). [En línea] Disponible en: http://www.streetdirectory.com/travel_guide/135488/software/software_development_process_and_its_importance.html [accedido el día 23/05/2012]

[17] : E. Gamma, R. Helm, R. Johnson y J. Vlissides, (1997) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley

[18] : Mary Campione (1997), "Deciding What SecurityManager Methods to Override" en Custom Networking and Security. [En línea] Disponible en: http://journals.ecs.soton.ac.uk/java/tutorial/networking/index.html [accedido el día 23/05/2012]

[19] : Wikipedia Foundation (2012), "Enterprise JavaBeans" en Wikipedia (en español). [En línea] Disponible en: http://es.wikipedia.org/wiki/Enterprise_JavaBeans [accedido el día 23/05/2012]

[20] : S. Sicard, N. De Palma, D. Hagimont, (2006) "J2EE Server Scalability through EJB Replication" en SAC '06 Proceedings of the 2006 ACM symposium on Applied computing, páginas 778-785. doi>10.1145/1141277.1141455

[21] : E. Cecchet, J. Marguerite y W. Zwaenepoel (2002) "Performance and scalability of EJB applications" en ACM SIGPLAN. Edición 37, asunto 11, páginas 246 - 261. doi>10.1145/583854.582443

[22] : Wikipedia Foundation (2012), "Java Remote Method Invocation" en Wikipedia (en

español). [En línea] Disponible en: http://es.wikipedia.org/wiki/Java_Remote_Method_Invocation [accedido el día 23/05/2012]

[23] : MaxMind, Inc (2012), "GeoLite City" en maxmind.com. [En línea] Disponible en: http://www.maxmind.com/app/geolitecity [accedido el día 23/05/2012]

[24] : MaxMind, Inc. (2012), "GeoIP City Accuracy for Selected Countries" en maxmind.com. [En línea] Disponible en: http://www.maxmind.com/app/city_accuracy [accedido el día 23/05/2012]

[25] : Maxpowel (2011), "NoSQL: No es oro todo lo que reluce" en Con G de GNU. [En línea] Disponible en: http://www.congdegnu.es/2011/01/31/nosql-no-es-oro-todo-lo-que-reluce/ [accedido el día 23/05/2012]

[26] : Wikipedia Foundation (2012), "MySQL" en Wikipedia (en inglés). [En línea] Disponible en: http://en.wikipedia.org/wiki/MySQL [accedido el día 23/05/2012]

[27] : Emmanuel Cecchet (2004), "HOWTO use C-JDBC with Apache Derby" en apache.org. [En línea] Disponible en: http://db.apache.org/derby/binaries/HOWTO_CJDBC_Derby_v0.2.pdf [accedido el día 23/05/2012]

Otros libros que han sido de ayuda:

F. Marinescu, (2002) "EJB Design Patterns". Wiley Computer Publishing.

T. Hermansson y M. Åkerlund, "EJB - A Deployment Evaluation". Tesis en la Universidad de Umeå.

M. Fowler, K. Beck, J. Brant, W. Opdyke y don Roberts, (2002) “Refactoring: Improving the Design of Existing Code”. Addison-Wesley

Antonio Berthier. (2006) “El sistema de Referencias Harvard” en ConocimientoySociedad. [En línea]. Disponible en: http://www.conocimientoysociedad.com/Harvard.html [Consultado el 15 de junio de 2008].

Capítulo 6: Bibliografía

7 Apéndices

7.1 Implementación del Security Manager del motor de juego

```
package motor;
import java.io.FileDescriptor;
import java.net.InetAddress;
import java.security.Permission;
public class SeguridadUsuarios extends SecurityManager {
     boolean b;
     public SeguridadUsuarios(){
          b=true;
     }
     void setAccesible(boolean c) {
/*El valor de b será TRUE cuando se ejecute código del motor, y será false
cuando el código sea del usuario*/
          b=c;
     }
     private void nain() {
//Este método deniega la petición de permisos
          throw new SecurityException();
     }
     @Override
     public void checkAccept(String host, int port) {
          if(b)
               super.checkAccept(host, port);
          else nain();
     }
     @Override
     public void checkAccess(Thread t) {
          if(b)
               super.checkAccess(t);
          else nain();
     }
     @Override
     public void checkAccess(ThreadGroup g) {
          if(b)
               super.checkAccess(g);
          else nain();
     }
     @Override
     public void checkAwtEventQueueAccess() {
          if(b)
               super.checkAwtEventQueueAccess();
          else nain();
     }
     @Override
```

```
public void checkConnect(String host, int port, Object context) {
     if(b)
          super.checkConnect(host, port, context);
     else nain();
}
@Override
public void checkConnect(String host, int port) {
     if(b)
          super.checkConnect(host, port);
     else nain();
}
@Override
public void checkCreateClassLoader() {
     if(b)
          super.checkCreateClassLoader();
     else nain();
}
@Override
public void checkDelete(String file) {
     if(b)
          super.checkDelete(file);
     else nain();
}
@Override
public void checkExec(String cmd) {
     if(b)
          super.checkExec(cmd);
     else nain();
}
@Override
public void checkExit(int status) {
     if(b)
          super.checkExit(status);
     else nain();
}

@Override
public void checkLink(String lib) {
     if(b)
          super.checkLink(lib);
     else nain();
}
@Override
public void checkListen(int port) {
     if(b)
          super.checkListen(port);
     else nain();
}
@Override
```

```
public void checkMulticast(InetAddress maddr, byte ttl) {
     if(b)
          super.checkMulticast(maddr, ttl);
     else nain();
}
@Override
public void checkMulticast(InetAddress maddr) {
     if(b)
          super.checkMulticast(maddr);
     else nain();
}
@Override
public void checkPackageAccess(String pkg) {
     if(!b){
          if(pkg.equals("java.lang"))
              nain();
     }
     super.checkPackageAccess(pkg);
}
@Override
public void checkPackageDefinition(String pkg) {
     if(b)
          super.checkPackageDefinition(pkg);
     else nain();
}
@Override
public void checkPermission(Permission perm, Object context) {
     if(b)
          super.checkPermission(perm, context);
     else nain();
}
@Override
public void checkPermission(Permission perm) {
     if(!b)
          nain();
}
@Override
public void checkPrintJobAccess() {
     if(b)
          super.checkPrintJobAccess();
     else nain();
}
@Override
public void checkPropertiesAccess() {
     if(b)
          super.checkPropertiesAccess();
     else nain();
}
@Override
```

```
public void checkPropertyAccess(String key) {
     if(b)
          super.checkPropertyAccess(key);
     else nain();
}
@Override
public void checkRead(FileDescriptor fd) {
     if(b)
          super.checkRead(fd);
     else nain();
}
@Override
public void checkRead(String file, Object context) {
     if(b)
          super.checkRead(file, context);
     else nain();
}
@Override
public void checkRead(String file) {
     if(b)
          super.checkRead(file);
     else nain();
}
@Override
public void checkSecurityAccess(String target) {
     if(b)
          super.checkSecurityAccess(target);
     else nain();
}
@Override
public void checkSetFactory() {
     if(b)
          super.checkSetFactory();
     else nain();
}
@Override
public void checkSystemClipboardAccess() {
     if(b)
          super.checkSystemClipboardAccess();
     else nain();
}
@Override
public boolean checkTopLevelWindow(Object window) {
     if(b)
          return super.checkTopLevelWindow(window);
     return false;
}
@Override
public void checkWrite(FileDescriptor fd) {
```

```
            if(b)
                super.checkWrite(fd);
            else nain();
        }
        @Override
        public void checkWrite(String file) {
            if(b)
                super.checkWrite(file);
            else nain();
        }
        @Override
        public void checkMemberAccess(Class<?> clazz, int which) {
            if(!b)
                super.checkMemberAccess(clazz, which);
        }
}
```

7.2 XML de configuración de partidas

```
<?xml version="1.0" encoding="UTF-8"?>
<mapa>
    <cajas>
        <caja>
            <esquina1>(-20,-20)</esquina1>
            <esquina2>(0,1044)</esquina2>
        </caja>
        <caja>
            <esquina1>(-20,-20)</esquina1>
            <esquina2>(1044,0)</esquina2>
        </caja>
        <caja>
            <esquina1>(1024,-20)</esquina1>
            <esquina2>(1044,1044)</esquina2>
        </caja>
        <caja>
            <esquina1>(-20,1024)</esquina1>
            <esquina2>(1044,1044)</esquina2>
        </caja>
        <caja>
            <esquina1>(20,20)</esquina1>
            <esquina2>(40,25)</esquina2>
        </caja>
        <caja>
            <esquina1>(20,60)</esquina1>
            <esquina2>(40,65)</esquina2>
        </caja>
        <caja>
            <esquina1>(40,25)</esquina1>
            <esquina2>(45,60)</esquina2>
        </caja>
    </cajas>
  <soldados>
    <equipo nombre="Carlos">
      <soldado nombre="Carlos1" intelecto="Zombie.java">
        <propiedad nombre="SALUD" tipo="float">120.00001</propiedad>
        <propiedad nombre="MAXSALUD" tipo="float">120.00001</propiedad>
        <propiedad nombre="DEFPROYECTIL" tipo="float">-5.25</propiedad>
        <propiedad nombre="VELCORRER" tipo="float">11.759999</propiedad>
        <propiedad nombre="DEFDISPARO" tipo="float">-5.25</propiedad>
        <propiedad nombre="VELANDAR" tipo="float">2.4</propiedad>
        <propiedad nombre="DISTOIDO" tipo="float">360.0</propiedad>
      </soldado>
      <soldado nombre="Carlos2" intelecto="Zombie.java">
        <propiedad nombre="SALUD" tipo="float">120.00001</propiedad>
        <propiedad nombre="MAXSALUD" tipo="float">120.00001</propiedad>
```

```
        <propiedad nombre="DEFPROYECTIL" tipo="float">-5.25</propiedad>
        <propiedad nombre="VELCORRER" tipo="float">11.759999</propiedad>
        <propiedad nombre="DEFDISPARO" tipo="float">-5.25</propiedad>
        <propiedad nombre="VELANDAR" tipo="float">2.4</propiedad>
        <propiedad nombre="DISTOIDO" tipo="float">360.0</propiedad>
      </soldado>
      <soldado nombre="Carlos3" intelecto="Zombie.java">
        <propiedad nombre="SALUD" tipo="float">120.00001</propiedad>
        <propiedad nombre="MAXSALUD" tipo="float">120.00001</propiedad>
        <propiedad nombre="DEFPROYECTIL" tipo="float">-5.25</propiedad>
        <propiedad nombre="VELCORRER" tipo="float">11.759999</propiedad>
        <propiedad nombre="DEFDISPARO" tipo="float">-5.25</propiedad>
        <propiedad nombre="VELANDAR" tipo="float">2.4</propiedad>
        <propiedad nombre="DISTOIDO" tipo="float">360.0</propiedad>
      </soldado>
      <soldado nombre="Carlos4" intelecto="Zombie.java">
        <propiedad nombre="SALUD" tipo="float">120.00001</propiedad>
        <propiedad nombre="MAXSALUD" tipo="float">120.00001</propiedad>
        <propiedad nombre="VELCORRER" tipo="float">8.4</propiedad>
        <propiedad nombre="DISTOIDO" tipo="float">360.0</propiedad>
      </soldado>
      <soldado nombre="Carlos6" intelecto="Zombie.java">
        <propiedad nombre="SALUD" tipo="float">120.00001</propiedad>
        <propiedad nombre="MAXSALUD" tipo="float">120.00001</propiedad>
        <propiedad nombre="VELCORRER" tipo="float">8.4</propiedad>
        <propiedad nombre="DISTOIDO" tipo="float">360.0</propiedad>
      </soldado>
    </equipo>
    <equipo nombre="Peter">
      <soldado nombre="Romualdez" intelecto="Antizombie.java">
        <propiedad nombre="DEFPROYECTIL" tipo="float">-5.5</propiedad>
        <propiedad nombre="VELCORRER" tipo="float">3.6000001</propiedad>
        <propiedad nombre="DEFDISPARO" tipo="float">-6.0</propiedad>
        <propiedad nombre="ANGDESVIO" tipo="float">0.026998064</propiedad>
        <propiedad nombre="DEFCUERPO" tipo="float">-8.25</propiedad>
        <propiedad nombre="TAMANO" tipo="float">0.8</propiedad>
        <propiedad nombre="DISTOIDO" tipo="float">240.0</propiedad>
        <propiedad nombre="DISTCUERPO" tipo="float">3.0</propiedad>
        <propiedad nombre="VELANDAR" tipo="float">1.6</propiedad>
        <propiedad nombre="HABCUERPO" tipo="float">165.0</propiedad>
        <arma nombre="AK-47" numarmas="1" municion="60" precio="3000"
preciomunicion="80">
          <propiedad nombre="TAMCARGADOR" tipo="int">30</propiedad>
          <propiedad nombre="MUNTOTAL" tipo="int">60</propiedad>
          <propiedad nombre="PRECISION" tipo="float">0.032725</propiedad>
          <propiedad nombre="DANO" tipo="float">30.0</propiedad>
          <propiedad nombre="MUNCARGADOR" tipo="int">0</propiedad>
          <propiedad nombre="TAMRAFAGA" tipo="int">2</propiedad>
          <propiedad nombre="ALCANCE" tipo="float">1000.0</propiedad>
```

```
        </arma>
      </soldado>
      <soldado nombre="Japet" intelecto="Antizombie.java">
        <propiedad nombre="ALCANCE" tipo="float">220.0</propiedad>
        <propiedad nombre="TAMANO" tipo="float">1.2</propiedad>
        <propiedad nombre="DEFPROYECTIL" tipo="float">-8.25</propiedad>
        <propiedad nombre="DEFCUERPO" tipo="float">-8.25</propiedad>
        <propiedad nombre="ANGVISTA" tipo="float">1.6493361</propiedad>
        <propiedad nombre="DISTCORTA" tipo="float">33.0</propiedad>
        <propiedad nombre="DEFDISPARO" tipo="float">-18.0</propiedad>
        <propiedad nombre="DISTOIDO" tipo="float">240.0</propiedad>
        <arma nombre="Desert Eagle" numarmas="1" municion="7" precio="3000"
preciomunicion="12">
          <propiedad nombre="TAMCARGADOR" tipo="int">7</propiedad>
          <propiedad nombre="MUNTOTAL" tipo="int">7</propiedad>
          <propiedad nombre="PRECISION" tipo="float">0.03</propiedad>
          <propiedad nombre="DANO" tipo="float">40.0</propiedad>
          <propiedad nombre="MUNCARGADOR" tipo="int">0</propiedad>
          <propiedad nombre="TAMRAFAGA" tipo="int">1</propiedad>
          <propiedad nombre="ALCANCE" tipo="float">300.0</propiedad>
        </arma>
      </soldado>
    </equipo>
  </soldados>
</mapa>
```

7.3 Implementación del intelecto "Zombie"

```
package NOMBREUSUARIO;
import entidades.Soldado;
import sentidos.*;
import acciones.*;
import intelecto.Intelecto;
import basic.*;
import acciones.*;
import sonidos.*;

public class Zombie extends Intelecto {
     public Zombie(){}
     /*
      * Orden de decisiones de un zombie:
      * 1- Si ve varios humanos: correr a por el m?s cercano
      * 2- Si escucha un sonido: girarse hacia el
      * 3- Si no escucha nada: andar y girar aleatoriamente.
      */
     @Override
     public Accion decidir(Vista vista, Oido oido,
               Terreno terreno, Radio radio){
     //Situacion 1
     Posicion p=null;
          p=masCercano(vista);
          if(p!=null){
               if(p.distancia()<estado.getDistCuerpo()+estado.getTamano()*2)
                   if(p.distancia()<estado.getDistCuerpo()/4.0f+
                              estado.getTamano()*2)
                         return new AccionGolpear(p);
                   else
                         return new AccionGolpearMovimiento(p);
               else
                   return new AccionCorrer(p);
          }

          //Situacion 2
          p=sonidoCercano(oido);
          if(p!=null){
                   return new AccionGirar(p);
          }
          //Situacion 3 -- Provisional
          return new
AccionGirar(Angulo.sumar(estado.getOrientacion(),1.6f));
     }

     public Posicion masCercano(Vista v){
          ListaLectora<Soldado> l=v.dameEnemigos();
```

```
        Posicion cercana=null;
        float dist=999999;
        for(int i=0;i<l.length();i++){
             Posicion tmp=l.get(i).getPosicion();
             if(tmp.distancia()<dist){
                 cercana=tmp;
                 dist=cercana.distancia();
             }
        }
        return cercana;
    }
    public Posicion sonidoCercano(Oido o){
        ListaLectora<Sonido> l=o.dameSonidos();
        Posicion cercana=null;
        float dist=999999;
        for(int i=0;i<l.length();i++){
             Posicion tmp=l.get(i).getPosicion();
             if(tmp.distancia()<dist)
                 cercana=tmp;
        }
        return cercana;
    }
}
```

7.4 Implementación del intelecto “Antizombie”

```
package NOMBREUSUARIO;
import entidades.Soldado;
import sentidos.*;
import acciones.*;
import intelecto.Intelecto;
import basic.*;
import acciones.*;
import armas.*;
import sonidos.*;

public class Antizombie extends Intelecto {
     public Antizombie(){}
     /* Orden de decisiones de un zombie:
      * 1- Si ve varios zombies y el mas cercano esta lejos: arrojarles una
granada
      * 2- Si ve varios zombies: disparar al m?s cercano
      * 3- Si ve zombies a poca distancia: andar hacia atras mientras
dispara.
      * 4- Si ve a alguien muy cerca: golpear
      * 5- Si escucha algo: girarse hacia ello
      * 6- Si no escucha nada: andar y girar aleatoriamente. */
     @Override
     public Accion decidir(Vista vista, Oido oido,
               Terreno terreno, Radio radio){
          //Situacion 1
          Posicion p=null;
          p=masCercano(vista);
          int numvistos=numVistos(vista);
          if(numvistos>1 && p.distancia()>30){
               Arrojable arr=armeria.getArrojables().get(0);
               if(arr!=null && arr.getNumero()>0){
                   return new AccionArrojar(p, arr);
               }
          }
          if(p!=null){
               Arma arma=armeria.getArmas().get(0);
               //Situacion 4
               if(p.distancia()<estado.getDistCuerpo()/1.5f){
                   return new AccionGolpear(p);
               }
               else{
                   //Situacion 2
                   if(arma!=null && arma.getMunicionCargada()>0){
                         //Situacion 3
                         if(p.distancia()<35f){
                               Angulo
op=Angulo.sumar(estado.getOrientacion(),Angulo.grados(180));
```

```
                              return new AccionDoble(new
AccionAndar(Posicion.getMax(op,estado.getVelAndar())),new
AccionDisparar(p,arma,arma.getBalasRafaga()));
                        }
                        return new
AccionDisparar(p,arma,arma.getBalasRafaga());
                  }
                  if(arma!=null && arma.getMunicionTotal()>0){
                        return new AccionRecargar(arma);
                  }
             }
        }
        //Situacion 5
        p=sonidoCercano(oido);
        if(p!=null){
             return new AccionGirar(p);
        }
        //Situacion 6 -- Solo gira
        return new AccionGirar(Angulo.sumar(estado.getOrientacion(),
             estado.getAngVista()));
    }
    public int numVistos(Vista v){
        return v.dameEnemigos().length();
    }
    public Posicion masCercano(Vista v){
        ListaLectora<Soldado> l=v.dameEnemigos();
        Posicion cercana=null;
        float dist=999999;
        for(int i=0;i<l.length();i++){
             Posicion tmp=l.get(i).getPosicion();
             if(tmp.distancia()<dist){
                 cercana=tmp;
                 dist=cercana.distancia();
             }
        }
        return cercana;
    }
    public Posicion sonidoCercano(Oido o){
        ListaLectora<Sonido> l=o.dameSonidos();
        Posicion cercana=null;
        float dist=999999;
        for(int i=0;i<l.length();i++){
             Posicion tmp=l.get(i).getPosicion();
             if(tmp.distancia()<dist)
                 cercana=tmp;
        }
        return cercana;
    }
}
```