



**Universidad Rey Juan Carlos**

Escuela Técnica Superior de Ingeniería Informática

Departamento de Ciencias de la Computación, Arquitectura de  
Computadores, Lenguajes y Sistemas Informáticos y Estadística e  
Investigación Operativa.

---

**Estudio comparativo de la escalabilidad de distintos  
entornos de programación para Sistemas  
Distribuidos**

---

**Trabajo Fin de Grado**  
Grado en Ingeniería del Software

— Autor —

Cristian Rodríguez Bernal

—Tutor—

Carlos Cuesta Quintero

11 de julio de 2015



*Al acercarse al misterioso hombre con un loro en el hombro...*

**Vendedor de mapas:** Perdona, ¿tienes un primo que se llama Sven?

**Guybrush:** No; pero una vez tuve un barbero llamado Dominique.

**Vendedor de mapas:** Se parece bastante. Vayamos al grano.



# Agradecimientos

Han sido largos meses de trabajo, de muchísimo esfuerzo y de largas horas mirando el monitor sin contar con todos los cambios en mi vida personal.

Me gustaría agradecer todo el apoyo que me han mostrado los miembros de mi familia, en especial la de mi padre que, aunque siempre me ha infravalorado para que me intentara superar en todo lo que hago, aquí me tiene, con un proyecto acabado y con ganas de comerme al mundo, y a mi madre y mi hermana, sufridoras de mis largas horas frente al ordenador.

También quiero agradecer su ayuda y comprensión al tutor de este proyecto, un gran profesor y un buen amigo (a pesar de que en que cada reunión habláramos más de música que del proyecto): Carlos E. Cuesta.

Agradecer a los pocos viejos compañeros de doble grado con los cuales coincidí todos mis años de carrera (Sergio, Aza y Juan), por nuestra gran labor como alumnos de la primera generación de nuestra doble titulación y a muchos otros amigos y compañeros que he conocido estos últimos años en la Universidad: Fermín, Sandra, Alberto, Christian, Dani, Johnny, David, Alex, Dani Lord Junglear, Sandy, ...

Para finalizar, quiero darle muchas gracias a Clara por mostrarme todo su apoyo en mis buenos y malos días, sin importar su estado anímico, siempre dispuesta a ayudarme a levantarme, haciendo que cada día sea mejor persona a su lado.

Por todo esto, muchas gracias a todos (incluidos los futuros y esperados lectores)...



# Resumen

Hace bastantes años que los juegos multijugador están en boca de todos. No obstante, debido a los cambios en hardware, en especial en los procesadores de alto rendimiento, se ha tenido que reducir la frecuencia de procesamiento para evitar llegar a sobrecalentarlos. Por ello, las arquitecturas multiprocesador o multicore han sido el principal diseño de computadores modernos.

Este cambio de arquitectura requiere de otros paradigmas de desarrollo software/hardware más encaminados al procesamiento concurrente o en paralelo con el fin de mejorar y utilizar al 100 % todos los recursos del computador en uso.

Por ello, el papel principal de este presente trabajo es comparar y valorar varios modelos de programación (basado en agentes, basados en hilos, basados en procesos, . . .) en multitud de lenguajes y frameworks de programación, con el fin de comparar la facilidad de desarrollo, librerías nativas, rendimiento, entre otros aspectos.





# Índice general

<b>Agradecimientos</b>	<b>III</b>
<b>Resumen</b>	<b>v</b>
<b>1. Introducción</b>	<b>1</b>
1.0.1. Metodología . . . . .	3
1.0.2. Estructura de la memoria . . . . .	4
<b>2. Objetivo</b>	<b>5</b>
<b>3. Contexto</b>	<b>9</b>
3.1. Versión inicial . . . . .	9
3.2. Versión final . . . . .	10
<b>4. Arquitectura</b>	<b>13</b>
4.1. Introducción . . . . .	13
4.2. Protocolo de intercambio de información . . . . .	13
4.3. Arquitectura global . . . . .	14
4.4. Desarrollo de los clientes . . . . .	15
4.4.1. Introducción . . . . .	15
4.4.2. Desarrollo . . . . .	16
4.4.3. Cliente C++ . . . . .	17
4.4.4. Cliente de pruebas . . . . .	19
4.5. Desarrollo de los servidores . . . . .	22
4.5.1. Arquitectura común . . . . .	22
4.5.2. Paradigmas utilizados . . . . .	25
4.6. Implementaciones concretas . . . . .	32
4.6.1. Servidor NodeJS . . . . .	32
4.6.2. Servidor IOJS . . . . .	35
4.6.3. Servidor Go . . . . .	35

4.6.4. Servidor Python . . . . .	39
4.6.5. Servidor Python Twisted . . . . .	41
4.6.6. Servidor Ruby . . . . .	42
4.6.7. Servidor Ruby EventMachine . . . . .	45
4.6.8. Servidor C . . . . .	47
4.6.9. Servidor C++ . . . . .	48
4.6.10. Servidor D . . . . .	51
4.6.11. Servidor Java Apache MINA . . . . .	53
4.6.12. Servidor C# . . . . .	55
4.6.13. Servidor F# . . . . .	57
4.6.14. Servidor Akka (Scala) . . . . .	59
4.6.15. Servidor Akka (Java) . . . . .	62
4.6.16. Servidor Groovy . . . . .	63
4.6.17. Servidor Julia . . . . .	65
<b>5. Resultados Experimentales</b>	<b>69</b>
5.1. Estudio comparativo . . . . .	69
5.1.1. Comparativa Scala vs. NodeJS . . . . .	69
5.1.2. Comparativa Scala vs. Java . . . . .	70
5.1.3. Comparativa Groovy vs. Java . . . . .	71
5.1.4. Comparativa Java Akka y Apache Mina . . . . .	72
5.1.5. Comparativa Ruby y Event Machine . . . . .	72
5.1.6. Comparativa Python y Python Twisted . . . . .	73
5.1.7. Comparativa Java y C# (Concurrencia) . . . . .	73
5.1.8. Comparativa Groovy vs. Ruby . . . . .	73
5.1.9. Comparativa Scala vs. Go . . . . .	74
5.1.10. Comparativa Scala y F# . . . . .	75
5.1.11. Comparativa Go y NodeJS . . . . .	76
5.2. Pruebas de tiempo y rendimiento . . . . .	77
5.2.1. Cliente de prueba . . . . .	77
5.2.2. Comparativas JSON . . . . .	80
<b>6. Conclusiones y trabajos futuros</b>	<b>83</b>
6.1. Objetivos . . . . .	83
6.2. Resultados . . . . .	84
6.3. Trabajos futuros . . . . .	84
6.4. Reflexiones finales . . . . .	86

---

<b>Anexos</b>	<b>87</b>
<b>Bibliografía</b>	<b>94</b>



# Índice de figuras

1.1. Captura de videojuego Doom II en su modo multijugador . . . . .	2
4.1. Esquema aplicación global . . . . .	14
4.2. Personajes del juego . . . . .	16
4.3. Cliente gráfico (dos clientes) . . . . .	18
4.4. Diagrama de secuencia . . . . .	21
4.5. Ejemplo de fichero final de prueba . . . . .	22
4.6. Esquema básico clases servidor . . . . .	23
4.7. Diagrama casos de uso: Tareas principales . . . . .	24
4.8. Diagrama casos de uso: Tareas secundarias . . . . .	25
4.9. Esquema Modelo basado en procesos . . . . .	26
4.10. Esquema Modelo basado en hilos . . . . .	28
4.11. Esquema Modelo basado en actores . . . . .	29
4.12. Esquema Modelo basado en E/S asíncrona . . . . .	30
4.13. Esquema Modelo reactivo . . . . .	31
4.14. Esquema servidor NodeJS . . . . .	34
4.15. Esquema servidor Go . . . . .	37
4.16. Esquema servidor Python . . . . .	40
4.17. Esquema servidor Python Twisted . . . . .	42
4.18. Esquema servidor Ruby . . . . .	44
4.19. Esquema servidor Ruby EventMachine . . . . .	46
4.20. Esquema servidor C++ . . . . .	50
4.21. Esquema servidor D . . . . .	52
4.22. Esquema servidor Java Apache Mina . . . . .	54
4.23. Esquema servidor C# . . . . .	56
4.24. Esquema servidor Akka(Scala) UML y Actores, respectivamente . . . . .	60
4.25. Esquema servidor Akka(Java) UML y Actores, respectivamente . . . . .	63
4.26. Esquema servidor Groovy UML y Actores, respectivamente . . . . .	64
4.27. Esquema servidor Julia . . . . .	66

5.1. Comparativa código Java vs. código Scala . . . . .	70
5.2. Gráfica de rendimiento de servidores con base de datos . . . . .	78
5.3. Gráfica de rendimiento de servidores sin base de datos . . . . .	79
5.4. Gráfica de rendimiento de servidores global . . . . .	80
5.5. Tabla comparativa de tratamiento de JSON en los servidores implementados	81
6.1. Esquema Entidad/Relación SQL . . . . .	87
6.2. UML clases principales del cliente gráfico (no incluidas estructuras) . . . .	89
6.3. Acciones asíncronas en C# [Blu] . . . . .	89

# Capítulo 1

## Introducción

Durante bastante tiempo ha sido cuestionado cuál fue el primer videojuego, principalmente debido a las múltiples definiciones que se le ha dado al concepto “videojuego” en todos los años de existencia, pero se puede considerar como primer videojuego el Nought and crosses (OXO), desarrollado por Alexander S. Douglas en 1952, una versión del “Tres en raya”, con modo jugador contra máquina. No obstante, no es hasta 1958, cuando William Higinbotham crea el juego más conocido, Tennis for Two, un simulador de tenis de mesa para dos jugadores.

No fue sino hasta 1966, cuando Ralph Baer desarrolló, junto a Albert Maricon y Red Dabney, el primer proyecto de videojuego doméstico, Fox and Hounds, que evolucionaría hasta convertirse en la Magnavox Odyssey en torno a 1972. En este mismo año, se funda Atari con la primera versión comercial del juego de Higinbotham, al que llaman Pong. Con el transcurso de los años, se suceden diversas etapas, como la década de los 8 bits (1980-1989), donde se empezaron a poner de moda las máquinas recreativas. Es en esta década cuando nace Nintendo y Sega.

La década de los 16 bits (1990-1999) se centró en los juegos en 3D, siendo de sus principales videojuegos Wolfenstein 3D y Doom. También en esa década se creó la primera consola de Sony, la Playstation One.

En torno al año 2000, el mundo de las consolas es altamente competitivo, debido a la gran cantidad de consolas y videojuegos de todas las formas y colores, y con una gran cantidad de temáticas distintas.

En la época actual, con el auge de las nuevas tecnologías y la moda de los MMO o MMOG (Massively Multiplayer Online Game) como *DoTA2* o *Smite*, los viejos sistemas basados en enormes servidores codificados principalmente en C de la época de Doom II han pasado a la historia, debido principalmente al nacimiento (o resurgir, en algunos casos), de nuevas tecnologías, librerías y lenguajes de programación orientados a la web.



**Figura 1.1:** Captura de videojuego Doom II en su modo multijugador

Empezando desde los primeros lenguajes de programación como C pasando por lenguajes orientados a objetos (*C++*, *Java*, *C#*), los lenguajes funcionales (*Scala*, *F#*), lenguajes de scripting (*Ruby*, *Python*, *Groovy*) y otros lenguajes más modernos (*Julia*, *Go*, *NodeJS*, *IOJS*), este trabajo se centra en la investigación y el análisis del rendimiento y la escalabilidad de los lenguajes más utilizados en el mercado actualmente, comparando sus principales funcionalidades en el lado del servidor frente a un cliente de escritorio.

Estos lenguajes utilizados permiten centrarnos en varios puntos claves en un desarrollo de aplicaciones en red, como puede ser la práctica con Sistemas Distribuidos (capacidad de distribución de información y procesamiento entre nodos, soluciones colaborativas entre nodos, ...) o la búsqueda de escalabilidad (no todos los lenguajes y plataformas reaccionan de igual manera cuando el sistema está falto de recursos).

Otros puntos interesantes a tener en cuenta son la tolerancia a fallos, la existencia de diferentes paradigmas de programación, la facilidad para realizar transacciones hacia Bases de Datos o encontrar cuál es el tipo de Protocolo de intercambio de información adecuado para este tipo de proyectos.

Para realizar esta investigación se desarrollará un cliente de escritorio en forma de videojuego en 2D (es importante destacar que no se quiere comprobar el rendimiento con aplicaciones con alto rendimiento gráfico, sino que se quieren probar aplicaciones con alto rendimiento en conexiones con servidores) que se conectarán con los distintos servidores desarrollados (más de 12 servidores, algunos realizados con librerías nativas y otras con librerías de terceros para mejorar la productividad).



Debido a que se han utilizado diversos lenguajes, cada uno de ellos contiene su propia implementación de los protocolos de red, por lo que cada funcionamiento varía entre cada implementación.

La comunicación entre clientes y servidores se realizará mediante un paso de mensajes con unas reglas prefijadas, de forma que exista una arquitectura o marco común que permita comparar con mayor exactitud las distintas arquitecturas, modelos de desarrollo, lenguajes de programación y frameworks que permitan mejorar la facilidad de desarrollo. Es importante destacar la existencia de gran cantidad de paradigmas de desarrollo para programación en red, así como varios protocolos de intercambio de información. Por ello, se han seleccionado minuciosamente los paradigmas a utilizar (procesos, hilos, actores, programación asíncrona y modelo reactivo) y el protocolo de intercambio de mensajes (TCP).

Junto a la comparación de rendimiento entre todos los servidores, es importante remarcar algunas comparaciones clásicas entre lenguajes de programación y bibliotecas que aumentar la funcionalidad de estos (*Akka*, por ejemplo), lenguajes que trabajan sobre una misma máquina virtual (*Scala*, *Java*, *Groovy*), entre otros aspectos.

Finalmente, y no menos importante, se plantea también un análisis de la facilidad de desarrollo (librerías nativas, sintaxis, legibilidad, tolerancia a fallos) y un análisis sobre el manejo de objetos JSON complejos (existencia de librerías nativas, facilidad para realizar marshall/unmarshall, librerías de terceros, ...) en todos los lenguajes utilizados.

### 1.0.1. Metodología

Durante el desarrollo de este trabajo se han utilizado diversas metodologías de desarrollo, empezando inicialmente por una metodología basada en el modelo tradicional y el *Proceso Unificado*<sup>1</sup> con el fin de centrar las bases sobre las que se sustentarían los objetivos más importantes (Cliente gráfico y servidor *Go*. Se puede obtener más información al respecto en el Capítulo de objetivos<sup>5</sup>) del proyecto.

Debido a que este proyecto tiene una gran envergadura, se dividió de forma minuciosa los distintos componentes de la aplicación, de forma que cada siguiente evolución del proyecto

---

<sup>1</sup>El **Proceso Unificado de Desarrollo Software** o simplemente **Proceso Unificado** es un marco de desarrollo de software que se caracteriza por estar dirigido por casos de uso, centrado en la arquitectura y por ser iterativo e incremental. El refinamiento más conocido y documentado del Proceso Unificado es el Proceso Unificado de Rational o simplemente RUP.

El Proceso Unificado no es simplemente un proceso, sino un marco de trabajo extensible que puede ser adaptado a organizaciones o proyectos específicos. De la misma forma, el *Proceso Unificado de Rational*, también es un marco de trabajo extensible, por lo que muchas veces resulta imposible decir si un refinamiento particular del proceso ha sido derivado del Proceso Unificado o del RUP. Por dicho motivo, los dos nombres suelen utilizarse para referirse a un mismo concepto.[Pro]

ampliara los márgenes iniciales de la anterior capa.

Una vez que se pudo establecer la arquitectura global de todo el sistema a desarrollar, la metodología utilizaba se enfocó en un desarrollo dirigido por pruebas (al ya contar con un cliente contra el que ejecutar las tareas a realizar) más encaminado al mundo ágil<sup>2</sup>.

### 1.0.2. Estructura de la memoria

En las líneas posteriores a este capítulo, nos adentraremos de lleno en la memoria del proyecto en sí. Para facilitar la lectura y que se vuelva más amena, describiremos brevemente cada una de las secciones encontradas en éste.

- **Objetivos:** Antes de encararse al problema en cuestión, es importante marcar y separar los distintos objetivos a realizar en este proyecto. Es importante señalar el peso de cada uno de los objetivos separando entre objetivos principales (obligatorios) y secundarios (no concluyentes para el fin del proyecto en sí).
- **Contexto:** Es importante marcar el contexto del proyecto, desde cómo se inició a todos los puntos seguidos para llegar a la solución final aquí mostrada.
- **Arquitectura:** El punto más complejo y completo de todo el proyecto. En esta sección se detalla uno por uno los distintos paradigmas disponibles y la implementación del cliente y los distintos tipos de servidores.
- **Resultados experimentales:** Una vez detallados todos los servidores, en esta sección nos centraremos en analizar los resultados obtenidos.
- **Conclusiones:** Para finalizar de manera correcta el presente trabajo, se plantearán una serie de mejoras a futuro y algunas conclusiones al respecto frente a todo lo realizado en el proyecto.

---

<sup>2</sup>El **desarrollo ágil de software** refiere a métodos de ingeniería del software basados en el desarrollo iterativo e incremental, donde los requisitos y soluciones evolucionan mediante la colaboración de grupos auto organizados y multidisciplinarios. Existen muchos métodos de desarrollo ágil; la mayoría minimiza riesgos desarrollando software en lapsos cortos. El software desarrollado en una unidad de tiempo es llamado una iteración, la cual debe durar de una a cuatro semanas. Cada iteración del ciclo de vida incluye: planificación, análisis de requisitos, diseño, codificación, revisión y documentación. Una iteración no debe agregar demasiada funcionalidad para justificar el lanzamiento del producto al mercado, sino que la meta es tener una «demo» (sin errores) al final de cada iteración. Al final de cada iteración el equipo vuelve a evaluar las prioridades del proyecto.[Des]

# Capítulo 2

## Objetivo

Debido a que este trabajo es un proyecto de gran envergadura, se han seleccionado minuciosamente los objetivos principales como las tareas de mayor importancia a la hora de verificar la eficiencia de cada uno de los lenguajes utilizados. Por ello, la siguiente lista detallará cada uno de los retos a superar en el transcurso, junto a tareas no tan necesarias para la finalización de los problemas planteados.

- **Practicar con Sistemas Distribuidos**: Cada vez se requieren más conocimientos acerca de este tema a la hora de buscar un trabajo digno que difiera del trabajo habitual de desarrollo web. Para ello, tener un buen nivel de conocimiento sobre estas materias te abre muchas más salidas laborales, además de una mayor agilidad para encontrar y solucionar problemas.
- **Desarrollo de videojuegos**: Cada día salen al mercado más juegos online que requieren una distinta gestión del lado del servidor para el envío de información. Aunque el sistema habitual es el paso de mensajes, no todos los lenguajes de programación gestionan igual el servidor, por lo que este proyecto sirve también como comparativa en cuanto al nivel dificultad-sintaxis-librerías disponibles en cada lenguaje y plataforma.
- **Trabajar con Go**: El lenguaje Go (o *Golang*) es un lenguaje desarrollado por varios desarrolladores de Google. Este lenguaje provee de una sintaxis sencilla para controlar la concurrencia a través del uso de *gorutinas* y canales. Uno de los principales objetivos de este proyecto, al menos desde su inicio, fue ver el rendimiento de esta plataforma de programación tanto en el lado cliente como en el lado servidor.
- **Búsqueda de escalabilidad**: Existen muchos lenguajes y plataformas del mercado, pero no todas tienen el mismo rendimiento dependiendo del contexto en el que se utilicen, y no todas tienen gran facilidad de escalabilidad y gestión de procesos

internos. Por ello, queremos comprobar qué lenguajes se adaptan mejor al problema aquí planteado.

- **Plataformas de programación:** Debido al gran boom actual en el mundo de los lenguajes de programación, librerías y frameworks de desarrollo, queremos realizar un pequeño estudio comparativo sobre todas éstas librerías y lenguajes para, como ya hemos nombrado en un objetivo anterior, buscar lo más idóneo para el desarrollo de juegos multijugador.

Fuera de todos estos objetivos primarios, se encuentran varios objetivos secundarios, que son aspectos que nos gustaría poder tocar, aunque no con influencias en cierta parte en los resultados finales esperados. Podemos separar entre dos tipos de objetivos:

- **Objetivos secundarios generales:** Estos objetivos no forman parte del cliente implementado, pero proveen de mayor funcionalidad y de otros enfoques a la hora de desarrollar.
  - **Otros paradigmas:** Es un hecho que muchos lenguajes y librerías permiten desarrollar programación concurrente y/o paralela con distintos paradigmas de programación. Por ello, plantear un diseño y arquitectura distinta (pero sin salirse de la especificación original) puede facilitar (o empeorar), el rendimiento o facilidad de desarrollo.
  - **Tolerancia a fallos:** Muchos lenguajes de programación son difíciles de tratar y es muy fácil encontrarse con errores inesperados. Queremos comprobar la facilidad para que un programa pueda recuperarse de los errores.
  - **Uso de bases de datos:** En la realidad, las sesiones de los usuarios en los juegos suelen guardarse en bases de datos distribuidas. Para ello, se plantea un uso intensivo de una base de datos *MySQL* para guardar las posiciones de los distintos usuarios, con el fin de poder cargar sesiones anteriores.
- **Objetivos secundarios específicos de un juego:** Estos objetivos forman parte de la implementación del cliente, aunque solo aportan cierta funcionalidad que no es obligatoria para obtener los resultados experimentales finales.
  - **Sistema de combates:** Se plantea crear un sistema de combates basado en un juego simple utilizando un dado desde el lado servidor que genere las victorias y fracasos de dos jugadores.

- **Eventos basados en tareas programadas**: Una de las ideas para plantear el resultado y finalización de un combate es ejecuta eventos o tareas de un tiempo fijo. Para ello, cada lenguaje provee de distintas funcionalidades: *schedule*, *sleep*, *setTimeout*,...
- **Objetos en el escenario**: Para proveer de más dinamismo al juego, algunas zonas del mapa solo podrán ser accesibles si se llevan ciertos objetos encima. Por ello, se plantea un sistema basado en mochilas y objetos que pueden cogerse o soltarse por el mapa.



# Capítulo 3

## Contexto

### 3.1. Versión inicial

Para ponernos en el contexto de desarrollo de proyecto, es necesario empezar explicando cuál fue el origen con el que se formó este Trabajo de Fin de Grado.

Inicialmente se había planteado “toquetear” con el mundo de los Sistemas Distribuidos desarrollando una aplicación Peer To Peer<sup>1</sup>, pero dos puntos clave de este proyecto eran bastante difíciles de solucionar:

- La primera eran las librerías desactualizadas y la poca información encontrada en otras librerías. Además, destacar la dificultad de compilación y de entendimiento de *C* en aquellos momentos.
- La segunda era la búsqueda de servicios *Cloud*<sup>2</sup> con los cuales poder hacer pruebas. Amazon y su *Amazon Web Services* era capaz de proveernos de un servicio mínimo para realizar pruebas, pero se nos quedaba pequeño, y no podíamos empezar a gastar dinero si no sabíamos si el trabajo se podría avanzar o no.

Durante una de las reuniones de seguimiento del proyecto, debido a la moda de los videojuegos online, se planteó una posible alternativa para el indefinido proyecto. En esa reunión se puso como reto el desarrollo de un juego online utilizando el lenguaje de programación Go.

Este lenguaje facilitaba enormemente el desarrollo de aplicaciones distribuidas debido a una API limpia y elegante, y la incorporación de funcionalidades para paso de mensajes, concurrencia y paralelismo. Debido a que *Go* podía llamar a código compilado en *C*,

---

<sup>1</sup>Una **red peer-to-peer**, **red de pares**, **red entre iguales** o **red entre pares** (**P2P**, por sus siglas en inglés) es una red de computadoras en la que todos o algunos aspectos funcionan sin clientes ni servidores fijos, sino una serie de nodos que se comportan como iguales entre sí.

<sup>2</sup>Paradigma que permite ofrecer servicios de computación a través de Internet.

decidimos en un primer momento desarrollar un juego completo usando esta tecnología y además desarrollar todo el servidor, usando para ello un wrapper de *CSFML*<sup>3</sup> llamado *GOSFML*<sup>4</sup>, bajo un modelo Cliente-Servidor con conexión TCP.

En pocos días se codificó gran parte del código del cliente, pero muchas funcionalidades contenidas en la versión en *C* (y en la de *C++*, que es el origen de la librería original) no estaban realmente optimizadas o no funcionaban correctamente, tuvimos que descartar rápidamente esta idea.

La siguiente idea fue utilizar el mismo servidor desarrollado en *Go* con un cliente en *C++* usando la librería original, *SFML*<sup>5</sup>. La funcionalidad básica del cliente se codificó bastante rápido, pero el conocimiento de concurrencia en *Go* por aquel entonces era escaso, teniendo que parar y buscar otras alternativas. De la alternativa encontrada se empezó a formar la versión final.

## 3.2. Versión final

Como ya se ha comentado en los objetivos principales de este proyecto, queremos ver el comportamiento de diversos lenguajes y frameworks, tanto usando las bibliotecas nativas de los lenguajes, como usando otras librerías que cambian el modelo de concurrencia del lenguaje hasta puntos insospechables. Llegados a este punto nos inspiramos a la hora de desarrollar en el modelo incremental de desarrollo de software propuesto por Harlan Mills en el año 1980, ya que no sabíamos hasta qué punto podríamos llegar, motivo por el cual tampoco utilizados los procesos ágiles, porque un lenguaje o librería podría llevar más tiempo que otra, y los ciclos serían muy oscilantes.

Debido a que hay muchos lenguajes y librerías en el mercado, se decidió desde un primer momento solamente codificar los servidores en los lenguajes utilizados en la carrera, pero debido a que la tecnología más utilizada durante la doble carrera fue Java, se descartó su uso (debido en parte al poco interés en este lenguaje y a que ya había programado algunos servidores TCP y no suponían ningún reto) hasta el descubrimiento de *Java Mina* mucho más adelante, que cambiaba en parte el modelo de desarrollo del servidor. Los lenguajes utilizados en esta primera vuelta fueron dos: *Go* y *NodeJS*, lenguajes que en ese momento eran los que estaban más de moda y que me abrirán otras salidas laborales, fuera del negocio habitual de las consultorías, donde principalmente se desarrolla en *Java* y, en

---

<sup>3</sup><https://github.com/SFML/CSFML>

<sup>4</sup><https://bitbucket.org/krepa098/gosfml2/wiki/Home>

<sup>5</sup>[www.sfml-dev.org/](http://www.sfml-dev.org/)



menor medida, *.NET*.

En una segunda iteración se empezaron a agregar otros lenguajes menos habituales en España, pero más centrados en la programación rápida de aplicaciones, generalmente web, como eran *Python* y *Ruby*, junto a una librería centrada en la programación dirigida por eventos para cada uno de los dos lenguajes (*Twisted* y *EventMachine*, respectivamente). Este punto es clave para este trabajo, debido a que estábamos comparando la programación basada en Threads nativos y un modelo basado en canales, respectivamente, con dos librerías propias de programación de eventos. Es destacable de esta sección que aunque *NodeJS* se comporte en cierta manera como programación basada en eventos de forma nativa, *Twisted* y *EventMachine* se basan en la misma idea, pero utilizando nativamente un lenguaje no centrado de forma general en la programación con callbacks. De esta segunda iteración, solo se descartó *Ruby* con bibliotecas nativas por problemas de rendimiento.

Llegados a este punto, nos adentramos en el mundo de los programas compilados, y es en ese punto donde se incluyen *C*, *C++* y *D* al proyecto. De estos tres lenguajes, solo dos se pudieron completar, principalmente porque la API propuesta por Windows para la programación con sockets era bastante escueta, mientras que la propuesta por las propias librerías de *D* y por las librerías de *SFML* para *C++* eran bastante más completas. Cabe destacar de esta iteración que se descartó el usar una distribución Linux (que facilitaría el servidor en *C*), debido a que en la siguiente iteración entraría en escena la plataforma *.NET*, cuyo soporte por aquel (2015) entonces en otros Sistemas Operativos diferentes a Windows era bastante deficiente o nulo.

En la antepenúltima fase de desarrollo, se centró la idea en torno a los lenguajes prioritarios en los Sistemas de Información en España: *Java* y *.NET*. Como ya se comenta en líneas anteriores, no se quiso desarrollar un servidor TCP enteramente con la API nativa de *Java*, por lo que se decidió utilizar *Apache Mina*. En cuanto a *.NET*, se centró el desarrollo en ver cómo funcionaban los métodos y funciones asíncronas que provee el lenguaje. Esta nueva incorporación supuso un punto de ruptura en elegir el Sistema Operativo más adecuado para el despliegue de todos los servidores desarrollados hasta ese momento, centrando todo en un Sistema Operativo Windows 8.1 de 64 bits (aunque con algunas librerías en 32 bits).

Debido al auge de la programación funcional, principalmente en el ámbito del Data Mining, se decidió añadir varios lenguajes funcionales al proyecto, en concreto: *F#* y *Scala*. Es en esta parte del desarrollo donde se volvió a utilizar *Java*, pero esta vez con

las librerías de *Akka*, con el fin de comparar la rapidez de programación y de respuesta entre *Java* y *Scala* usando una librería común basada en actores.

En la última iteración se empezaron a mirar otros lenguajes que se podrían complementar con el resto de lenguajes y librerías utilizadas en el proyecto, pero debido al escaso tiempo para dedicarle a la programación y desarrollo (y a estudiar un nuevo lenguaje), a la escasa información sobre estos nuevos lenguajes en la red y a la gran cantidad de versiones de cada lenguaje a utilizar, se descartaron gran parte de ellos. Estos lenguajes y librerías son las siguientes: *Julia*<sup>6</sup>, *Elixir*<sup>7</sup> (o *Erlang*<sup>8</sup>), *Vert.x*<sup>9</sup>, *Nim*<sup>10</sup> y *Rust*<sup>11</sup>, de los cuales solo se llegó a implementar y “probar” el primero.

---

<sup>6</sup><http://julialang.org/>

<sup>7</sup><http://elixir-lang.org/>

<sup>8</sup><http://www.erlang.org/>

<sup>9</sup><http://vertx.io/>

<sup>10</sup><http://nim-lang.org/>

<sup>11</sup>[www.rust-lang.org/](http://www.rust-lang.org/)

# Capítulo 4

## Arquitectura

### 4.1. Introducción

En esta sección nos centraremos en la arquitectura y desarrollo de los clientes y de todos los servidores y la implementación de todos los objetivos planteados en líneas anteriores.

Primeramente se hablará brevemente sobre el protocolo de intercambio de información que se utilizará como método de comunicación entre los clientes y servidores, asimismo, se incluirá una razón de peso para determinar que la elección ha sido la acertada.

A continuación se detallará la arquitectura e implementación del cliente gráfico y el cliente de pruebas utilizado para obtener los resultados experimentales. Nos centraremos en la elección una librería gráfica adecuada, pasando por la arquitectura básica y a su posterior implementación, además del algoritmo básico utilizado para pintar y detectar las distintas acciones del juego.

Finalmente, nos centraremos en el desarrollo de cada uno de los servidores a analizar. Por cada uno de ellos, hablaremos primeramente sobre el contexto del lenguaje de programación/framework (historia, principalmente), sobre la arquitectura interna y algunas notas importantes acerca de la implementación y una serie de pros y contras.

### 4.2. Protocolo de intercambio de información

En el Protocolo de Internet, encontramos dos tipos de tráfico: **TCP** (*Transmission Control Protocol* o *Protocolo de Control de Transmisión*) y **UDP** (*User/Universal Datagram Protocol* o *Protocolo Universal de Datos/Usuario*). Mientras que TCP está orientado a conexiones, donde una vez establecida la conexión, la información se puede transmitir en ambas direcciones. UDP por el contrario es un protocolo de Internet más sencillo, sin necesidad de conexiones. Con él se pueden enviar múltiples mensajes en grupos (paquetes) de datos (aunque pueden llegar desordenados).

Debido a que el cliente gráfico de prueba requiere que los mensajes lleguen ordenados y se mantenga una conexión directa entre ambos extremos, la elección elegida es TCP.

### 4.3. Arquitectura global

Para centrar los límites y el marco común de todo el proyecto, se ha establecido una arquitectura global que se mantendrá durante todos los servidores implementados (ver Figura 4.1).

En esta arquitectura tenemos los siguientes componentes:

- **Servidor:** Punto principal a tratar en este proyecto (como ya se ha comentado varias veces atrás). En la figura 4.1 se ha utilizado una separación entre “*Core*” y “*Threads*”. Si bien es cierto que no se utilizarán threads para todos los servidores, esta separación funciona perfectamente para conseguir distinguir entre el servidor en sí (escuchador) o “*Core*” y los diferentes clientes conectados (sockets) o “*Threads*” como si de un servidor HTTP se tratase. No obstante, como hemos remarcado en líneas anteriores, utilizaremos conexiones basadas en TCP.
- **Base de Datos:** Como se establecieron en los objetivos secundarios, se quiere utilizar una base de datos con el fin de poder guardar los distintos datos que se utilizan en la aplicación. No obstante, como se indicó en esos objetivos, no es un objetivo principal del proyecto, por lo que no todos los servidores presentan este componente en su arquitectura.
- **Cliente:** Centro principal de pruebas del proyecto. Mediante una aplicación gráfica (desarrollada principalmente con *OpenGL* y *C++*) y una conexión activa TCP, clientes y servidores se encargarán de comunicarse.

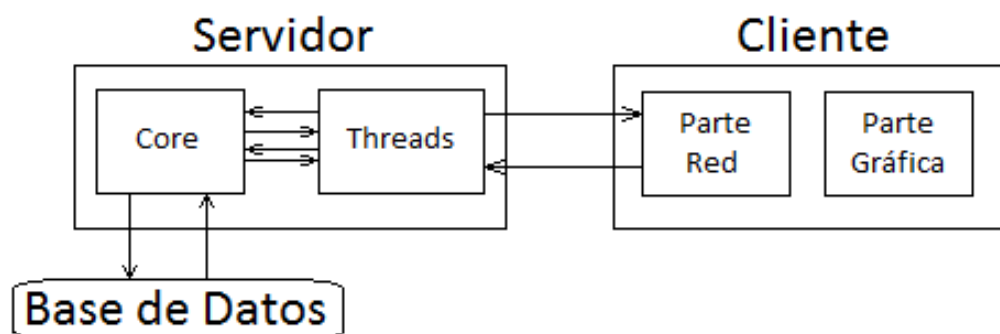


Figura 4.1: Esquema aplicación global

## 4.4. Desarrollo de los clientes

El principal centro de pruebas se centra en el mundo del cliente. En esta sección nos centraremos en dar una pequeña introducción a las librerías disponibles y a explicar la arquitectura genérica del cliente junto a las dos implementaciones disponibles más el cliente de pruebas utilizado para comprobar el rendimiento de los servidores.

### 4.4.1. Introducción

La idea principal a desarrollar en el lado cliente era un juego simple donde dos o más jugadores podían ver el movimiento de cada uno de los personajes, con el fin de realizar un estudio comparativo sobre rendimiento y escalabilidad de servidores TCP.

Para ello, se plantearon dos alternativas para desarrollar el cliente utilizando *C++*:

- **SDL (Simple DirectMedia Layer<sup>1</sup>):** Se trata de un conjunto de bibliotecas desarrolladas en el lenguaje de programación *C* que proporcionan funciones básicas para realizar operaciones de dibujo en dos dimensiones, gestión de efectos de sonido y música, además de carga y gestión de imágenes. Fueron desarrolladas inicialmente por Sam Lantinga, un desarrollador de videojuegos para la plataforma GNU/Linux.
- **SFML (Simple and Fast Multimedia Library<sup>2</sup>):** Se trata de una API portable, escrita en *C++*. Su propósito principal es ofrecer una biblioteca alternativa a la biblioteca *SDL*, usando un enfoque orientado a objetos. Gracias a sus numerosos módulos, SFML puede ser usada como un sistema mínimo de ventanas para interactuar con *OpenGL*<sup>3</sup> o como una biblioteca multimedia cuyas funcionalidades permiten al usuario crear videojuegos y programas interactivos. Provee de una completa funcionalidad multiplataforma, sobre todo en el tema red que es el tema que nos interesaba. Tiene gran cantidad de bindings, como *Python*, *Ruby*, *.Net*, *C* o *Java*.

Dentro de *SFML*, aunque el lenguaje de desarrollo original eran en *C++*, se encontraban diversos wrappers, de los cuales se decidió utilizar dos: *Python* y *C#*, aunque finalmente, debido a falta de tiempo, solo se implementó en *C++* casi al 90% y en *Python* solo los movimientos básicos (conexión, movimiento y desconexión) para demostrar la portabilidad del juego y la independencia del lenguaje de programación. A pesar de que se portó gran parte del código *C++* a *Python*, la versión del cliente en *Python* entregada en el disco adjunto a la memoria se encuentra incompleta.

---

<sup>1</sup><https://www.libsdl.org/>

<sup>2</sup><http://www.sfml-dev.org/>

<sup>3</sup><https://www.opengl.org/>

### 4.4.2. Desarrollo

- **Control de paso de mensajes con el servidor:** El principal sistema de comunicación entre los clientes y los distintos servidores es un paso de mensajes reglado bajo la notación JavaScript Object Notation (*JSON*), la cual facilita la serialización y deserialización en ambos lados.
- **Parte gráfica:** Ambos clientes utilizan las librerías SFML correspondientes (*SFML* y *pySFML*, respectivamente). Las dos versiones de la librería tienen una API muy parecida, por lo que es muy sencillo traducir de un lenguaje a otro (salvando las distancias, como por ejemplo los pasos por referencia, o los tipos de estructuras nativas de los lenguajes, entre otros)
- **Personajes del juego:** Para simular la existencia de varios tipos de jugadores, se han tomado dos tipos de personajes distintos (una quimera y un dragón), con el fin de diferenciar a los jugadores (el tipo de personaje se calcula dependiendo del identificador entero del usuario en el servidor, siendo la quimera si es un número par y el dragón en caso contrario).

Por motivos de diseño, la visualización del resto de clientes conectados en un cliente consiste en un cuadrado de un color determinado (cuatro colores distintos). Junto a esto, los dos personajes tienen un control distinto (Flechas vs. WASD), con el fin de poder mover a dos personajes sin necesidad de estar con el foco en ambas ventanas y probar el movimiento de los dos a la vez.



**Figura 4.2:** Personajes del juego

- **TileMap:** La estructura básica para separar los diferentes tiles de nuestro mapa es una matriz de Vertex Array <sup>4</sup>. La carga del mapa se hace a través un objeto JSON que contiene los siguientes elementos:
  - ★ **Tileset:** Cadena de números comprendidos entre 0 y 7, separados por “,”.
  - ★ **Height:** Número de tiles por cada columna.

---

<sup>4</sup>Vertex Array: Objeto OpenGL que agrupa los datos de los vértices (coordenadas, normales, coordenadas de textura, color de los vértices, ...).

★ **Width:** Número de tiles por cada fila.

### 4.4.3. Cliente C++

El cliente gráfico está compuesto por diversas clases para separar la creación de nodos, Tilemap, marshall/unmarshall de JSON y el hilo principal de ejecución, siendo éste último el código más completo y complejo de toda esta parte del proyecto. Debido a esta división en clases, podemos utilizar esta estructura (ver Figura 6.2) para descomponer estas clases como subsecciones de esta sección.

- **TCPEvents:** Se trata de módulo o pequeña librería para realizar las etapas de Marshall y Unmarshall de objetos JSON. Para reducir la complejidad de este módulo, se han utilizado estructuras de *C* en lugar de clases de *C++*. Como las bibliotecas nativas no incluyen soporte para el uso de JSON, se ha utilizado la librería *Jzon*<sup>5</sup>. Para separar los envíos hacia el servidor desde el cliente y viceversa, se han separado las distintas estructuras bajo dos namespace:
  - **ClientToServer:** Estructuras con un único método para generar mensajes en formato JSON que se enviarán al servidor. Estructuras disponibles: *TInitWName*, *TGetObject*, *TFreeObject*, *TMove*, *TExit*, *TFight*, *TDiePlayer*.
  - **ServerToClient:** Estructuras para generar struct con el contenido de los mensajes JSON recibidos del servidor. Estructuras disponibles: *TMove*, *TNew*, *TExit*, *TFinishBattle*, *TRemObject*, *TAddObj*, *TGetObjFromServer*, *TLiberateObj*.
- **TileMap:** Esta clase contiene cargado el mapa actual del juego en memoria. Utilizando objetos *VertexArray*, se cargan los diferentes tiles del mapa. Para facilitar el pintado, esta clase deriva de las clases *sf::Drawable* y *sf::Transformable*, ambas originarias de SFML.
- **Key:** Clase de entidad<sup>6</sup> para guardar las referencias a los objetos (llaves) que se encuentran dispersados por el mapa. Los atributos de esta clase son las posiciones, visibilidad, disponibilidad o identificador, además del sprite y textura del tile en cuestión.
- **Animation y AnimatedSprite:** Estas dos clases son códigos desarrollados por Maximilian Wagenbach, y se encuentran incluidos en la wiki interna del repositorio

---

<sup>5</sup>Librería Open Source desarrollada por Johannes Häggqvist. Permite generar parsear objetos dado un JSON o generar mensajes JSON a partir de objetos en *C++*. Link de descarga: <https://github.com/Zguy/Jzon>

<sup>6</sup>Objetos que pueden ser “dibujados”.

de SFML en Github. La clase *Animation* guarda los diferentes estados de una animación dados dos puntos (esquina superior izquierda y esquina inferior derecha) y el número del frame), mientras que la clase *AnimatedSprite* engloban las animaciones y permiten cambiar de frame o el frametime (tiempo entre dos frames). Ambos códigos se encuentran en <https://github.com/SFML/SFML/wiki/Source:-AnimatedSprite>.



**Figura 4.3:** Cliente gráfico (dos clientes)

- **PlayerSprite:** Debido a que no han requerido animaciones para el resto de clientes conectados, esta clase provee un nodo de entidad que permite pintar e identificar a cada cliente.
- **Source:** Clase principal. En esta clase nos encargamos de cargar todos los datos en memoria del TileMap, clientes, objetos, .... Antes de entrar al bucle de ejecución del juego, nos encargamos de realizar una conexión bajo TCP hacia un puerto. Para inicializar el juego, una vez que se realiza la conexión con éxito, pedimos al cliente su nombre de usuario.

En el bucle de ejecución, encontramos dos salidas distintas, dependiendo de si estamos en estado de combate o en el overworld (el mapa del juego).

- **Combate:** En este flujo de actividad, se desarrollan diversas animaciones simulando que se están realizando cálculos en el cliente, aunque realmente solo se queda a la espera de la resolución del combate por parte del servidor.
- **Overworld:** Flujo principal del cliente. En este se procesan gran parte de las entradas de los sockets y las E/S del ratón y teclado, además de casi todo el cauce gráfico de la aplicación. El orden de pasos de ejecución es el siguiente:
  1. Consultamos el estado del socket para saber si se han recibido datos y, en caso de obtener datos, según el campo “Action” del mensaje, realizamos diferentes tareas:



- *Move*: Este mensaje se recibe cuando movemos a un cliente según el identificador recibido.
  - *Fight*: Empezamos un combate contra otro cliente.
  - *New*: Se recibe cuando se conecta un nuevo jugador. El nuevo cliente se inserta en la lista de clientes conectados.
  - *Exit*: Este mensaje se recibe cuando otro cliente se desconecta. En este evento eliminamos el cliente de los usuarios conectados.
  - *Hide*: Cuando dos clientes entran en combate, el resto de clientes se esconden para que ningún otro cliente pueda interactuar con ellos.
  - *RemObj*: Este evento se produce cuando un cliente obtiene un objeto.
  - *AddObj*: Este evento se produce cuando un cliente suelta un objeto.
  - *GetObjFromServer*: Este mensaje se recibe cuando un cliente intenta obtener un objeto. Según el valor del campo “ok”, sabremos si el objeto se ha adquirido con éxito.
  - *sendInitFight*: Si un usuario recibe este mensaje es que otro cliente ha querido empezar un combate contra él. Como respuesta, se envía un mensaje para confirmar que el combate se ha aceptado.
  - *OkInitFight*: Si un usuario obtiene este mensaje, quiere decir que el otro cliente ha aceptado el combate.
2. Procesamos la entrada del teclado y del ratón. Los únicos eventos procesados son el cierre de la ventana (botón de cierre o tecla Esc) y las teclas P e Y para simular una doble y triple velocidad (respecto a la velocidad por defecto), respectivamente.
  3. Nos encargamos de mover a los personajes y realizar las colisiones correspondientes. Si el movimiento se ha realizado correctamente, enviamos un mensaje al servidor para avisar de un nuevo movimiento.
  4. Antes de acabar el bucle, movemos los distintos tipos de nodos gráficos (clientes, textos, ...), realizamos colisiones entre jugadores (para empezar combates) y colisiones con objetos (para obtener objetos). Finalmente, limpiamos la ventana, configuramos la cámara y pintamos todos los nodos gráficos.

#### 4.4.4. Cliente de pruebas

Inicialmente se plantearon diversos lenguajes de programación para desarrollar el cliente de pruebas (sin parte gráfica para no ralentizar el ordenador): *Python* y *Java*.

Aún después de optar por *Python*, el control de errores a veces resultaba bastante tedioso, y mucho más difícil de depurar. Es por ello por lo que se decidió migrar a *Java*, consiguiendo un código más fácil de leer e interpretar.

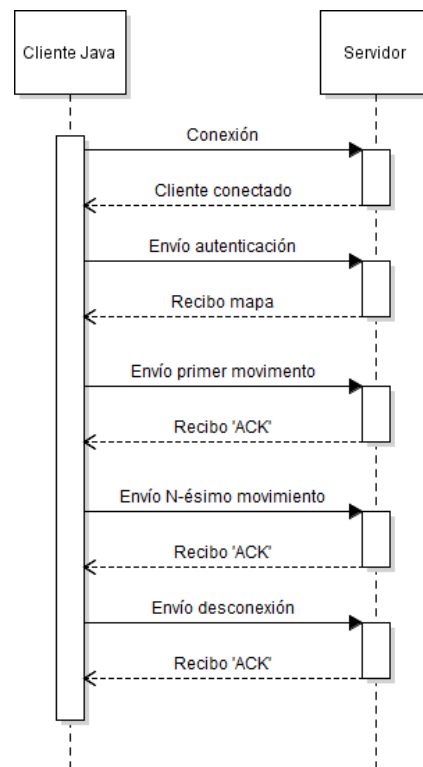
Es necesario indicar que se ha tenido que tocar parte del código original de cada uno de los servidores para implementar el modo de juego normal y el modo de juego para pruebas. La única diferencia entre ambos es que en el modo de pruebas envía determinados mensajes de respuesta, como simulación de los “ACK” en muchos servicios en red.

El algoritmo utilizado para desarrollar el cliente es bastante sencillo (pero muy eficaz) para comprobar el rendimiento a través del tiempo (ver Figura 4.4). Para ello, utilizando un número exponencial de usuarios con cierta cantidad de movimientos (todos con el mismo número), se realizan envíos entre clientes y servidor.

1. Se reciben como parámetros el número de clientes (**N**), el número de movimientos (**M**), el nombre del lenguaje o librería a utilizar (**L**) y el puerto de escucha (**P**).
2. Usando una combinación entre los tres argumentos, se genera un fichero .txt donde se guardarán los distintos valores de los tiempos entre cada mensaje enviado al servidor y su respuesta (basado en la teoría de ACKS).
3. Cada cliente conectado equivale a un hilo en ejecución, por lo que se crean un número **N** de ellos pasado como argumentos el número de movimientos. Debido a cómo se crean los hilos en Java (a diferencia de los *pthreads*<sup>7</sup>, que se ejecutan al crearlos), guardamos todos en una colección.
4. Una vez creados todos los hilos, lanzamos uno por uno y realizamos un join por cada uno para evitar que el proceso main finalice antes de acabar el algoritmo. Internamente, cada hilo guarda el tiempo actual del ordenador para calcular el tiempo total de ejecución de este “cliente”.
5. Por cada hilo, se envía una petición de conexión: { “Action”: “initWName”, “Name”: “User” + **N** }, guardando una referencia al tiempo del ordenador en el momento antes de enviar y la referencia al tiempo tras llegar el mapa. La diferencia entre ambos tiempos se guarda en una lista interna. Esta lista es utilizada para guardar las referencias a cada mensaje procesado. El hecho de que no se guarden directamente los mensajes en un fichero es debido a que la I/O a veces resulta algo lenta, por lo que el volcado en fichero se produce cuando el hilo va a finalizar su ejecución.

---

<sup>7</sup>Biblioteca que cumple los estándares POSIX y que nos permite trabajar con distintos hilos de ejecución (threads) al mismo tiempo.



**Figura 4.4:** Diagrama de secuencia

En esta lectura del mensaje recibido obtenemos el identificador del usuario, que utilizaremos para el siguiente paso.

6. De forma secuencial, se lanzan varias peticiones para simular que nuestros clientes se mueven por el mapa. Como no nos es útil la posición actual del usuario en los tests (aunque es necesario enviar datos para que los mensajes tengan información), generamos dos números aleatorios y los enviamos. Para verificar que el mensaje ha llegado correctamente, utilizamos el identificador del usuario obtenido en el punto anterior con el identificador devuelto por el “ACK”. Al igual que en el paso anterior, obtenemos el tiempo entre que se envía el mensaje, se recibe y se procesa y luego se añade a la lista.
7. En antepenúltimo lugar, mandamos un mensaje de finalización.  
Este mensaje (`{ “Action”: “exit”, “Id”: “Me” }`), difiere de los mensajes enviados en el cliente gráfico, debido a que enviamos un segundo atributo en el mensaje (Id). Al igual que en el paso anterior, comparamos el “ACK” (esta vez, el valor a comparar es “Me”), y lo guardamos en la lista.
8. Una vez completadas estas llamadas, se obtiene el tiempo total de ejecución y lo guardamos en la lista.

9. Para finalizar, creamos el fichero en la ruta anteriormente marcada en modo sobre escritura (en caso de existir el fichero) y guardamos cada mensaje en la lista en cada línea (4.5).

```
Init: 10
Move: user 13 time: 6
Move: user 13 time: 1
Move: user 13 time: 2
. . . . .
Move: user 13 time: 1
Move: user 13 time: 1
Move: user 13 time: 1
Move: user 13 time: 28
Exit: 3
Time Alive: 12086
```

**Figura 4.5:** Ejemplo de fichero final de prueba

## 4.5. Desarrollo de los servidores

El principal punto fuerte de este proyecto es la experimentación con distintos lenguajes y librerías en el lado servidor. En las siguientes secciones nos centraremos en la arquitectura básica, las diferentes soluciones actuales, el funcionamiento, desarrollo y pros y contras de cada uno de los implementados.

### 4.5.1. Arquitectura común

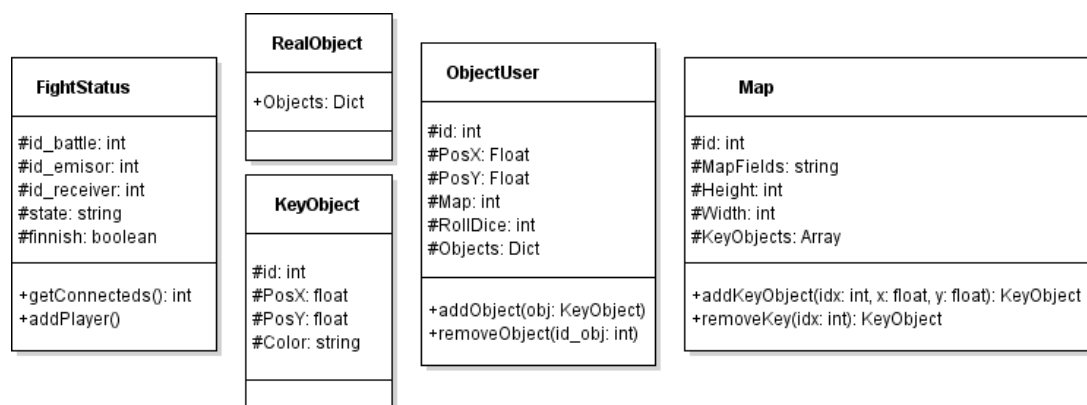
Debido a que para realizar una comparativa correcta entre distintas implementaciones, es necesario tener varios esquemas e ideas claras acerca de la arquitectura común de los servidores (ver Figura 4.6), los cuales serán los puntos a analizar en la comparativa final. Junto a esta arquitectura común (ver Figura 4.7), cada servidor llega a implementar funcionales secundarias (gestión de objetos, conexiones a bases de datos, combates de duración aleatoria, ...). Las principales funciones requeridas son:

- **Nuevas conexiones:** Cada vez que un nuevo usuario (o socket, para ser más preciso) se conecte a la aplicación, éste entra en estado conectado, con lo que puede interactuar con el servidor para obtener o recibir mensajes.
- **Login:** Para mantener un estado de identificación entre el cliente y servidor, se debe enviar un mensaje con el nombre de usuario a utilizar. Para esta prueba de concepto no se ha validado que se reciban dos usuarios con el mismo nombre (esto supone un agujero de seguridad, pero no se ha querido implementar esta tarea por

no ser totalmente necesaria para el funcionamiento de la aplicación).

Una vez que el cliente está logueado, se les envía el mapa con los objetos, posiciones de usuarios, .... Al resto de clientes se les envía la nueva conexión, con el identificador y posición del usuario.

- **Movimientos:** Cada vez que un cliente se mueve, se reenvía su posición al servidor, y éste se encargará de avisar al resto de clientes la nueva posición. Además, es necesario actualizar la posición del cliente en el servidor (en memoria o en base de datos).
- **Desconexión:** Con el fin de tener un protocolo seguro de desconexión, se necesita un paso de mensajes específico. Una vez que este mensaje de conexión al servidor, éste lo reenviará al resto de clientes. Como caso excepcional, si un socket se cierra de manera incorrecta sin este paso de mensaje, se eliminará el socket y también se enviará al resto de clientes la desconexión para que borren al cliente activo.

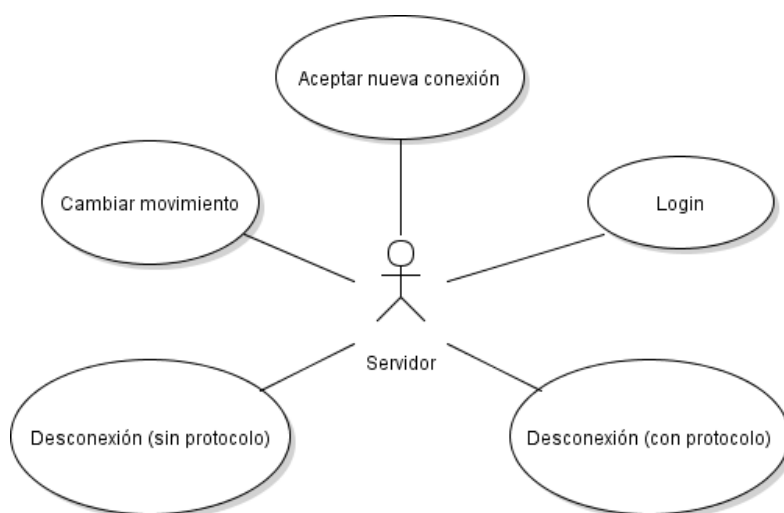


**Figura 4.6:** Esquema básico clases servidor

Junto a las tareas principales, según el tiempo dedicado a cada uno de los servidores, se realizaron varias acciones secundarias, cuyo papel no era necesario para el total funcionamiento y desarrollo de este trabajo (ver Figura 4.8):

- **Capturar objeto:** En el mapa de juego se encuentran esparcidos diferentes objetos, y algunos de estos pueden ser utilizados para entrar o realizar otras acciones.
- **Soltar objeto:** Con el fin de aumentar la cooperación entre jugadores, los objetos se pueden soltar por el mapa. Además, cuando un personaje muere, suelta el objeto en el último punto donde se encontraba éste.
- **Nuevo combate:** Cuando dos personajes se encuentran<sup>8</sup>, se puede realizar una

<sup>8</sup>Dos personajes “se encuentran” cuando el área de un sprite entra dentro del área conformada por otro sprite



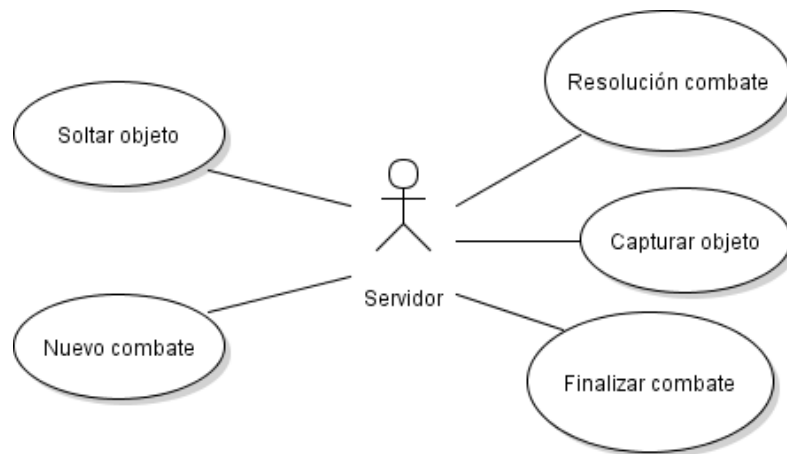
**Figura 4.7:** Diagrama casos de uso: Tareas principales

acción de combate. Este combate consiste en una tirada de dados, ganando el que consiga la mayor puntuación. Cuando dos personajes combaten, son borrados de la partida hasta que finalice combate.

- **Resolución del combate:** Mediante rutinas o procesos concurrentes/paralelos, el servidor reenvía a los combatientes el valor final del combate. Las posibles casos son: Un ganador - un perdedor o empate. El personaje perdedor es desconectado del juego.
- **Finalizar combate:** Cuando una persona pierde un combate, el servidor envía un mensaje para avisar al resto de usuario que se ha desconectado un usuario. Además, el ganador reaparece de nuevo en el juego.
- **Acceso a Base de Datos:** Para mantener la integridad de los datos y permitir una recuperación de la sesión anterior, se utiliza una base de datos SQL donde guardar los datos de objetos, clientes, . . . . El esquema E/R se puede ver en la Figura 6.1.

Fuera de las acciones del servidor, es importante destacar dos puntos:

- La parte de los datos no es realmente importante. Mucha información transmitida entre clientes y servidores trabaja como contenido dummy.
- La concurrencia es gestionada manualmente, de forma que tenemos un total control sobre todas las acciones. Aunque en algunos servidores se utilicen bases de datos, a pesar de la integridad referencial, todas las consultas y peticiones se gestionan en la lógica de negocio del servidor.



**Figura 4.8:** Diagrama casos de uso: Tareas secundarias

### 4.5.2. Paradigmas utilizados

Durante el desarrollo de los distintos servidores se han utilizado diferentes paradigmas de programación en red. Para ello, es importante describir los distintos paradigmas utilizados.

#### Basado en procesos

Un proceso o fork es un programa en ejecución, los cuales pueden tener réplicas del mismo programa ejecutado y son independientes, aunque sean instancias del mismo programa. Un proceso no es solamente un código del programa, ya que incluye otra información como contador de programa, registros del procesador o imagen de memoria.

Para controlar el estado del proceso, se utilizan diversos estados ya definidos: *Nuevo* (en creación), *Listo* (en espera de asignación a un procesador), *En ejecución* (ejecutando), *Bloqueado* (espera de suceso, como E/S, comunicación, sincronización, ...) y *Terminado* (ver Figura 4.9). Las principales particularidades de los procesos son:

- El proceso hijo tendrá su propio identificador de proceso único, además de su propia copia del descriptor de fichero de los padres.
- Los filelocks definidos por el proceso padre no serán heredados por el proceso hijo.
- Cualquier semáforo abierto por el proceso padre también podrá ser utilizado por el proceso hijo.
- El proceso hijo tendrá su propia copia de descriptores de cola de mensajes de los padres, además de su propio espacio de direcciones y memoria.

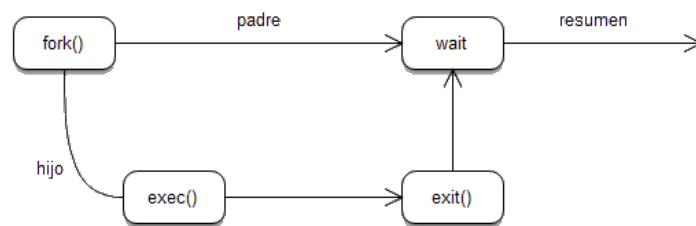
Los forks son más aceptados por encima de los threads por diversas razones:

- El desarrollo utilizando forks es mucho más fácil a nivel de implementación.
- El código basado en forks es mucho más mantenible y portable, además de ser mucho más fácil que depurar que uno basado en threads.
- Los procesos son mucho más privativos y seguros debido a que cada uno corre en una espacio de direcciones virtuales distinto. Si un proceso falla, no afecta al resto de procesos.

No obstante, los procesos tienen diversos problemas:

- Cada nuevo proceso debería tener su propio espacio de memoria, por lo que hace que tenga un tiempo de inicio y de parada bastante largo.
- Cuando hay dos procesos independientes que necesitan comunicarse entre ellos, ésta comunicación interprocesos es muy costosa.
- Cuando el proceso padre finaliza, los procesos hijos se convierten en procesos fantasma.
- Si no hay espacio suficiente en el sistema, el fork dará un error.

Algunas aplicaciones que utilizan un sistema basado en forks son: *PostgreSQL*<sup>9</sup>, *Apache2*<sup>10</sup> y *vsftpd*<sup>11</sup>.



**Figura 4.9:** Esquema Modelo basado en procesos

### Basado en hilos

Un hilo o thread es la unidad de procesamiento más pequeña que puede ser planificada por el Sistema Operativo. Los hilos requieren menos espacio que los procesos porque el sistema no necesita inicializar un nuevo espacio de memoria virtual, entre otros. Los sistemas basados en multiprocesador son los más utilizados para el desarrollo con hilos,

<sup>9</sup><http://www.postgresql.org/>

<sup>10</sup><http://httpd.apache.org/>

<sup>11</sup><http://vsftpd.beasts.org/>



debido a que se puede paralelizar o distribuir las distintas ejecuciones de un programa (ver Figura 4.10).

Los threads en un mismo proceso pueden compartir las instrucciones, casi todos los datos, ficheros abiertos (descriptores), señales y handlers, el directorio de trabajo actual y el usuario e identificador de grupo, entre otros. Además, cada thread tiene un identificador único, un stack pointer, un conjunto de variables locales, una máscara de señal, prioridad y una variable de retorno.

Las principales particularidades de los threads son:

- Para cada hilo, solo es necesario un único proceso/tabla de hilos y un planificador, siendo mucho más efectivos para sistemas multiprocesador o multicore.
- Todos los hilos de un proceso comparten el mismo espacio de direcciones, y ninguno de ellos debería saber quién ha sido el que lo creó.
- Los hilos son más efectivos en la gestión de memoria, ya que utilizan el mismo bloque de memoria de su padre en vez de crear uno nuevo y reducen la sobrecarga compartiendo partes fundamentales.

No obstante, el uso de thread también tiene bastantes problemas:

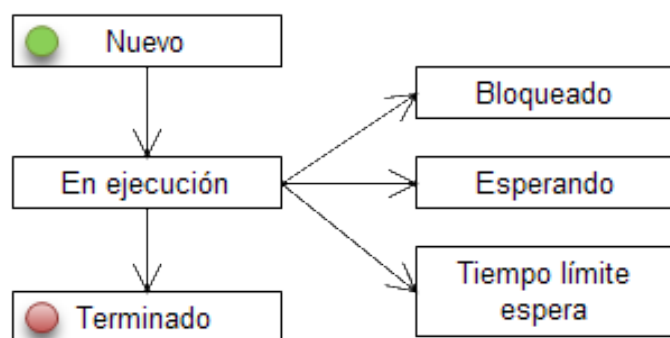
- **Condiciones de carrera:** La gran decepción con el uso de threads es la inexistencia de una protección natural frente a la ejecución de varios threads que utilizan los mismos datos a la vez sin saber que otros thread están utilizando a esto. Esto se llama condición de carrera. Para solucionar problemas inesperados, se utilizan herramientas como mutex y joins, con el fin de predecir el orden de ejecución y los resultados.
- **Seguridad en hilos:** No es fácil determinar si una parte de código es segura en cuanto a los hilos o no. No obstante, hay varios indicadores que requieren tenerse en cuenta para saber si es inseguro, como al acceso a variable globales o heap, la asignación o liberación de recursos con límites globales (ficheros, procesos, ...), accesos indirectos a punteros, o cualquier otro efecto colateral visible.

Para solucionar estos problemas, existen varias soluciones: código reentrante (interrupción de tareas, uso de variables locales), exclusión mutua (deadlocks, condición de carrera, inanición, ...), almacenamiento thread-local (copias privadas de variables, thread-safe) u operaciones atómicas.

Las principales ventajas de los threads es que, al compartir el mismo espacio de memoria entre ellos, la comunicación interproceso es realmente rápida, no existe cambio de

contexto entre procesos y que son rápidos a la hora de inicializarse y terminarse.

Algunas aplicaciones basadas en uso de Threads son: *MySQL*<sup>12</sup>, *Firebird*<sup>13</sup> o *Apache*<sup>14</sup>.



**Figura 4.10:** Esquema Modelo basado en hilos

### Basado en actores

El modelo de actores, propuesto por primera vez con Carl Hewitt en 1973, se trata de un modelo de concurrencia computacional que, al igual que los hilos, tratan de solucionar el problema de la concurrencia.

En el modelo de actores, cada objeto es un actor. Ésta es una entidad que tiene una cola de mensajes o buzón y un comportamiento. Los mensajes pueden ser intercambiados entre los actores y se almacenan en el buzón. Al recibir un mensaje, el comportamiento del actor se ejecuta (ver Figura 4.11).

El actor puede:

- Enviar una serie de mensajes a otros actores.
- Crear una serie de actores.
- Asumir un nuevo comportamiento para el próximo mensaje

La importancia en este modelo es que todas las comunicaciones se llevan a cabo de forma asíncrona. Esto implica que el remitente no espera a que un mensaje sea recibido en el momento que lo envió, sino que directamente sigue su ejecución.

Una segunda característica importante es que todas las comunicaciones se producen por medio de mensajes: no hay un estado compartido entre los actores. Si un actor desea obtener información sobre el estado interno de otro actor, se tendrá que utilizar mensajes

<sup>12</sup><https://www.mysql.com/>

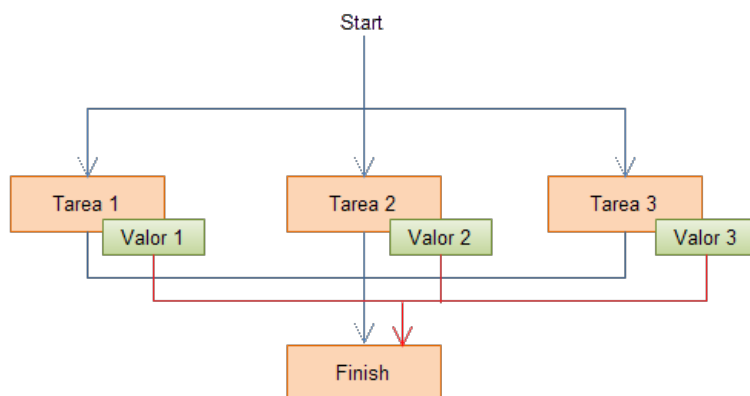
<sup>13</sup><http://www.firebirdsql.org/>

<sup>14</sup><http://httpd.apache.org/>



te, el uso de las retrollamadas concatenadas es una de las soluciones más utilizadas (ver Figura 4.12).

Los lenguajes y plataformas precursores de esta paradigma son: *NodeJS*<sup>18</sup> y la plataforma *Microsoft .NET*<sup>19</sup>, en especial, *C#*.



**Figura 4.12:** Esquema Modelo basado en E/S asíncrona

### Modelo reactivo (dirigido por eventos)

El patrón de diseño reactor es un patrón de programación concurrente para manejar los pedidos de servicio entregados de forma concurrente a un manejador de servicio desde una o más entradas. Este manejador demultiplexa los pedidos entrantes y los entrega de forma síncrona a los manejadores de pedidos asociados (ver Figura 4.13).

La estructura básica del patrón reactor lo conforman:

- **Recursos:** Cualquier medio capaz de proveer entrada o salida del sistema.
- **Demultiplexor síncrono de eventos:** Utiliza un bucle de eventos para bloquear a todos los recursos. Cuando es posible comenzar una operación síncrona en un recurso sin necesidad de bloqueo, el demultiplexor lo envía al dispatcher.
- **Dispatcher:** Maneja el registro y las bajas de los manejadores de pedidos. Entrega los recursos desde el multiplexador al handler de pedidos asociado.

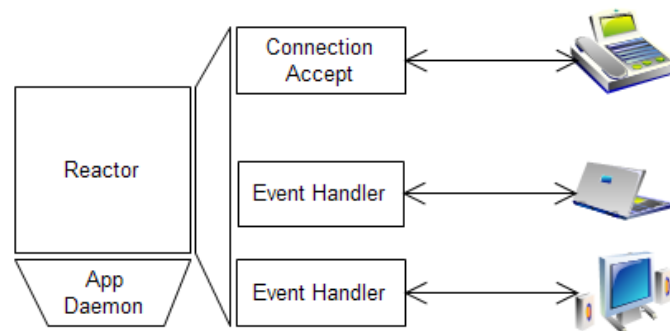
Los principales beneficios de este patrón es poder separar el código específico de la implementación del reactor, permitiendo que los componentes de la aplicación puedan ser divididos en módulos reutilizables. Además, la llamada síncrona de los handlers de pedidos permite concurrencia, de “grano grueso”, sin agregar la complejidad de múltiples hilos en el sistema.

<sup>18</sup><https://nodejs.org/>

<sup>19</sup><http://www.microsoft.com/net>

Paradigma	Lenguaje
Procesos	<i>C, D, Ruby</i>
Hilos	<i>C++, Go, Python</i>
Actores	<i>Akka (Scala, Java), GPars(Groovy)</i>
Asíncronos	<i>C#, F#, Julia, NodeJS, IOJS</i>
Reactivos	<i>Apache Mina, Python Twisted, RubyEM</i>

Este tipo de patrón suele ser implementado mediante librerías de terceros, como *Twisted (Python)*<sup>20</sup>, *EventMachine (Ruby)*<sup>21</sup> o *React PHP(PHP)*<sup>22</sup>, . . . , pero, no obstante, existen algunos lenguajes reactivos por definición, como *NodeJS*.



**Figura 4.13:** Esquema Modelo reactivo

Antes de describir las diferentes implementaciones, es importante recurrir a una tabla con un resumen de los distintos servidores junto al paradigma (ver Figura 4.5.2).

<sup>20</sup><https://twistedmatrix.com/>

<sup>21</sup><https://github.com/eventmachine/eventmachine/wiki>

<sup>22</sup><http://reactphp.org/>

## 4.6. Implementaciones concretas

En las siguientes líneas se detallarán cada uno de los servidores desarrolladores para este proyecto. Para facilitar su lectura, se distribuyen en diferentes subsecciones que incluyen una pequeña historia acerca del lenguaje/librería, implementación y una serie de pros y contras generales.

### 4.6.1. Servidor NodeJS

#### Historia

Desarrollado por Ryan Dahl (y otros desarrolladores dentro de la empresa Joyent) en 2009, NodeJS es un entorno de programación Open Source y multiplataforma en la capa del servidor (aunque no limitado únicamente a ello), basado en el lenguaje de programación ECMAScript, asíncrono, con I/O de datos basada en una arquitectura orientada a eventos. Su origen proviene del motor V8 de Google.

Aunque en su origen fue creado con el enfoque de facilitar la creación de programas en red altamente escalables, como servidores web, se puede utilizar para otras funcionalidades.

La inspiración para la creación de este lenguaje proviene de la barra de subida de ficheros de Flickr, la cual no demostraba claramente cuánta cantidad de fichero se había subido ya. Además, utiliza ciertas ideas de otros modelos, como es el caso de EventMachine (Ruby) o Twisted (Python), entre otros.

Pese a que NodeJS únicamente se basa en sistemas monothread, se puede activar el modo multithread a partir de la versión 0.10+. No obstante, NodeJS utiliza un sistema multithreads a la hora de controlar ficheros o de eventos.

#### Desarrollo

Debido al modelo de programación dirigido por prototipos y eventos por parte de JavaScript-NodeJS, y según la API de sockets de NodeJS, la arquitectura básica del servidor se basa en un socket que “escucha” las nuevas peticiones, creando internamente otros sockets hijos, a los cuales se les puede enviar mensajes.

Este socket padre se encarga principalmente de tres eventos básicos más el tratamiento de nuevas conexiones.

Los tres eventos básicos son:

- **data(data):** Se ejecuta cuando un socket recibe un dato por un socket cliente. En nuestro caso se ha utilizado para, dependiendo del tipo de dato recibido, enviar un tipo de mensaje al resto de clientes y, a veces, algún mensaje al emisor original.

- **error(exception):** Se ejecuta cuando se produce una excepción. Generalmente se ha ejecutado cuando se ha desconectado un socket sin el mensaje de cancelación definido.
- **exit:** Se ejecuta cuando un socket se desconecta. No se ha utilizado debido al sistema de desconexión por mensaje desarrollado.

La parte encargada de las nuevas conexiones se ejecuta en el mismo cuerpo del socket padre. Éste código inicialmente se encargaba de guardar todas las referencias a las nuevas conexiones en una colección, pero debido al sistema de identificación planteado, estas referencias se guardan al recibir el action *“initWName”*.

Para lanzar el servidor, utilizamos el host local (*localhost*) y el puerto 8089 y, para simplificar la codificación, se decidió utilizar *CoffeeScript*<sup>23</sup>, una pequeña librería de azúcar sintáctico con cierto parecido a la sintaxis de *Ruby*.

Debido a que el comportamiento interno de los objetos en JavaScript es JSON (*JavaScript Object Notation*), la conversión de los mensajes a JSON resultó muy sencillo, y mucho más el acceso a los atributos.

Como ya se explicó al inicio de esta parte, los mensajes están regidos por JSON serializado, el cual se deserializa en el servidor y, analizando el atributo ‘Action’, se ejecuta una tarea u otra.

Internamente, para separar algunas funciones grandes dentro del evento de tratamiento de datos, se incluyeron varias funciones como parte del socket padre, como los avisos a las nuevas conexiones al resto de clientes o el envío del mapa a los nuevos conectados.

Ya en las últimas fases de desarrollo, se incluyó el acceso a una base de datos MySQL, la cual fue difícil de tratar, debido a que las peticiones se realizaban asíncronamente, lo que requirió la instalación de otra librería para orquestaciones: “async”. Junto a esta, se añadió otra librería para simular locks con el objetivo de añadir un sistema de combates de duración aleatoria, la cual no se llegó a terminar de implementar.

La librería utilizada para acceder a *MySQL* ha sido: <https://www.npmjs.com/package/mysql>.

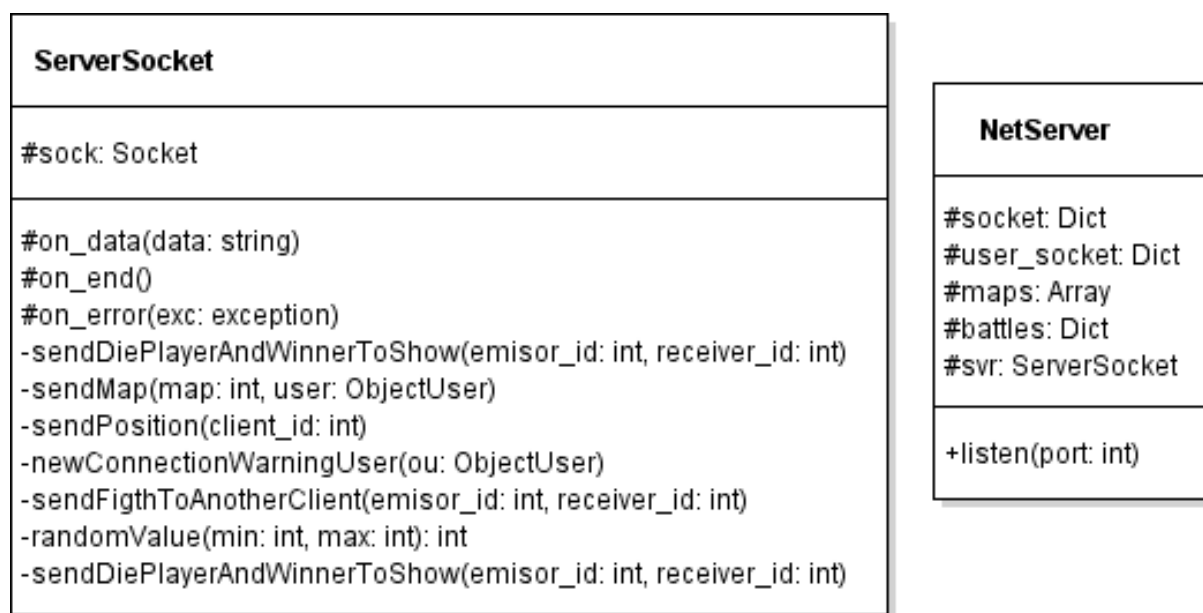
## Ventajas y desventajas

### Pros

- NodeJS es un lenguaje moderno, con una sintaxis simple y sencilla. Parsear datos en formato JSON, que es el tratamiento típico de los objetos JavaScript facilita mucho

---

<sup>23</sup><http://coffeescript.org/>



**Figura 4.14:** Esquema servidor NodeJS

la programación y aumenta claramente el rendimiento de la aplicación.

- Separar la lógica de nuevas conexiones, nuevos mensajes, excepciones y errores permite distinguir bien la funcionalidad de cada parte.
- El hecho de ejecutar “asíncronamente” las instrucciones, en muchos casos, como modificaciones en la base de datos, evitan la creación de threads o tener que dejar códigos síncronos como en otros lenguajes y plataformas. Además, el control de los errores de consultas a base de datos se puede mandar rápidamente al cliente al tener siempre acceso a su socket.
- Al tratarse de un lenguaje puntero, hay una gran comunidad y, por ello, gran facilidad para resolver dudas sobre el lenguaje.

### Contras

- Debido a la poca información encontrada acerca de la creación de servidores TCP, si un socket se conecta de forma incorrecta, no se ha encontrado una forma automática de detectar quién se ha desconectado, habiendo tenido que crear un protocolo de desconexión desde los clientes desarrollados.
- JavaScript no es un lenguaje fácil de entender y requiere de muchas horas de aprendizaje para controlar realmente las funcionalidades básicas de orientación a prototipos.
- Al utilizar código “asíncrono”, es necesario controlar el código de forma síncrona utilizando librerías de terceros (principalmente para facilitar el desarrollo).



### 4.6.2. Servidor IOJS

#### Historia

IO.JS nace con la privatización del proyecto NodeJS por parte de Joyent. Se trata de un proyecto Open Source basado en un fork realizado sobre el propio NodeJS, donde los propios usuarios contribuyen para mejorar la plataforma, de forma que se sigue un modelo abierto.

#### Desarrollo

El código y arquitectura es el mismo que en NodeJS (ver Figura 4.14). Únicamente se ha utilizado IO.JS para lanzar el servidor con el mismo código fuente (comando *iojs* en lugar de *node*).

#### Pros y contras

Los pros y contras son los mismos que en NodeJS, salvando las distancias entre las diferentes implementaciones de métodos, que pueden variar el rendimiento o una mayor cantidad de bugs con actualizaciones del motor V8.

### 4.6.3. Servidor Go

#### Historia

Go (más conocido como *Golang*), es un lenguaje inicialmente desarrollado en Google en 2007 por R. Griesemer, R. Pike y K. Thompson.

Se trata de un lenguaje con tipado estático y con una sintaxis muy parecida a *C*, pero que añade un recolector de basura, control de tipos o capacidad para tipos dinámicos. Además, permite utilizar librerías en *C*.

Go está centrado en el cálculo Pi y provee una potente API para desarrollo de programación concurrente y programación paralela.

#### Desarrollo

Tras buscar información sobre el desarrollo de aplicaciones bajo TCP en Go, se plantearon varias alternativas, las cuales se implementaron para probar cuál era la más rápida y eficiente:

- **Selector con canales:** El selector se encarga de encaminar la dirección del programa por cada uno de los canales implementados (nuevos usuarios, broadcast y

desconexiones). A pesar de ser una solución rápida, solo resultó efectiva para servicios con baja intensidad de mensajes por cliente, como un pequeño chat. Muy difícil de manejar para envío masivo de información.

- **Selector con canales y uso de gorutinas:** Partiendo de la misma idea anterior, el envío de mensajes se realizaba mediante gorutinas. Esa solución mejoraba en parte la rapidez de envío de mensajes entre los diferentes clientes para envío masivo, pero muchos paquetes llegaban de forma tardía.
- **Clientes con canal propio como buffer:** Solución final implementada para probar el rendimiento de Go para juegos online. Partiendo de una idea parecida a las dos primeras, cada cliente tendría un canal propio. De la versión anterior se mantiene el canal para gestionar las nuevas conexiones.

Siguiendo con la explicación de la solución final elegida, debido a la ineficiencia de los mensajes JSON en Go con algunos de los mandados por el servidor al cliente, se decidió utilizar la librería *ffjson*<sup>24</sup>. Esta librería no se compila junto con el mismo proyecto, sino que se trata de un generador de código para el manejo de JSON de las estructuras utilizadas en la aplicación.

Frente al segundo modelo, el servidor tiene tres funciones para sobrescribir el control de nuevas conexiones, nuevos mensajes y desconexiones mediante un uso de funciones anónimas y closures (ver Figura 4.15).

Ya en las últimas fases de desarrollo, se añadió un servicio adicional para trabajar con MySQL, además de que se optimizaron muchas partes de programa, como el uso de gorutinas para envío de mensajes mediante un uso abusivo de closures.

La librería utilizada para acceder a *MySQL* ha sido: <https://github.com/go-sql-driver/mysql>.

## Ventajas y desventajas

### Pros

- Go es un lenguaje moderno orientado a la programación concurrente, lo cual facilita en parte el desarrollo de aplicaciones.
- Una de las particularidades del modelo elegido para la arquitectura del servidor es facilitar un canal por cada cliente de forma que, de manera sencilla, podemos ir recogiendo los datos, procesarlos, y enviarlos a los respectivos clientes.

---

<sup>24</sup><https://github.com/pquerna/ffjson>

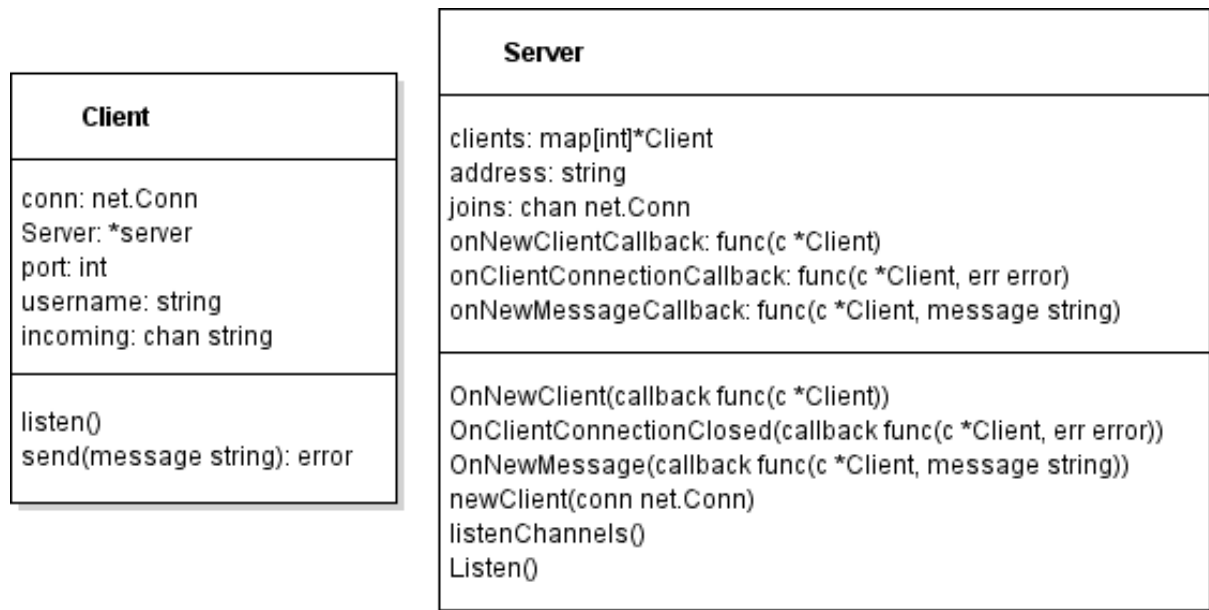


Figura 4.15: Esquema servidor Go

- A diferencia de otros lenguajes que también cuentan con recolector de basura, el manejo de la memoria es bastante eficiente. Además, obliga al desarrollador a utilizar un código compacto, sin librerías de más o variables sin utilizar.
- Es posible realizar pasos por referencia, cosa que otros lenguajes como Python o Java no disponen (o no de una manera tan sencilla) mediante el uso de punteros.
- Go permite retornar una cantidad variable de variables como salida de sus funciones.
- Es muy sencillo tanto crear, enviar variables, en una gorutina, siendo mucho más sencillo que C o C++. Además permite interactuar con librerías desarrolladas en C y genera ficheros binarios.
- Debido a su compilación y a que está diseñado para ocupar el menor espacio posible, el propio compilador no te dejará compilar el proyecto si existen variables no utilizadas o importaciones innecesarias, lo que consigue hacer un código elegante y eficiente.
- Posibilidad de cambiar el número máximo de cores que puede utilizar la aplicación. Por defecto, asigna el máximo.
- A diferencia de otros lenguajes de programación, Go permite programar en otras codificaciones diferentes de ASCII.
- En Go, las funciones son consideradas “funciones” de primera clase. Además, ‘posee

una potente API para la reflexión o posibilidad de usar funciones anónimas, y no existe dependencia circular entre las librerías del lenguaje.

- Gran cantidad de proyectos han sido migrados a Go, como ciertas partes de Dropbox o Google, incluso existen muchos proyectos desarrollados casi por completo en Go, como Docker o el Sistema Operativo Gofy <sup>25</sup>.

## Contras

- Al tratarse de una extensión de C, la orientación a objetos que incluye es bastante simple y, a veces, difícil de entender a primera vista, debido a que se basa en extensión de estructuras, ya que dispone de un diseño orientado a objetos dirigido por la composición.
- Debido a la poca comunidad actualmente (2015), algunos problemas que se han tenido no han tenido solución y se han tenido que optar por otras opciones, aunque hay multitud de vídeos y diapositivas de congresos de Google que facilitan el aprendizaje de ciertas funcionalidades.
- El tratamiento de JSON nativo es bastante lento en comparación con librerías de terceros, como *ffjson*.
- Es muy difícil ver a primera vista si una estructura define los métodos de una interfaz si no se dispone de un buen IDE, aunque tampoco hay ningún IDE realmente útil en el mercado.
- El manejo de mapas y slice es muy engorroso frente a otros lenguajes de programación como C++ o Python.
- Las colecciones de datos incluidas en las librerías son un tanto escasas, echando en falta estructuras como conjuntos, aunque se disponen de librerías de terceros para solucionar este problema.
- La existencia de tipos basados en 32 o 64 bits a veces dificulta el desarrollo y ralentiza el parte el sistema, debido al gran número de castings, lo que conlleva además, pérdida de información, como cierto número de decimales que, aunque en esta aplicación no es realmente importante, puede ser peligroso para sistemas embebidos o en tiempo real.

---

<sup>25</sup><http://gofy.cat-v.org/>

- A diferencia de muchos otros lenguajes, Go solo permite recorridos en forma de `for` (tanto como `foreach`, intervalos o simulaciones de bucles `while`). No hay posibilidad de sobrescribir funciones ni operadores.

#### 4.6.4. Servidor Python

##### Historia

Desarrollado por Guido Van Rossum en torno a 1991, Python es un lenguaje de propósito general, multiparadigma, Open Source y multiplataforma. Su filosofía de diseño es empatizar la legibilidad de código con una sintaxis simple que permite escribir en pocas líneas programas complejos.

Entre los paradigmas soportados, destacan la programación orientada a objetos, la programación imperativa, la programación funcional y la programación procedural. Además, incluye un tipado dinámico y un recolector de basura propio.

Uno de los principales problemas de este lenguaje es la existencia de dos versiones en paralelo que difieren, en parte, en la sintaxis de desarrollo.

Debido a la gran comunidad que hay por debajo, existen gran cantidad de librerías de terceros para muchos ámbitos, como por ejemplo, librerías de cálculo matemático, muy utilizadas en el ámbito científico al ser más fácil de utilizar y codificar que otros lenguajes. No obstante, como causa de la gran cantidad de librerías y a la deficiencia del gestor de paquetes, se ha desarrollado un gran entorno de desarrollo, Anaconda, que permite seleccionar las librerías necesarias, evitando problemas de dependencias entre paquetes.

La versión original de Python fue desarrollada bajo un compilador en *C* (*Cython*), aunque existen otras alternativas como *Jython* (*Java*), *IronPython* (*.NET*), ...

##### Desarrollo

Python es un lenguaje que ha mejorado enormemente en la programación con threads desde sus inicios, facilitando una potente arma de trabajo que, unido a la sencillez de programación, aumenta enormemente el rendimiento de nuestras aplicaciones.

La idea principal es crear un thread al cual, como argumento *“target”*, se le facilita la función a ejecutar. Todo el funcionamiento interno del servidor funciona a base de procesos y forks internos.

Gracias a la facilidad para el manejo de JSON, la separación de la lógica de las distintas acciones resulta verdaderamente sencilla. Junto a esto, la librería *SocketServer*<sup>26</sup> de Python permite evitar tener que controlar las nuevas conexiones. (ver Figura 4.16)

---

<sup>26</sup><https://docs.python.org/2/library/socketserver.html>

Aunque parece que en las últimas versiones de Python se permite que las variables compartidas tengan condición de carrera, se ha preferido utilizar algunos locks para controlar los nuevos sockets y posiciones de los usuarios.

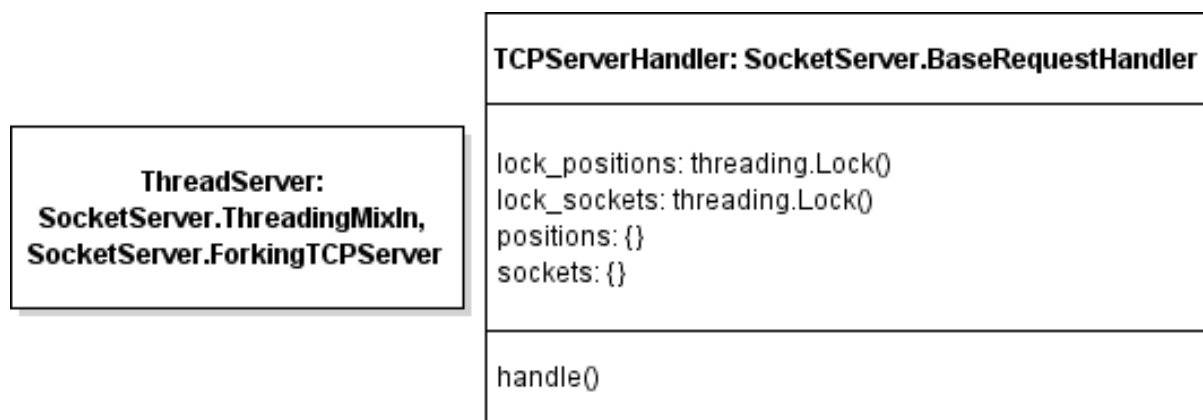


Figura 4.16: Esquema servidor Python

## Ventajas y desventajas

### Pros

- Entre las principales ventajas de Python se encuentran la limpieza de código, la gran cantidad de bibliotecas disponibles, ser multiplataforma y una gran comunidad de desarrollo.
- Se trata de un lenguaje portable, por lo que es posible ejecutar el código en cualquier plataforma que soporte Python, especialmente las distribuciones Linux, donde este lenguaje se incluye en la instalación del Sistema Operativo.
- Python incluye una terminal para realizar pruebas sin necesidad de tener un entorno de desarrollo, o escribir en un fichero, lo cual reduce determinadas operaciones de pruebas.
- Tiene una sintaxis muy parecida a pseudocódigo, lo cual facilita su aprendizaje y, al ejecutarse en tiempo real, no es necesario esperar a compilar el proyecto.
- *SocketServer* permite separar de forma sencilla una aplicación TCP.

### Contras

- Es bastante más complejo de depurar que otros lenguajes de programación compilado o con código intermedio, debido a que existen pocos IDE's para este lenguaje.

- Al tratarse de un lenguaje dinámico, algunos errores solo pueden ser detectados en tiempo de ejecución, lo que en cierta parte, en contraposición a la rapidez de programación, ralentiza la etapa de pruebas.
- Toda su sintaxis y bloques de instrucciones se basan en la tabulación, lo que a veces, si se tiene un mal diseño, el código se vuelve difícil de leer y de interpretar por un programador.
- Debido a que existen dos versiones del lenguaje en paralelo, 2.7 y 3.4, muchos códigos y librerías no funcionan y necesitan modificar parte del código.
- Cuanto más grande es el proyecto, más difícil es gestionar todo el código debido, principalmente, a que no hay un acceso rápido en desarrollo sobre qué tipo de variable se está tocando y, mucho menos, saber qué funciones y atributos contiene.
- Python es mucho más lento que *Java*.

#### 4.6.5. Servidor Python Twisted

##### Historia

Twisted es un framework de red desarrollado en Python, basado en el paradigma de la programación dirigida por eventos.

Tiene soporte para varias arquitecturas, como TCP, UDP o Unix domain sockets, un gran número de protocolos (incluidos HTTP o FTP), entre otros.

La idea principal de este framework es separar los distintos eventos en pequeños callbacks predefinidos, de forma que es más fácil separar las tareas complejas.

Entre las aplicaciones más conocidas que utilizan esta herramienta, destacan *TwitchTV*<sup>27</sup>, *Omegle*<sup>28</sup> (en sus primeras versiones), o el sistema de alojamiento de archivos de *Ubuntu One*<sup>29</sup>.

##### Desarrollo

La librería Python Twisted consigue separar de una forma bastante lógica las nuevas conexiones, conexiones perdidas y procesos de broadcast. Para ello, se basa en el uso de factorías, las cuales almacenan las variables globales al sistema. En este caso se ha optado por un modelo de datos muy parecido al desarrollado en Python con threads del punto anterior.

---

<sup>27</sup>[www.twitch.tv/](http://www.twitch.tv/)

<sup>28</sup>[www.omegle.com/](http://www.omegle.com/)

<sup>29</sup><https://one.ubuntu.com/>

A diferencia del modelo basado en threads, podemos crear variables dinámicamente dentro de la factoría, lo cual facilita bastante la gestión de identificadores de sockets. (ver Figura 4.17)

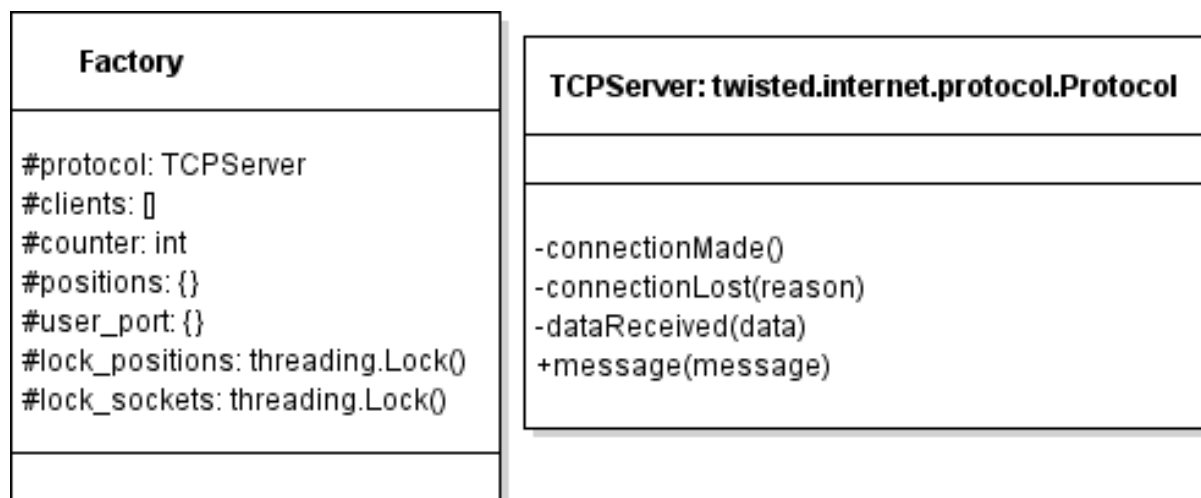


Figura 4.17: Esquema servidor Python Twisted

## Ventajas y desventajas

### Pros

- A diferencia de la programación con threads, Twisted permite separar los distintos eventos de nuevas conexiones o mensajes de forma más rápida, separando la lógica de cada parte.
- Replicar gran parte de la funcionalidad del servidor simple en Python es bastante sencillo si se ha mantenido una división correcta de las tareas en el paso anterior.

### Contras

- Como se comentó en las contras de Python, existe un problema por existir dos versiones en paralelo. Cabe destacar que Python Twisted solo funciona para la versión 2.7.

## 4.6.6. Servidor Ruby

### Historia

Ruby es un lenguaje de programación interpretado, reflexivo y orientado a objetos, creado por el programador japonés Yukihiro “Matz” Matsumoto, quien comenzó a trabajar en Ruby en 1993, y lo presentó públicamente en 1995.



Combina una sintaxis inspirada en Python y Perl, con características de programación orientada a objetos similares a Smalltalk. Además, comparte diversas funcionalidades con otros lenguajes de programación, como Lisp o Lua.

Su nombre viene de una broma que los compañeros de Manz aludiendo al lenguaje Perl. El objetivo principal de Ruby es mejorar la productividad y la diversión del desarrollador, siguiendo para ello varios principios de una buena interfaz de usuario. Además, sostiene que “el diseño de sistemas necesita enfatizar más las necesidades humanas que las de las máquinas.” (cita recogida en [Pre]).

## Desarrollo

A la hora de desarrollar el servidor en Ruby con las librerías nativas, se utilizó la información recogida en el artículo *Sockets programming in Ruby de IBM* [IBM]. La idea principal de este servidor en Ruby nativo fue un selector que se encargara de la siguiente acción o evento a tratar.

Pese a varios intentos en la implementación del servidor siguiendo este manual y algunos otros, el principal cuello de botella se encontraba en la recogida de información (stdin) de los sockets, debido a que no soporta gran cantidad de mensajes por segundo. Junto a esto, se encontraron diversos “tuits”<sup>30</sup> y post acerca sobre la escasa y mala información sobre los sockets en Ruby. (ver Figura 4.18)

Tras una búsqueda exhaustiva por encontrar más información por la red, otros usuarios se encontraban con el mismo problema y algunos planteaban una librería alternativa al uso nativo de sockets de Ruby: *EventMachine*.

Por motivos históricos, se dejó contemplada esta implementación dentro del trabajo debido a todas las horas utilizadas para optimizar los diversos errores encontrados son las entradas de los sockets.

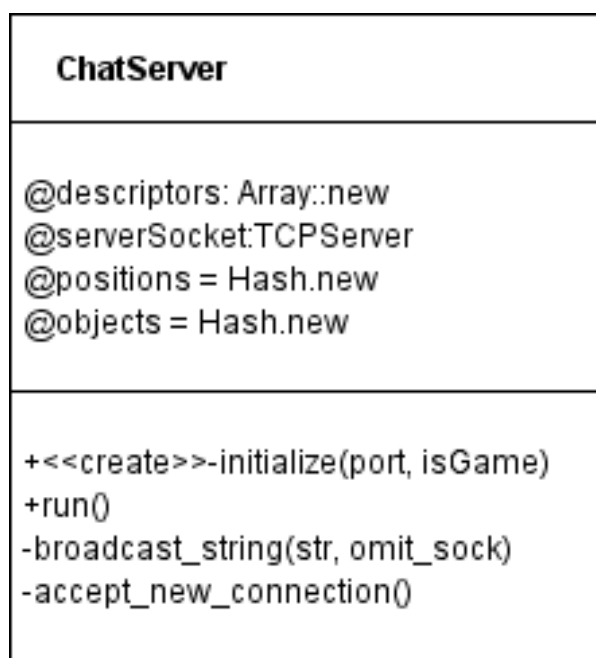
## Ventajas y desventajas

### Pros

- Se trata de un lenguaje con una sintaxis muy limpia y legible, y con gran información en la red, lo que conlleva a que sea muy fácil desarrollar códigos en Ruby. Podemos crear nuevos atributos dentro de una clase en tiempo de ejecución (aunque esto no siempre es una ventaja).
- A diferencia de *Python*, para Ruby existen varios plugins para IDE’s, destacando el papel de *Netbeans*, el cual facilita las tareas de depuración.

---

<sup>30</sup><https://twitter.com/maccaw/status/2108118075>



**Figura 4.18:** Esquema servidor Ruby

- Al igual que *Python*, Ruby no requiere compilación y dispone de una terminal para realizar pruebas.
- Gracias al framework *Ruby on Rails*<sup>31</sup>, el uso de Ruby ha aumentado bastante y existe por ello una gran comunidad en la web.
- Debido a su sintaxis, los códigos desarrollados en Ruby intentan imitar al lenguaje humano y permite una orientación a objetos mucho mejor que *Java* o *Python*.

### Contras

- A pesar de la existencia de muchísima información en la red, casi toda está centrada en el desarrollo de Ruby con *Ruby on Rails*, *RPG MAKER XP*<sup>32</sup>, o desarrollo de escritorio, pero hay muy poca información sobre el desarrollo multithread. y mucho menos para el desarrollo de servidores TCP.
- Al tratarse de un lenguaje dinámico, algunos errores solo pueden ser detectados en tiempo de ejecución, lo que en cierta parte, en contraposición a la rapidez de programación, ralentiza la etapa de pruebas.
- A pesar de su sencillez y su rapidez de desarrollo, Ruby es mucho más lento que *Python* y *Java*.

<sup>31</sup><http://rubyonrails.org/>

<sup>32</sup><http://www.rpgmakerweb.com/products/programs/rpg-maker-xp>

### 4.6.7. Servidor Ruby EventMachine

#### Historia

Desarrollado por Francis Cianfrocca en 2006, EventMachine es un software de alto nivel de escalabilidad en *Ruby*. Está basado en un modelo dirigido a eventos usando el patrón Reactor.

EventMachine está diseñado siguiendo dos ideas clave:

- Un alto nivel de escalabilidad, performance y estabilidad para entornos de producción con mucha demanda.
- Una API que elimina la complejidad del control de la programación con threads, liberando a los desarrolladores a solo concentrarse en la parte lógica de la aplicación.

#### Desarrollo

Como ya se ha comentado en la historia de la librería, EventMachine facilita la programación de servidores sobrescribiendo diferentes métodos. Esto, junto a la ya implementación en NodeJS del servidor, facilitó un desarrollo muy ágil y rápido, ya que los mismos eventos que se trataban en *NodeJS*, existían, aunque con otro nombre, en EventMachine. Las principales funciones del servidor EventMachine son los siguientes:

- **post\_init**: Utilizado para nuevas conexiones. Aunque se utilizó en las primeras fases de desarrollo, se descartó su uso al añadir los sockets con *“initWName”*.
- **receive\_data (msg)**: Encargado de recibir nuevos mensajes y de su tratamiento. Como en el resto de servidores, recibe el dato, lo intenta parsear a JSON y, según el tipo de *“Action”*, se envían unos determinados mensajes a emisor y resto de clientes conectados.
- **unbind**: Encargado de las desconexiones de mensajes. Al contrario que en *NodeJS*, si tenemos un acceso más sencillo al socket cliente desconectado. Esta función solo se utiliza como salida auxiliar para la cancelación en caso de un error en el mensaje de cancelación.

A diferencia de *NodeJS*, el servidor se ha desarrollado utilizando un módulo Ruby (module Ruby) (ver Figura 4.19).

En las últimas fases de desarrollo, se incluyó el acceso a *MySQL* con la librería *mysql2*<sup>33</sup>, aunque no se implementó toda su funcionalidad al completo. A diferencia de *NodeJS*, las

---

<sup>33</sup><https://github.com/brianmario/mysql2>

consultas se realizan de forma síncrona, la cual facilita, en algunas partes, la orquestación.

**Nota:** Como medida de optimización debido al uso de una base de datos, gran parte de las consultas son invocadas mediante uso de threads, para evitar bloqueos y tiempos de espera entre los clientes y el servidor. No obstante, algunas consultas a la base de datos se han programado de forma bloqueante, como la obtención de los datos de partida al iniciar el servidor, o la recarga de antiguos usuarios en la base de datos.

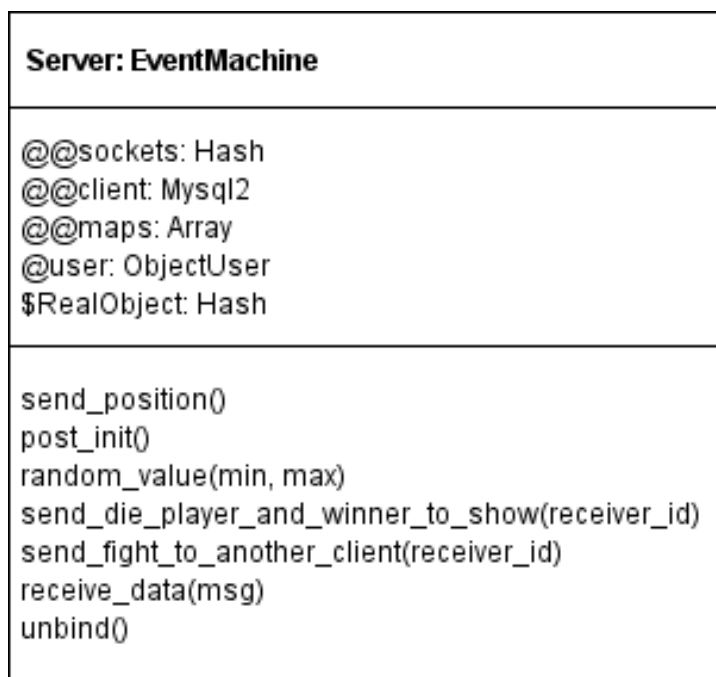


Figura 4.19: Esquema servidor Ruby EventMachine

## Ventajas y desventajas

### Pros

- EventMachine tiene separada las distintas funcionalidades más utilizadas para desarrollar servidores (nueva conexión, envío de datos y desconexión), de forma que es muy fácil gestionar el código.

### Contras

- Como cualquier lenguaje dinámico, el hecho de tener que interpretar el código ralentiza en parte los tiempos de carga.
- El poder crear nuevas variables (tanto de clase o de módulo) como estáticas en tiempo de ejecución puede conllevar muchos errores para desarrollar, debido a que no tenemos centralizadas las variables en un sitio común.

### 4.6.8. Servidor C

#### Historia

C es un lenguaje de programación compilado creado en 1972 por Dennis M. Ritchie en los Laboratorios Bell, como evolución del anterior lenguaje B. Se trata de un lenguaje orientado a la implementación de Sistemas Operativos, y es muy apreciado por la eficiencia del código producido, además de ser el más popular para el desarrollo de software de sistemas y, en menor parte, de aplicaciones.

C nació como un deseo de Ken Thompson y Brian Kernighan, junto a Dennis Ritchie, de poder crear un Sistema Operativo en un lenguaje de alto nivel, independiente del hardware donde se ejecutase. Este desacople con la tecnología facilitó la portabilidad, la reutilización de código y la reducción de tiempos de desarrollo.

Con el paso del tiempo se han desarrollado cientos de bibliotecas que permiten que otros desarrolladores puedan utilizar códigos de terceros para desarrollar sus propias aplicaciones.

La última versión del estándar data del 2007, pero la versión más utilizada es el ANSI-C.

#### Desarrollo

La implementación elegida para el desarrollo en C bajo Windows hace un uso de la librería nativa de este Sistema Operativo: *Winsock*<sup>34</sup>.

Esta interfaz se trata de una biblioteca dinámica de funciones DLL creada para la implementación de TCP/IP. Incluye soporte para envío y recepción de paquetes de datos a través de sockets BSD.

La implementación de gran parte del servidor no se completó debido a diversos problemas relacionados con estructuras de datos y a la escasa funcionalidad y documentación ofrecida y encontrada, respectivamente, respecto a *Winsock*. No obstante, se incluye en el disco disponible con este trabajo el código desarrollado hasta el momento.

#### Ventajas y desventajas

##### Pros

- C es uno de los primeros lenguajes de programación, y al ser desarrollado como una alternativa directa desde lenguaje ensamblador, el código producido es muy eficiente y rápido. Desarrollar utilizando estructuras es muy rápido, debido a que no es necesario crear constructores, ni métodos de acceso y edición de atributos.

---

<sup>34</sup>Biblioteca dinámica de funciones DLL para Windows que se hizo con la finalidad de implementar TCP/IP.

- A diferencia de otros lenguajes utilizados en este trabajo, podemos separar los métodos de los módulos de las implementaciones.
- En C podemos controlar la memoria, la cual facilita que el código sea realmente pequeño en compilación, y extensible en tiempo de ejecución. No obstante, requiere que el mismo programador se encargue de liberar los datos cuando no se necesiten.

### Contras

Antes de mostrar las desventajas, hay que destacar que algunas desventajas principalmente provienen de Windows, no del lenguaje en sí.

- Las librerías de sockets de Windows tienen muy poca funcionalidad, por lo que a veces se dificulta gran parte del desarrollo, además de que al ser librerías propias del Sistema Operativo, tiene una portabilidad casi nula.
- C no tiene orientación a objetos pura (aunque se puede simular), lo que implica tener que replicar muchas funcionalidades para cada tipo de estructura permitida por la aplicación.
- Las estructuras no pueden privatizar el acceso a sus atributos, por lo que la “única” forma de privatizar los accesos es compilar dichos ficheros, y proveer al desarrollador únicamente las librerías y los ficheros de cabecera.
- Al no existir genéricos o plantillas, es necesario utilizar librerías de terceros para utilizar estructuras de datos diferentes a arrays, structs o punteros.
- C no provee en sus librerías control para leer y crear JSON de forma nativa.

### 4.6.9. Servidor C++

#### Historia

Desarrollado por Bjarne Stroustrup en 1979, C++ nace como parte de su tesis doctoral. Se trata de un lenguaje de programación imperativo, orientado a objetos, con posibilidad de acceso a instrucciones a bajo nivel y manejo de punteros (debido a ser una extensión de C).

Diseñado principalmente para la programación de sistemas (sistemas embebidos, kernels, etc), provee de un gran rendimiento, eficiencia y flexibilidad, por lo que también se utiliza para el desarrollo de aplicaciones de escritorio, servidores, sistemas críticos o compiladores. Debido a que las primeras versiones no incluyen diversas funcionalidades mostradas en otros lenguajes posteriores, como *Java* o *C#*, con el paso de los años se han desarrollado

otras versiones del lenguaje, como la versión 11, que incluye manejo de funciones lambda o recolección de punteros.

## Desarrollo

Cuando se empezó a plantear el desarrollo del servidor en C++, se empezó a cuestionar cuál era la mejor opción, ya que los sockets nativos, como en el caso de C, no eran multiplataforma. Por tanto, se decidió el uso de la librería Network de las mismas librerías de SFML utilizadas en los clientes.

En este caso, creamos un escuchador TCP, el cual se encarga de aceptar las nuevas conexiones. Una vez añadidos los nuevos clientes en un mapa, recorremos cada uno de ellos y, según el estado de los sockets, se realiza una acción u otra. En cuanto a los eventos de broadcast, se decidió utilizar threads con un uso de funciones anónimas (ver Figura 4.20). En una última instancia se intentó añadir el acceso a MySQL, pero debido a varias librerías pesadas que dependían de ésta, se decidió no utilizar.

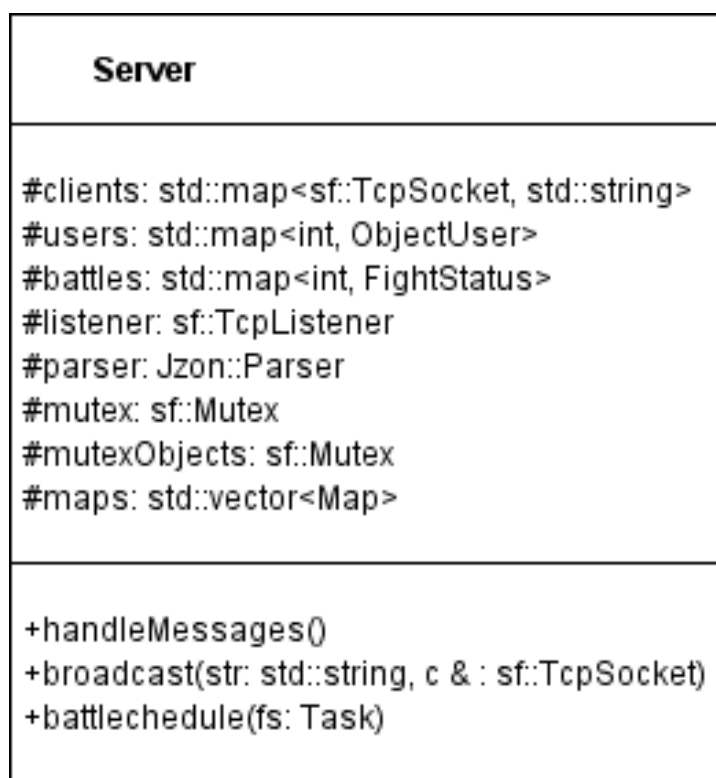
Cabe destacar que debido a que C++ no es un lenguaje centrado en el uso de la concurrencia y las variables compartidas (aunque actualmente se han integrado el uso de funciones lambda), se han utilizado algunos mutex para evitar problemas de condición de carrera.

**Nota 1:** Debido a que gran parte del cliente en C++ ya estaba desarrollado, parte del código del funcionamiento de envíos de mensajes por TCP y del tratamiento de JSON, se replicaron con facilidad.

## Ventajas y desventajas

### Pros

- Gracias a las librerías de red de la librería SFML, se ha podido facilitar la programación con sockets TCP.
- C++ es un lenguaje bastante óptimo, y dispone de muchas librerías nativas para el control de colecciones de datos.
- C++ desde sus últimas versiones dispone de funciones anónimas y de un control de memoria por parte del propio compilador, aunque el programador puede controlar cuándo se crea y cuando se borra la memoria.
- Gracias al uso de punteros, la modificación de objetos a punteros dentro de una colección es bastante rápido, debido a que tenemos acceso directo a la referencia a los objetos.



**Figura 4.20:** Esquema servidor C++

- Al ser una extensión de *C* (C con Clase), disponemos de clases y estructuras para organizar los nuevos tipos definidos por el programador, además de plantillas (genéricos) para estructuras de datos y clases, lo que evita la replicación de código de *C*.
- Gran parte de las librerías y códigos son multiplataforma, a diferencia de *C*, lo que mejora enormemente la portabilidad.
- Aunque la sintaxis de C++ es casi idéntica a *C*, es mucho menos verbosa, lo que facilita la escritura y lectura del código.
- C++ permite sobrescribir gran parte de los operadores básicos para las nuevas clases creadas por el usuario.

### Contras

- Aunque el desarrollo en C++ puede llegar a ser muy eficiente, encontrar librerías 100 % compatibles entre los distintos Sistemas Operativos del mercado.
- La liberación de memoria a veces resulta bastante tediosa.
- Los compiladores de C++ son un poco más lentos que los de *C*, pero dan un buen soporte y eficiencia frente a otros lenguajes.



- C++ no provee librerías nativas para trabajar con JSON.
- Los buffers no pueden ser dinámicos, por lo que, generalmente, se tiene que reservar mucha más memoria de la que se va a utilizar.
- C++ es un lenguaje destinado a aplicaciones con alto cómputo, pero no es tan potente para el tratamiento de cadenas.

#### 4.6.10. Servidor D

##### Historia

D (o D-lang) es un lenguaje de programación orientado a objetos, imperativo y multi-paradigma diseñado por Walter Bright en 2001. Entre los distintos paradigmas soportados, cabe destacar la metaprogramación, la programación funcional, la programación paralela y la concurrente, junto al a orientada a objetos e imperativa anteriormente mencionados. Diseñado como un rediseño de C++ con un enfoque más pragmático, D ha incorporado a sus librerías una gestión de memoria a través de un recolector de basura, programación funcional, control de concurrencia incorporada y multiplataforma (en contraposición a *C* y *C++* que dependen del Sistema Operativo). También incluye sobrecarga de operadores como C++ y la incorporación de código ensamblador, además de ciertas ideas de lenguajes interpretados a código intermedio como *y* y *C#*, pero con posibilidad de gestión propia de memoria.

No obstante, este lenguaje tiene varios detractores, debido a que no hay mucho soporte en la industria y la biblioteca estándar, *Phobos*<sup>35</sup>, no era muy completa hasta la versión 2.0 en la que se permite utilizar *Tango*<http://www.dsource.org/projects/tango>, otra biblioteca.

##### Desarrollo

Siguiendo un modelo muy parecido al de *C*, el servidor desarrollado en D, por motivos de un desarrollo bastante rápido, se decidió reservar toda la memoria a utilizar al iniciar el servidor para añadir un número determinado de usuarios (esto en la realidad no debería ser así, pero era la solución más rápida para implementar) (ver Figura 4.21).

La API de sockets de Dlang recomienda el uso de un selector<sup>36</sup>, el cuál recibe un conjunto con todos los sockets conectados.

A la hora de leer los cambios de estado de los sockets realizamos un recorrido uno por uno, controlando si se han recibido nuevos datos, de forma que podamos lanzar las distintas

---

<sup>35</sup><http://dlang.org/phobos/>

<sup>36</sup>[http://dlang.org/phobos/std\\_socket.html](http://dlang.org/phobos/std_socket.html)

acciones implementadas. A diferencia de otros servidores implementados, una vez completado el tamaño fijado para la lista de sockets, la aplicación avisa al usuario y desconecta al cliente.

Cabe destacar dos aspectos:

- Aunque el uso del *GOTO*<sup>37</sup> está casi totalmente prohibido en lenguajes de programación de alto nivel, se ha requerido de uso para no finalizar el bucle de control del estado de los sockets.
- Existe una librería llamada *vibe.d* (<http://vibed.org>), la cual permite crear, entre otros, servidores TCP y HTTP, muy parecido a *Apache Mina*<sup>38</sup>, *Python Twisted* o *Ruby EventMachine*. No se ha utilizado debido a que se encontró en fases tardías de desarrollo y no se dispuso de más tiempo.

Para finalizar, es importante destacar que la principal fuente de información sobre TCP en D proviene de un ejemplo de su propio repositorio de Github: <https://github.com/D-Programming-Language/dmd/blob/master/samples/listener.d>

Server
<pre>#listener: TcpSocket #reads: Socket[] #socketSet: SocketSet #positions: ObjectUser[int]</pre>
<pre>static sendPosition(ref Socket sock, int port) static sendFightToAnotherClient (ref Socket sock_emisor,     int id_emisor, ref Socket sock_receiver, int id_receiver)</pre>

**Figura 4.21:** Esquema servidor D

## Ventajas y desventajas

### Pros

- D es un lenguaje moderno nacido como una extensión de *C*, con ideas de *C++*, *Java* y *C#*, por lo que incluye manejo de estructuras y clases y, como *C++*, permite acceso a lenguaje ensamblador.

<sup>37</sup>El propósito de la instrucción GOTO es transferir el control a un punto determinado del código, donde debe continuar la ejecución. El punto al que se salta, viene indicado por una etiqueta. Se trata de una instrucción de salto incondicional.

<sup>38</sup><https://mina.apache.org/>

- D permite muchos tipos de paradigmas, por lo que podemos utilizar programación concurrente, programación paralela o programación funcional generando un ejecutable compilado.
- D dispone de un recolector de basura muy eficiente, por lo que el programador no debe controlar estos borrados. No obstante, el propio programador puede liberar la memoria sin que el recolector falle.

### Contras

- Hay muy poca información en internet sobre este lenguaje, siendo el único manual de referencia oficial *The D Programming Language* de Andrei Alexandrescu[Ale].
- La conversión de mensajes a estructuras JSON y viceversa de las librerías nativas es un poco difícil de manejar, por lo que se ha tenido que utilizar código de terceros para facilitar la tarea.

## 4.6.11. Servidor Java Apache MINA

### Historia

Apache MINA (Multipurpose Infrastructure for Network Applications) es un proyecto de código libre desarrollado en Java para el desarrollo de aplicaciones en red. MINA puede ser utilizado para facilitar la escalabilidad y una alta performance.

### Desarrollo

El modelo de desarrollo de servidores planteado por Apache Mina, en especial, para servidores TCP, es un sistema altamente escalable con un uso intensivo de eventos. Para ello, Apache Mina dispone de diversos métodos que pueden ser redefinidos (en nuestro caso, se ha sobrescrito los métodos de captura de excepciones, broadcast, y la creación y destrucción de sesiones).

Encapsulando los sockets o clientes bajo una clase *IoSession*<sup>39</sup>, podemos añadir dinámicamente atributos, como si de una sesión de un servidor HTTP se tratase (ver Figura 4.22).

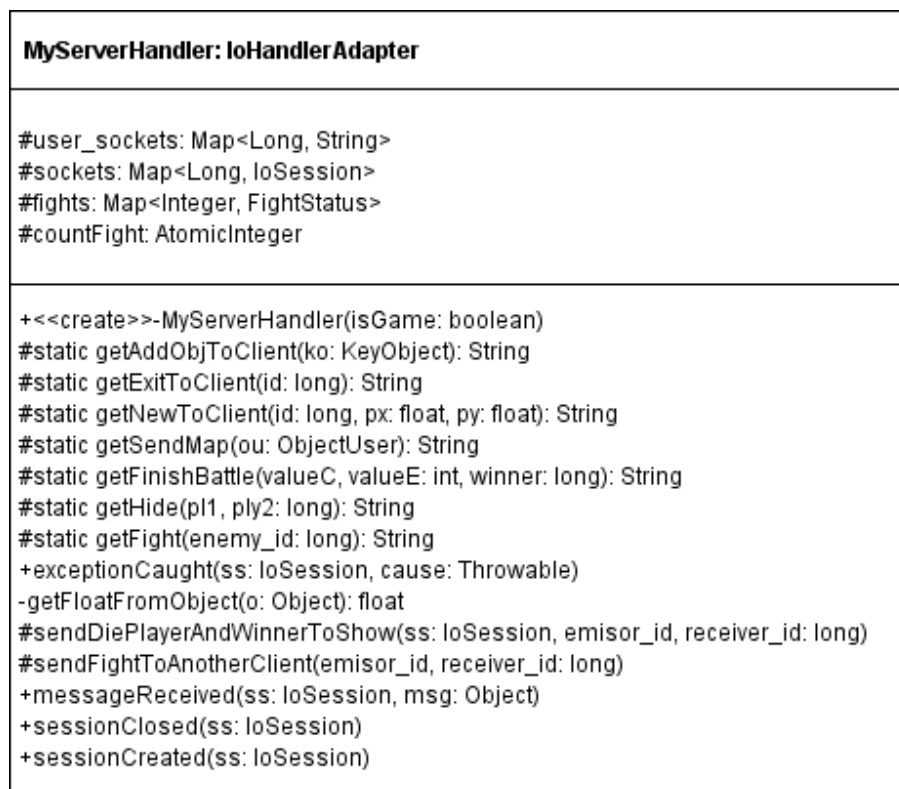
Para simplificar la creación de objetos JSON, se decidió crear un conjunto de objetos comunes a los cuales se les añadirían los campos necesarios, al igual que se hizo en los objetos JSON en el cliente *C++*.

En las últimas fases se incluyó acceso a *MySQL* y un uso de threads (con la API de *Java 8*) para evitar diversas pausas en la ejecución al recorrer todos los clientes.

---

<sup>39</sup><https://mina.apache.org/mina-project/apidocs/org/apache/mina/core/session/IoSession.html>

Además, se añadió una prueba de los *schedule*<sup>40</sup> para realizar combates aleatorios, el cual no se pudo llegar a probar, ya que no era una de los objetivos del trabajo.



**Figura 4.22:** Esquema servidor Java Apache Mina

## Ventajas y desventajas

### Pros

- Apache Mina permite separar distintas partes de la lógica del programa.
- Existen muchos editores de código libre para Java, muchos de ellos desarrollados bajo *Java*, como *Netbeans* o *Eclipse*, los cuales facilitan las tareas de desarrollo y las tareas de depuración (punto por el que destacan).
- Desarrollar aplicaciones basadas en *Java* es bastante rápido debido a la cantidad de herramientas de autocompletado existentes.
- Los compiladores de *Java* generar una aplicación *jar* basada en un código intermedio llamado *Bytecode*, lo cual acelera el programa a ejecutar, a diferencia de otros lenguajes interpretados como *Python* o *Ruby*.

<sup>40</sup>Tareas programadas

- Utilizar herramientas como *Apache Maven*<sup>41</sup> para gestionar las librerías facilitan enormemente el desarrollo en equipos de desarrollo de más de una persona, porque podemos evitar la dependencia de librerías que cada usuario podría guardar en diferentes directorios.

### Contras

- A pesar de tener muchas funciones para sobrescribir, muchas de ellas no son muy necesarias, como el hecho de poder controlar que recibes un evento y que estás enviando un mensaje, cuando estas dos funciones podrían encontrarse juntas. No obstante, la librería no requiere redefinir todas las funciones.
- Aunque los ejecutables de *Java* son muy pequeños, en tiempo de ejecución, *Java* utiliza gran cantidad de memoria.
- Las excepciones son el punto más débil de *Java*, debido a que recorre toda la pila de llamadas antes de finalizar la ejecución del programa.
- *Java* no dispone de librerías para tratamiento de JSON de forma nativa, por lo que se requiere el uso de librerías de terceros.

#### 4.6.12. Servidor C#

##### Historia

C# es un lenguaje de programación orientada a objetos creado por Anders Hejlsberg en torno al 2001, desarrollado y estandarizado por Microsoft como parte de su plataforma .NET.

Tiene una sintaxis derivada de *C/C++* y, aunque forme parte de la plataforma .NET, que es una interfaz de programación de aplicaciones, C# es un lenguaje independiente que originariamente se creó para producir programas sobre esta plataforma .NET.

C# incorpora todas las ventajas de *Java*, pero además, algunas ventajas de *C++*, como la sobrecarga de operadores.

##### Desarrollo

C# propone un cambio en el modelo de desarrollo utilizando métodos asíncronos, esto es, los métodos no se ejecutan en serie, sino en “paralelo”. De esta manera, dos o más tareas pueden ejecutarse a la vez bajo un uso intensivo de retrollamadas (lo que se ejecuta al finalizar) y de funciones anónimas (generalmente, la función ejecutada dentro del

---

<sup>41</sup><https://maven.apache.org/>

método asíncrono) (ver Figura 4.23).

Como casi todos los servidores implementados, se crea un socket principal de escucha de nuevas conexiones (usando en este caso una llamada asíncrona sin retrollamada). Una vez aceptada la conexión, se ejecuta un bloque asíncrono para el control de mensajes del socket.

Debido al sistema de programación de C#, no es necesario realizar retrollamadas para volver a ejecutar un bloque (a diferencia de otros lenguajes, como Erlang).

Para finalizar un bloque, utilizamos las correspondientes llamadas de la Figura 6.3.

**Nota de interés:** Es importante destacar que se ha tenido que cambiar el carácter para determinar si un número es un decimal o no (esta documentación sobre cómo se hace se incluye en el fichero *Program.cs*).

Program
<pre>-static Socket _serverSocket -static readonly List&lt;Socket&gt; _clientSockets -static Dictionary&lt;string, ObjectUser&gt; _positions -static readonly byte[] _buffer</pre>
<pre>+static void Main() -static SetupServer() -static CloseAllSockets() -static AcceptCallback(IAsyncResult AR) -static ReceiveCallback(IAsyncResult AR)</pre>

**Figura 4.23:** Esquema servidor C#

## Ventajas y desventajas

### Pros

- Una de las funcionalidades de C# por las que lo prefiero frente a Java son los delegates o funciones anónimas.
- El entorno de programación *.NET*, *Visual Studio*<sup>42</sup>, es un IDE muy completo, que facilita el tiempo de desarrollo, de forma que aumenta la eficacia. Dispone de un

<sup>42</sup><https://www.visualstudio.com/>

sistema de autocompletado, generación de clases, entre otros, como extensiones para ejecutar otros lenguajes de programación.

- A diferencia de *Java*, *C#* dispone de paso de primitivas por referencia, nuevos tipos de visibilidad de variables o delegates.
- Microsoft dispone de muchos manuales y ayudas para muchas dudas del lenguaje dentro de la página [https://msdn.microsoft.com/es-es/library/zkxk2fwf\(v=vs.90\).aspx](https://msdn.microsoft.com/es-es/library/zkxk2fwf(v=vs.90).aspx).
- La plataforma de videojuegos *Unity*<sup>43</sup> tiene a *C#* como uno de los lenguajes de scripting, por lo que hay una gran comunidad en la que apoyarse.

### Contras

- *C#* no provee de variables globales, pero se pueden simular mediante clases con dichas variables globales bajo el patrón Singleton.
- A día de hoy (inicios de 2015), la plataforma *.NET* se encuentra restringida al mundo Windows. Aunque existen algunos intentos de portabilidad con *MONO*<sup>44</sup>, los programas desarrollados en *.NET* todavía no son 100 % funcionales fuera de Windows. Se cree que para mediados del año 2015, con el nuevo cambio de CEO, *.NET* se pueda utilizar de manera correcta en otros Sistemas Operativos. Se puede obtener más información al respecto en <http://www.muylinux.com/2014/11/13/microsoft-plataforma-dot-net-open-source-linux>.

### 4.6.13. Servidor F#

#### Historia

F# es un lenguaje de programación multiparadigma de código abierto para la plataforma *.NET*, que conjunta la programación funcional con la imperativa y la orientación a objetos. Desarrollado inicialmente por Miguel Tentei Cortés y Dom Syme de *Microsoft Research*, actualmente está siendo supervisado por la *División de Desarrolladores de Microsoft*.

Se trata de un lenguaje fuertemente tipado que utiliza inferencia de tipos que, como resultado, no requiere el tipado explícito por parte del programador.

A diferencia de *C#*, F# facilita enormemente la creación de código asíncrono y paralelo, cosa que en otros lenguajes de *.NET* nos llevaría bastante más tiempo. Además, permite el desarrollo de código basado en agentes.

---

<sup>43</sup><https://unity3d.com>

<sup>44</sup><http://www.mono-project.com/>

## Desarrollo

El código desarrollado para este lenguaje utiliza el mismo paradigma que *C#*, es decir, programación asíncrona.

A diferencia de en *C#*, el código se simplifica bastante debido a la existencia de variables asíncronas y un uso de buzones para enviar la información a los clientes conectados.

El principal problema, el cuál coartó el posterior desarrollo completo de la aplicación era el manejo de JSON complejos, y diversos problemas asociados a ello. Por este motivo, el servidor en *F#* no se finalizó, aunque se dejó todo el código desarrollado en el disco disponible con este trabajo.

## Ventajas y desventajas

### Pros

- La sintaxis utilizada en *F#* es limpia y concisa, mucho más sencilla de entender (funcionalmente) que *C#*.
- *F#* provee de mecanismos para la programación asíncrona, en paralelo, programación orientada a objetos, imperativa, basada en agentes y meta-programación.
- Es muy fácil de integrar en Lenguajes Específicos de Dominio (DSL<sup>45</sup>).
- *F#* es bastante eficiente para algoritmos complejos y aplicaciones científicas y de finanzas.
- A diferencia de *C#*, *F#* es código libre, cuyo código se puede visitar en su página de Github: <https://github.com/fsharp/fsharp>.

### Contras

- El manejo de JSON complejos a veces resulta bastante difícil al no disponer de tantas librerías nativas como en *C#*.
- Desde un punto de vista funcional puro, *F#* permite la programación imperativa mientras ejecuta funcional, lo que conlleva que *F#* no sea programación funcional pura.

---

<sup>45</sup>Se trata de un tipo de lenguaje de programación o especificación dedicado a resolver un problema en particular, representar un problema específico y proveer una técnica para solucionar una situación particular.



#### 4.6.14. Servidor Akka (Scala)

##### Historia

*Akka* es un conjunto de herramientas Open Source para la construcción de sistemas concurrentes y distribuidos en la *JVM*. *Akka* soporta múltiples modelos de programación de concurrencia, pero está más empatizado con el modelo de concurrencia con actores, inspirado en *Erlang*<sup>46</sup>.

Los actores en *Scala* se desarrollaron en torno al 2006 por Philip Haller. Nació debido a la dificultad del desarrollo de programación de servidores complejos, repletos de memoria compartida y códigos sincronizados con locks. Para solucionar este problema, inspirándose en *Erlang*, Jonas Bonér desarrolló *Akka*.

Existen dos versiones en paralelo, con los que puedes desarrollar tanto en *Scala* (lenguaje original), como en *Java*.

##### Desarrollo

A la hora de desarrollar el servidor en *Scala*, en un principio se habían planteado dos posibilidades distintas: Una usando los actores propios de *Scala* y otro usando los actores de *Akka*.

Inicialmente se empezó con la primera opción, pero varios meses después, las librerías de *Scala* se actualizaron, desapareciendo con ello los actores. Como alternativa oficial, *Scala* ofrecía a los desarrolladores un manual de migración a los actores de *Akka* ([Scae]), por lo que al final solo nos pudimos decantar por la segunda opción (no obstante, se decidió entonces incorporar el uso de la API de *Akka* para *Java*, que se explicará más adelante, para comparar la eficiencia de ambos lenguajes).

Como se comenta en la historia de *Akka*, esta librería se basa en los actores de *Erlang*, lo que cambia en parte el sistema de desarrollo de concurrente o paralelo de los servidores desarrolladores con anterioridad, pero gracias a toda la información en la API oficial, es bastante sencillo migrar las ideas entre un modelo de concurrencia y otro.

Para desarrollar el servidor, como en cualquiera de los otros servidores, se ha utilizado un único actor cuya única labor es la de recibir conexiones, las cuales crean otros actores, que en nuestro caso son los propios clientes conectados.

Cada uno de estos actores dispone de dos acciones:

- **Received(data):** Cuando se recibe información.
- **Peer Closed:** Cuando el actor se desconecta.

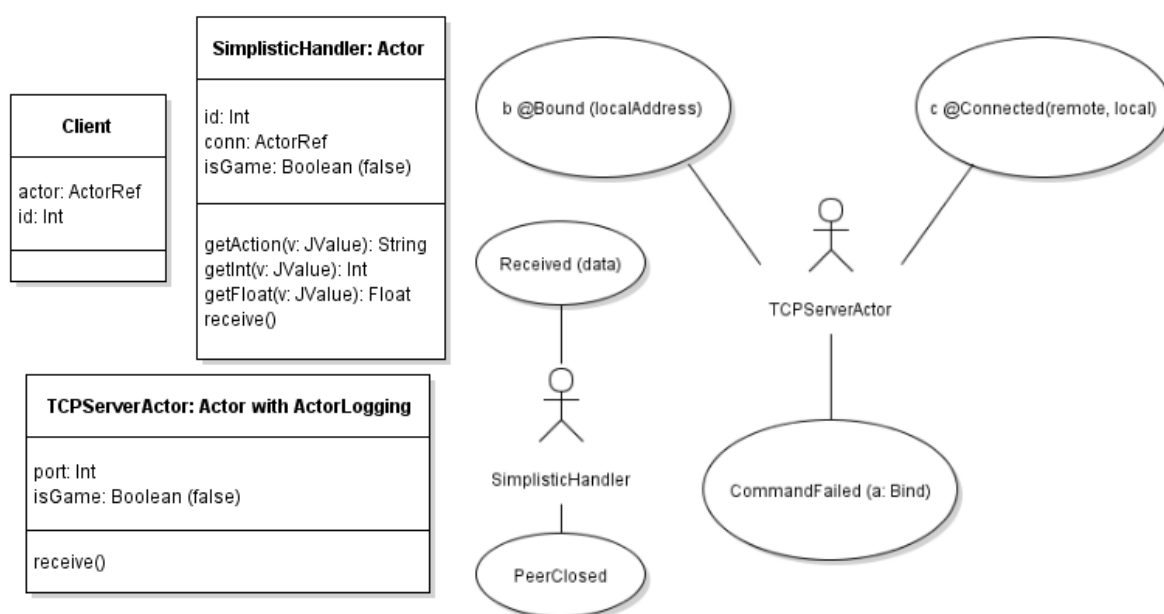
---

<sup>46</sup><http://www.erlang.org/>

Debido a que no se quería incrementar el nivel de dificultad de desarrollo, se optó por una solución sencilla sin uso de ACK's y de agentes para guardar las referencias al resto de actores: clases Singleton.

A la hora de gestionar los mensajes, se utilizó la librería *json4s*<sup>47</sup>, una librería muy sencilla de utilizar tanto para generar como para parsear cadenas JSON; y una lista mutable de clientes.

Esta solución basada en agentes facilita bastante la separación de labores de cada parte de la aplicación, pudiendo modificar fácilmente cambios sin que afecte al resto del programa (ver Figura 4.24).



**Figura 4.24:** Esquema servidor Akka(Scala) UML y Actores, respectivamente

## Ventajas y desventajas

### Pros

- *Akka* permite separar cada socket en diferentes actores, de forma que cada uno tenga “vida propia” y sea él mismo quien se encargue de comunicarse con el resto de actores.
- Al separar cada socket como un actor, podemos utilizar el actor maestro como “escuchador” y generar actores “slave”, que serán nuestros clientes conectados.

<sup>47</sup><https://github.com/json4s/json4s>

- La librería *Akka* es utilizada por muchas otras librerías, como *Apache Spark*<sup>48</sup>, *Spray*<sup>49</sup> o *Redis*<sup>50</sup>.
- Los actores de *Akka* son una alternativa a los ya extintos actores de **Scala** desde las versiones 0.10 de éste.
- *Akka* permite añadir fácilmente conexión a base de datos, envíos de ACKS y crear un sistema tolerante a fallos (ninguno de estos implementados).
- Nativamente no incluye soporte para JSON, pero la librería *Json4s* nos da un buen soporte tanto para crear como para parsear información.
- *Scala* es un lenguaje híbrido, por lo que soporta la programación funcional, iterativa, orientada a objetos, entre otras, además de casi un total acceso a todas las funcionalidades propias de *Java* (aunque no son recomendables).
- *TypeSafe Reactive Platform*<sup>51</sup> es un sistema creado por los desarrolladores de *Scala* y *Akka*, y provee diversas plantillas para desarrollar con *Scala*, *Akka* y *Java*, además de contener en su interior el sistema de gestión *SBT*.<sup>52</sup>, muy parecido a *Maven*, pero más abierto al mundo *Scala*.

### Contras

- Debido a que *Akka* está desarrollado en *Scala*, es muy difícil comprender a primera vista las funcionalidades básicas de la librería.
- Hay escasa información sobre desarrollo de servidores TCP con *Akka*, y casi todos los ejemplos difieren de la idea a implementar. Junto a esta problemática, el tratamiento de buffers resulta bastante engorroso.
- La sintaxis de *Akka* para *Scala* es un poco compleja si no se tienen conocimientos previos sobre *Scala*.
- Debido a que *Scala* no tiene mucho parecido al que es el principal lenguaje de la *JVM*, *Java*, ni es un lenguaje funcional puro, programar en este híbrido requiere grandes conocimientos del mundo de la programación funcional y del paradigma orientado a objetos.

---

<sup>48</sup><https://spark.apache.org/>

<sup>49</sup><http://spray.io/>

<sup>50</sup><http://redis.io/>

<sup>51</sup><https://typesafe.com/>

<sup>52</sup><http://www.scala-sbt.org/>

- *Scala* no es un lenguaje funcional puro, por lo que si se compara con otros lenguajes funcionales, perdería bastante potencia. Además, *Scala* es más difícil de usar si solo conoces el mundo *Java*.

#### 4.6.15. Servidor Akka (Java)

##### Historia

(No concluyente, es idéntica a la desarrollada en el Servidor *Akka* con *Scala*).

Es importante destacar que *Java* no dispone de actores de forma nativa, por lo que *Akka* provee de la funcionalidad necesaria para poder utilizarlos en *Java*.

##### Desarrollo

A diferencia de su contraparte con *Apache Mina*, y en conjunto con *Scala*, *Akka* permite cambiar el modelo de concurrencia y paralelismo mediante el uso de actores, basados en *Erlang*.

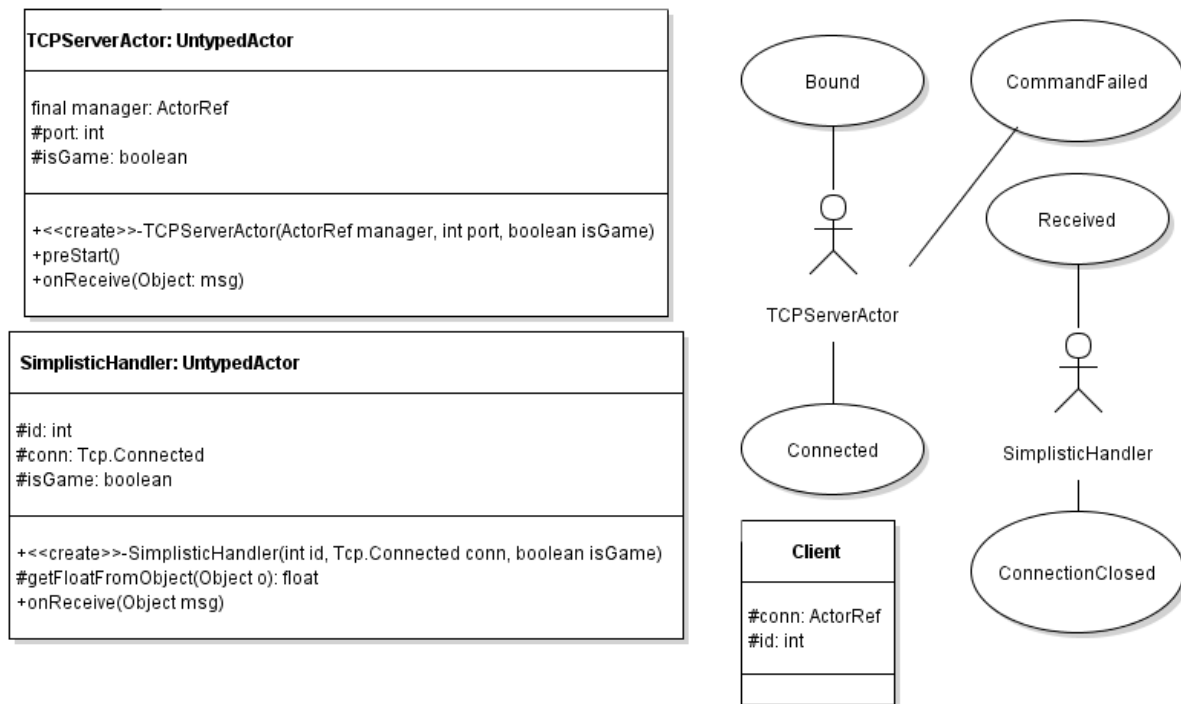
Debido al gran parecido de la API de *Java* con la de *Scala* (salvando excepciones), traducir el código fue algo bastante lineal, llegando al punto que ambos códigos eran un calco, únicamente cambiando la sintaxis, favoreciendo enormemente una exactitud de la funcionalidad de un 90 % (el único código distinto es la librería de gestión de JSON)<sup>4.25</sup>. Al igual que en *Scala* con *Akka*, guardamos las referencias a todos los clientes conectados con el fin de tener guardados todos los clientes y poder comunicar los distintos mensajes según las distintas acciones requeridas por los clientes.

##### Ventajas y desventajas

###### Pros

- La versión de *Akka Java* es mucho más rápida de codificar debido a que *Java* tiene más soporte a los autocompletados (generalmente por acceso al *JavaDoc* de las funciones) y a los proyectos *Maven*, a diferencia de *Scala*.
- Portar el código de *Akka* con *Scala* a *Akka* con *Java* es realmente rápido teniendo acceso a la API.
- Cambiar el modelo facilitado por *Apache Mina* al modelo basado en actores permite poder separar de forma más concisa todos los objetos que interactúan en nuestro sistema.

###### Contras



**Figura 4.25:** Esquema servidor Akka(Java) UML y Actores, respectivamente

- La sintaxis de la versión *Akka* con *Java* con la versión *Akka* con *Scala* difiere bastante, lo cual a veces conlleva que no exista una traducción directa entre ambos lenguajes.

#### 4.6.16. Servidor Groovy

##### Historia

*Groovy* es un potente lenguaje dinámico orientado a objetos con tipado opcional, con capacidad para compilar el tipado estático y dinámico que trabaja sobre la *JVM*.

Usa una sintaxis muy parecida a *Java*, compartiendo el modelo de objetos, hilos y seguridad, pero permite usar *Groovy* tanto de manera dinámica como lenguaje de scripting. También aprovisiona de funcionalidad como *DSL* o Lenguaje Específico de Dominio.

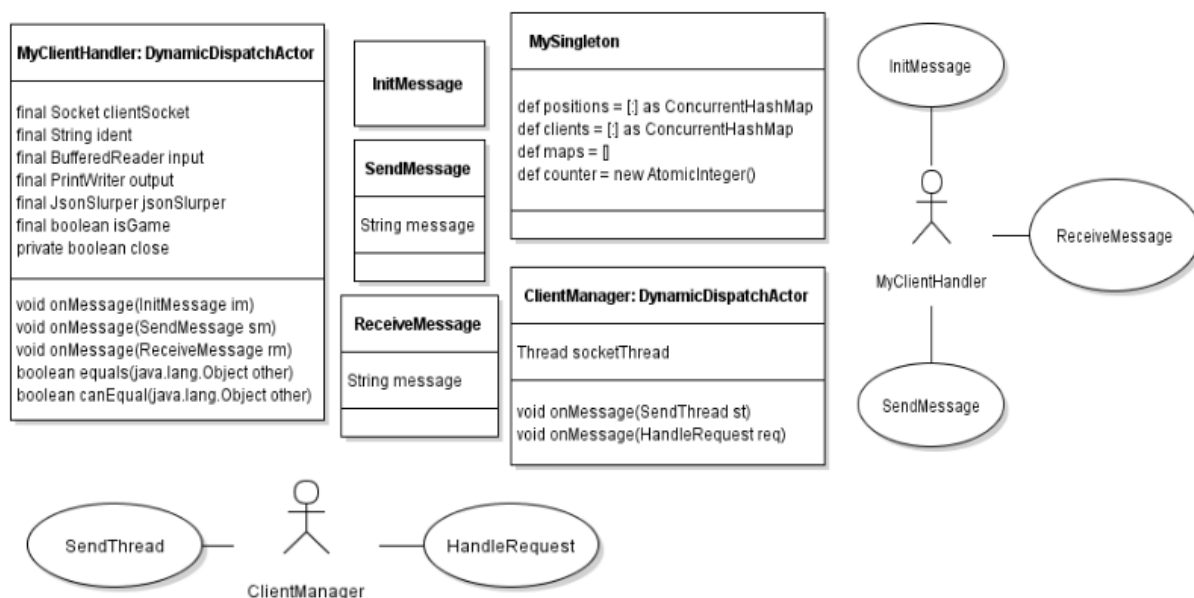
*GPar*s, por su parte, se trata de una librería para desarrollo concurrente basado en *Groovy*, que provee de hilos, procesos y actores, entre otras funcionalidades.

##### Desarrollo

La arquitectura de desarrollo utilizada para este servidor está basada en actores, con el fin de poder realizar una mejor comparación entre los tres lenguajes principales de la *JVM*: *Scala*, *Java* y *Groovy*. Debido a que *Groovy* no tiene actores por defecto (como en el caso de *Java* que se soluciona con *Akka*), este problema se soluciona con la librería

GPar<sup>53</sup>.

*Akka* internamente trabaja con *Akka IO* para proveer de funcionamiento a los actores para funcionar como sockets, pero *GPar*s no contiene esa funcionalidad, se ha tenido que simular usando actores que despachan las acciones de los sockets con estados internos<sup>4.26</sup>.



**Figura 4.26:** Esquema servidor Groovy UML y Actores, respectivamente

## Ventajas y desventajas

### Pros

- La curva de aprendizaje si se disponen conocimientos de *Java* es mínima, permitiendo mezclar código *Groovy* y *Java*.
- A diferencia de sus contrapartes *Java* y *Scala*, *Groovy* dispone de una librería interna para el tratamiento de JSON.
- Si se disponen de pocos conocimientos en lenguajes de scripting como *Bash* o *Visual*, puedes utilizar código *Java/Groovy* para realizar tareas automatizadas.
- Tiene un propio framework de desarrollo web con ciertas similitudes con *Ruby on Rails*: *Grails*<sup>54</sup>.
- Al igual que *Scala*, existe una terminal para poder ejecutar pruebas.

### Contras

<sup>53</sup><http://www.gpars.org/guide/index.html>

<sup>54</sup><https://grails.org/>

- Al tratarse de un lenguaje dinámico, gran parte de los errores no aparecen en tiempo de compilación (a diferencia de *Scala* y *Java*). Además, en parte es mucho más lento que *Java*, aún a pesar de compilar a *Bytecode*.

### 4.6.17. Servidor Julia

#### Historia

Julia es un lenguaje dinámico de alto nivel diseñado para ofrecer un alto rendimiento para computación numérica y científica, además de ser bastante efectivo para programación de propósito general. El nombre del lenguaje Julia fue puesto en honor a Gaston Julia, un matemático francés que descubrió los fractales.

Entre los aspectos distintivos de Julia destaca un sistema de tipado paramétrico en un lenguaje de programación completamente dinámico, además del envío múltiple como paradigma de programación distribuida. Permite programación paralela y computación distribuida, además de acceso directo a bibliotecas de C y Fortran.

Otras particularidades de Julia son el uso de un recolector de basura por defecto, bibliotecas muy eficientes de punto flotante, álgebra lineal, generación de números aleatorios, expresiones regulares y rápidas transformadas de Fourier.

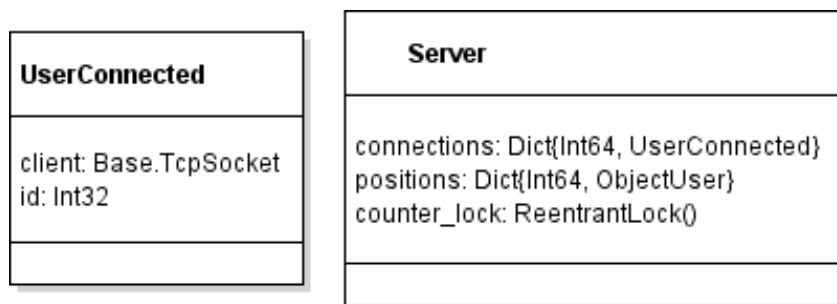
#### Desarrollo

El modelo de desarrollo concurrente propuesto con el lenguaje Julia consiste en un uso intensivo de bloques que se ejecutan de forma asíncrona.

En el servidor desarrollado existe un bloque que es utilizado para aceptar las nuevas peticiones y, cuando un usuario se conecta, se ejecuta otro bloque asíncrono mediante el uso de una corutina (*@async*) y, de esta forma, conviven en la ejecución del programa tantos hilos o procesos como clientes conectados hubiese4.27.

Debido al uso de la memoria compartida y la inexistencia de variables atómicas, se ha requerido un uso de *ReentrantLock* para evitar la inconsistencia de los datos. Al igual que en el resto de servidores, primeramente intentamos convertir a un objeto JSON el mensaje recibido y, según el valor del campo “*Action*”, se ejecuta una cantidad de instrucciones específicos.

A pesar de intentar solucionar diversos errores, las últimas versiones de Julia impedían la total ejecución del proyecto. Es por ello por lo que no se pudo llegar a terminar las diferentes acciones y sus respectivos tests de rendimiento, ya que ocurrieron bastantes cambios en la API del lenguaje.



**Figura 4.27:** Esquema servidor Julia

## Ventajas y desventajas

### Pros

- A pesar de su escasa documentación oficial, es posible entender ciertas funcionalidades a partir de la lectura del código fuente.
- Las librerías de JSON son bastante completas, además de sus librerías nativas para realizar cast.
- Al tratarse de un lenguaje dedicado a la alta computación numérica y científica, su sintaxis es bastante sencilla, muy parecida a *Python*. Además, ofrece gran funcionalidad para la programación distribuida y paralela.
- Debido a la gran habilidad para cálculos matemáticos y estadísticos, muchos programadores defienden que *Julia* puede presentarse como una alternativa a *Apache Hadoop*<sup>55</sup>.

Podemos obtener información al respecto en <http://geektheplanet.net/7810/julia-un-lenguaje-de-programacion-para-gobernarlos-todos.xhtml>. Además, *Julia* es bastante más rápido que *Python* o *R*<sup>56</sup> para temas de Big Data, debido a la facilidad de separar el tratamiento de datos en varios ordenadores en paralelo.

- El diseño de nuevos tipos de usuario se basan en un uso intensivo de estructuras, muy parecidas a *C* y *Fortran*<sup>57</sup>.

### Contras

- Muchas funcionalidades utilizadas en la realización del servidor no aparecen en la última versión estable, teniendo que optar por una versión “*nightly*”<sup>58</sup>.

<sup>55</sup><https://hadoop.apache.org/>

<sup>56</sup><http://www.r-project.org/>

<sup>57</sup>[www.fortran.com/](http://www.fortran.com/)

<sup>58</sup>Una versión nightly se trata de una compilación neutral que se lleva a cabo de forma automática cada cierto tiempo. Al no haber sido probada por los desarrolladores, puede contener errores.



- Debido a que *Julia* lleva pocos años en producción, tiene muchos errores. Por ejemplo, si se produce un error, a veces el programa se queda bloqueado sin saber la respuesta.
- La separación de acciones en el servidor TCP es bastante compleja, por lo que estaría bien que existiera un desarrollo de un middleware para poder separar bien, al menos, la conexión, desconexión, broadcast y excepciones, al igual que otros frameworks presentados en este proyecto, como *Apache Mina*.



# Capítulo 5

## Resultados Experimentales

### 5.1. Estudio comparativo

En esta sección nos centraremos en las comparaciones entre duplas de librerías / frameworks / lenguajes, con el fin de encontrar cuál es la mejor solución de desarrollo en tecnologías parecidas.

Por ello, se han establecido diversas parejas de lenguajes y librerías, en las cuales se analizarán diversos aspectos positivos y negativos.

#### 5.1.1. Comparativa Scala vs. NodeJS

*NodeJS* se trata de un lenguaje que permite crear de forma sencilla aplicaciones asíncronas no bloqueantes, con una estupenda IO muy requerida en el mundo web. Esto, en conjunción con el lenguaje que utiliza por debajo, hace que sea un gran ecosistema de desarrollo. No obstante, el gran problema de este lenguaje es que es bastante pobre en la gestión de tareas que requieran mucha CPU.

Por el contrario, *Scala* provee de muchas optimizaciones debido a que corre bajo la *JVM*, que lleva más de 20 años en activo. A diferencia de *NodeJS*, su ecosistema es más amplio, pero el aprendizaje de la programación en *Scala* es mucho más complicado.

En cuanto a nivel de tipado, *NodeJS* tiene tipado dinámico y *Scala* tiene tipado estático, incluso uso de variables inmutables para aumentar la performance en aplicaciones distribuidas o multihilo.

Para finalizar, es importante remarcar la existencia actual de un proyecto que permite obtener lo mejor de ambos ecosistemas: *ScalaJS* (<http://www.scala-js.org/>). Se trata de una librería que compila el código desarrollado en *Scala* a *JavaScript*, permitiendo optimizar (en parte), algunos proyectos conllevados de la programación asíncrona por parte de *NodeJS*, que se solventa fácilmente con el uso de actores de *Scala*.

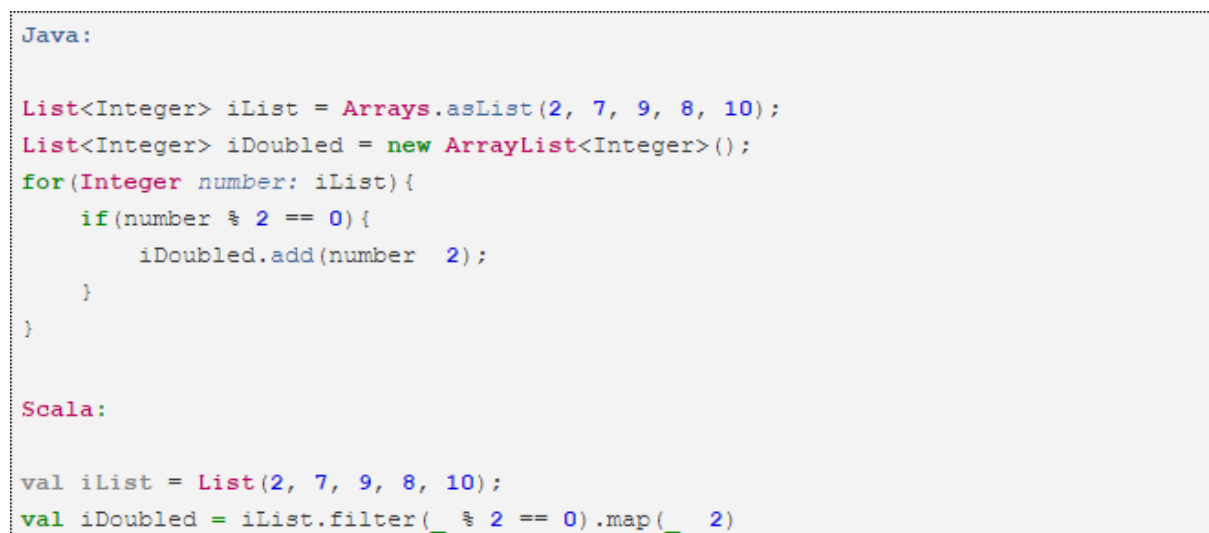
### 5.1.2. Comparativa Scala vs. Java

*Scala* y *Java* son dos lenguajes que se apoyan sobre la máquina virtual *JVM* para trabajar. Ambos lenguajes permiten desarrollar bajo OOP, pero *Scala* permite además programación funcional avanzada (aunque en *Java* en su última versión (*Java 8*) permite algunas funcionalidades reducidas de *Scala*).

El desarrollo con *Scala* es bastante intuitivo y evita la repetición continua de código, pero esto a veces resulta un problema, debido a que, una modificación de un campo de un objeto no es realmente trivial si usamos programación funcional pura.

A nivel de entornos de desarrollo, *Java* tiene mayor cantidad de Entornos Integrados de Desarrollo nativos (*Netbeans*, *Eclipse*, *IntelliJ*), mientras que *Scala* necesita apoyarse en los entornos de *Java* para poder trabajar. No obstante, el propio creador de *Scala*, Martin Odersky, ha desarrollado la plataforma *Typesafe*<sup>1</sup>, que provee de un entorno de desarrollo basado en Web, con gran cantidad de plantillas con *Apache Spark*, *Apache Kafka*, entre otros.

Sin duda una de las grandes ventajas de *Scala* frente a *Java* es la compilación a *Bytecode* en caliente, cosa que *Java* no permite. Otro de los puntos fuertes es la simplicidad de código, como se puede ver en la siguiente figura comparativa (ver Figura 5.1), además de poder ejecutar gran parte de la librería nativa de *Java*: Desde el punto de vista de la



```
Java:

List<Integer> iList = Arrays.asList(2, 7, 9, 8, 10);
List<Integer> iDoubled = new ArrayList<Integer>();
for(Integer number: iList){
    if(number % 2 == 0){
        iDoubled.add(number * 2);
    }
}

Scala:

val iList = List(2, 7, 9, 8, 10);
val iDoubled = iList.filter(_ % 2 == 0).map(_ * 2)
```

**Figura 5.1:** Comparativa código Java vs. código Scala

biblioteca *Akka*, el desarrollo con *Scala* resulta mucho más rápido e intuitivo porque la cantidad de código a utilizar es menor frente al código en *Java*.

Para finalizar esta comparativa, es importante remarcar tres de las dificultades de la programación en *Scala*:

---

<sup>1</sup><http://www.typesafe.com/>

- Si vienes del mundo *Haskell*<sup>2</sup>, *Scala* no es programación pura, puesto que permite diseñar código iterativo.
- Si vienes del mundo *Java*, resulta complicado tener en cuenta el uso de variables inmutables, a pesar de conseguir con esto la facilidad de la programación multihilo.
- *Scala* está centrado en el uso de actores, mientras que *Java* está más centrado en el uso de hilos.

### 5.1.3. Comparativa Groovy vs. Java

Esta comparación es bastante trivial, ya que *Groovy* es un superconjunto de *Java*, lo que conlleva poder ejecutar toda la librería nativa de *Java* usando un lenguaje de scripting como *Groovy*.

Sin duda, hay varios puntos por lo que *Groovy* destaca frente a *Java*:

- **Importaciones por defecto:** No es necesario importar todas las librerías por defecto, ya que el mismo compilador las declara explícitamente.
- **Comparación de objetos y cadenas:** Uno de los problemas más importantes cuando un desarrollador intenta migrar de otros lenguajes a *Java* o de *Java* a otros lenguajes es la comparación de objetos y cadenas. *Groovy* soluciona este problema con el uso de “==”, que actúa como wrapper al método *equals*. Además, provee otra nueva función “*Object.is(Object)*” para saber si dos objetos son iguales.
- **Puntos y coma:** No son necesarios (herencia de otros lenguajes de scripting como *Ruby* o *Python*).
- **Return opcional:** No es necesario incluir el retorno de la función, al igual que ocurre con *Scala*.
- **Simplicidad de código:** Al igual que se demostró en la comparativa de *Scala* contra *Java*, *Groovy* permite simplificar bastante el código utilizado, lo que facilita la lectura.
- **Métodos y clases públicos:** A diferencia de en *Java*, los métodos y atributos son públicos por defecto (en *Java* son “friendly” o de “paquete”).
- **Programación “funcional”:** Aunque en *Java 8* ya se permite un uso de programación funcional usando *Java*, *Groovy* ya permitía (al igual que *Scala*), un uso de programación funcional nativa.

---

<sup>2</sup><https://www.haskell.org/>

- **Nuevas funcionalidades no existentes en Java:** Metaprogramación, librerías nativas de JSON, mejoras en el switch-case, sintaxis nativa para listas y mapas (más parecida a lenguajes como *C++*), getters and setters automáticos, entre otros.

No obstante, Groovy también tiene defectos frente a *Java*:

- **Errores en tiempo de ejecución:** Al tratarse de un lenguaje dinámico, *Groovy* no muestra todos los errores, cosa que pasa igual que en otros lenguajes de scripting.
- **Lentitud:** Al no ser un lenguaje compilado directamente a *Bytecode*, el rendimiento para determinado tipo de aplicaciones es bastante más lento en *Groovy* que en *Java*.

#### 5.1.4. Comparativa Java Akka y Apache Mina

*Akka* y *Apache Mina* proveen de funcionalidad para mejorar el diseño de servidores, tanto HTTP, como TCP o UDP a través de una API sencilla (aunque a veces un tanto incompleta).

Mientras que *Apache Mina* se centra en un paradigma más centrado en un modelo Reactivo, *Akka* se centra en un paradigma orientado a actores. La diferente principal entre ambos paradigmas es que el modelo Reactivo solo funciona correctamente cuando el servidor recibe eventos del exterior, todo dentro de un mismo módulo (o varios), mientras que un modelo basado en actores permite separar cada instancia conectada (cliente) en objetos separados, de forma que cada uno pueda tener su tipo de funcionalidad propia (esto supone una mayor limpieza de código frente al modelo reactivo).

En cuando a la dificultad para desarrollar en ambas tecnologías, el framework que más ha costado utilizar ha sido *Akka* debido a ser un paradigma menos habitual, aunque los resultados de éste y la limpieza de código son mejores que los de *Apache Mina*.

#### 5.1.5. Comparativa Ruby y Event Machine

Esta comparativa realmente puede ser bastante breve, debido a los problemas encontrados para el desarrollo concurrente con *Ruby* en los inicios del proyecto.

*Ruby* es un lenguaje en el que es muy fácil desarrollar, pero esta facilidad repercute fácilmente en la rapidez de un sistema cuando se encuentra en ejecución (en este caso, interpretando).

La velocidad del selector de *Ruby* nativo es bastante lento y es muy fácil conseguir bloquearlo para aplicaciones en tiempo real. Por ello, *EventMachine* consigue, basándose en la arquitectura Reactiva, mejorar rápidamente todos los problemas de los canales y selectores a través de un uso masivo de pools de hilos, además de que permite separar los

distintos eventos de un servidor (nuevas conexiones, mensajes, desconexiones, ...)

En resumen, *EventMachine* mejora enormemente de una facilidad para el desarrollo de proyectos distribuidos en tiempo real.

### 5.1.6. Comparativa Python y Python Twisted

A diferencia del combo *Ruby-Event Machine*, el procesamiento con *Python* con sus librerías nativas dan un buen soporte y escalabilidad, y *Python Twisted* utiliza el mismo paradigma que *Event Machine* (aunque más cercano a *NodeJS*).

La facilidad de desarrollo en *Python* simplifica en gran parte toda la gestión de hilos y procesos a través de una completa API. Además, las últimas versiones de *Python* han mejorado en gran parte todos los problemas del pasado, en cuanto a lentitud y problemas asociados a instrucciones atómicas.

En contraparte, *Python Twisted* permite abstraernos todavía más de la gestión de servidores, separando las distintas acciones en diferentes funciones con posibilidad de sobreescritura: nuevas peticiones, desconexiones, tratamiento de mensajes, ....

### 5.1.7. Comparativa Java y C# (Concurrencia)

Como hemos podido comprobar en el anterior capítulo, *Java* de forma nativa provee de funcionalidad para desarrollo con hilos, es decir, basándose en una arquitectura de threads, aunque las bibliotecas *Apache Mina* y *Akka* consiguen la arquitectura reactiva y basada en actores, respectivamente. El problema de *Java* es que las actualizaciones de las librerías nativas son muy escasas, por lo que muchas funcionalidades son difíciles de conseguir.

*C#*, en cambio, es una plataforma mucho más rica debido a que *Microsoft* pone mucho más empeño que *Oracle* para mejorar sus plataformas. *C#* provee funcionalidad para desarrollo con hilos al igual que en *Java*, pero de forma nativa incluye una arquitectura asíncrona, enriqueciendo en gran parte la programación concurrente. No obstante, existen otras bibliotecas de terceros que permiten el uso de programación basada en actores, como *n-act*<sup>3</sup>, al igual que hace *Akka* en *Java* y *Scala*.

### 5.1.8. Comparativa Groovy vs. Ruby

Sin lugar a dudas, *Ruby* y *Groovy* tienen muchos importantes en común, empezando directamente por su principal framework de desarrollo web: *Rails* y *Grails*.

Para comparar ambos lenguajes, vamos a utilizar varios factores:

---

<sup>3</sup><https://code.google.com/p/n-act/>

- **Tipo de lenguaje:** Ambos lenguajes son lenguajes dinámicos, interpretados y con una fuerte orientación a objetos y con tipado dinámico (aunque *Groovy* permite tipado estático si se le avisa que actúe como tal).
- **Sintaxis:** La sintaxis de *Ruby* está más orientada a las conversaciones humanas, tal y como lo defiende su autor original (leer Arquitectura de *Ruby* en el capítulo anterior), mientras que la sintaxis de *Groovy* tiene más al mundo *Java* y *C++*.
- **Tiempo de desarrollo:** En conjunción con el factor anterior, Ruby permite abstraer más la información, aunque a veces esta abstracción resulta más compleja. Si, en cambio, vienes del mundo *Java*, migrar a *Groovy* no resulta nada complicado.
- **Madurez:** Ambos lenguajes llevan bastantes años en el mercado, aunque *Ruby* no ha tenido tanta cantidad de desarrollo hasta la versión 3 de su framework web *Ruby on Rails* (año 2011, aproximadamente). *Groovy* por su parte se ayuda del framework *Spring*<sup>4</sup> de *Java*, añadiendo nuevas funcionalidades que simplifican el código *Java*.
- **Librerías de testing:** Ruby está muy centrado en el mundo del testing *BDD*, lo que hace que existan muchos frameworks para testing automático, como *RSpec*<sup>5</sup>, *Cucumber*<sup>6</sup>, ..., mientras que la principal librería de testing de *Groovy* proviene del mismo *Java*: *JUnit*<sup>7</sup>.

### 5.1.9. Comparativa Scala vs. Go

*Go* y *Scala* son dos lenguajes modernos de desarrollo. En esta sección trataremos alguna de sus diferencias más significativas:

- **Compilación:** *Go* es un lenguaje minimalista que compila a código máquina, mientras que *Scala* es un lenguaje funcional y orientado a objetos que corre en la *JVM*.
- **Tipo de programación:** *Scala* permite una orientación funcional (no pura) y programación orientada a objetos, mientras que *Go* permite una simulación de orientación a objetos a través de estructuras (heredado de *C*).
- **Tipado:** Ambos lenguajes permiten tipado estático y dinámico.
- **Consumo de memoria:** Al tratarse de un lenguaje compilado a código máquina, *Go* provee de un menor gasto de memoria llegando a los pocos MB en ejecución,

---

<sup>4</sup><https://spring.io/>

<sup>5</sup><http://rspec.info/>

<sup>6</sup><https://cucumber.io/>

<sup>7</sup><http://junit.org/>



mientras que *Scala* consume bastante más memoria (alrededor de un 10 % más) al tener que depender de la *JVM*.

- **Facilidad de aprendizaje:** No se puede hacer una buena comparación en esta sección, debido a que cada lenguaje resulta más fácil según el nivel de conocimiento anterior. *Go* es un lenguaje más orientado al mundo *C* y de sistemas, mientras que *Scala* está más orientado al mundo funcional y al procesamiento masivo de datos.
- **Concurrencia:** Uno de los puntos claves de *Scala* es el uso de actores, que tienen la habilidad de interconectar diferentes máquinas remotas de forma transparente. *Go*, por el contrario, utilizando gorutinas y canales, requiere una mayor habilidad para interconectar diferentes máquinas remotas.
- **Sintaxis:** Otro de los puntos fuertes de *Scala* es que se necesitan pocas instrucciones para ejecutar algoritmos complejos, mientras que en *Go* a veces resulta bastante complejo la codificación eficiente de estos algoritmos.
- **Documentación oficial:** *Go* habilita a los desarrolladores de una API documentada en su página web, sencilla y eficaz. *Scala*, por su parte, deja a disposición de los usuarios una API que varía en parte entre versiones distintas de *Scala* (por ejemplo, a partir de las versiones superiores a la 10, los actores nativos son borrados de la librería oficial).

#### 5.1.10. Comparativa Scala y F#

Esta comparativa queda un poco en el aire al no haber podido completar todo el desarrollo en *F#* por problemas del manejo de JSON. No obstante, si se puede realizar una breve comparación entre ambos lenguajes que proveen de programación funcional al mundo *Java* y *.NET*, respectivamente.

- Desde un punto multiplataforma, *Scala* funciona en cualquier Sistema Operativo que permita la ejecución de la *JVM*, mientras que *F#* se centra particularmente en Windows, aunque es ejecutable, a día de hoy, en otros Sistemas Operativos a través de *MONO*.
- Desde un punto de vista más funcional, *F#* obliga a las todas las variables y objetos (incluidos atributos) a ser inmutables durante su ejecución, mientras que *Scala* permite tener variables y objetos variables.
- El concepto de currificación en *F#* está más extendido y es más fácil de utilizar a través del operador `|>`.

- En cuanto al tema de pattern matching, *Scala* utiliza una sintaxis muy parecida a *Java* (*case*), mientras que *F#* utiliza una sintaxis más parecida a *Haskell* (*match*).

### 5.1.11. Comparativa Go y NodeJS

La comparativa entre *Go* y *NodeJS*, objetivo principal del proyecto, es la parte más importante de todo este proyecto, pues fue el comienzo que desencadenó la gran extensión del trabajo aquí presentado.

*Go* y *NodeJS* son dos lenguajes que actualmente copan gran parte del mercado en lo que a nuevas tecnologías se refiere.

Sin lugar a dudas, *NodeJS* consigue desbancar fácilmente a *Go* por una sencilla razón: *JavaScript* resulta más sencillo de desarrollar que un lenguaje sin OOP, estructurado, muy parecido a *C*. *NodeJS* es un lenguaje que no requiere demasiada dificultad para usarlo, lo cual es muy óptimo para desarrolladores web, frente a *Go*, que es menos dificultoso para desarrolladores de sistemas.

Para comparar ambas plataformas, realizaremos una pequeña comparación de varios puntos clave:

- **Madurez:** *Go* provee de una mayor madurez debido a que tiene una API bastante más robusta, mientras que la API de *NodeJS* está siempre en constante cambio. Además, debido a la existencia de diversas charlas y conferencias por parte del departamento informático de Google en los *Google-IO*<sup>8</sup>, se ha favorecido el uso de esta plataforma en la comunidad de desarrollo.
- **Performance:** *Go* tiene un performance muy parecido a *C/C++* y genera un ejecutable compilado, mientras que *NodeJS* requiere una máquina virtual (*V8*) para interpretar el código.
- **Concurrencia:** El punto fuerte de *Go* es la concurrencia, llegando a superar en la mayoría de los casos a *C*. Su sistema de concurrencia está basado en el uso de selectores, canales y corutinas, mientras que *NodeJS* permite “simular” la concurrencia a través de librerías como “*async*”<sup>9</sup> y las instrucciones nativas “*setInterval*” y “*setTimeout*”, aunque principalmente funciona como una aplicación monohilo.
- **Escalabilidad:** *Go* está diseñado para una correcta escalabilidad y concurrencia con pocos problemas añadidos, mientras que *NodeJS* a veces resulta difícil de escalar.

---

<sup>8</sup><https://events.google.com/io2015/>

<sup>9</sup><https://github.com/caolan/async>

- **Facilidad de desarrollo:** *NodeJS* es un lenguaje bastante sencillo si provienes del mundo web, mientras que *Go* requiere conocimientos de programación concurrente avanzada para sacar el máximo rendimiento, además de conocimientos en programación estructurada.
- **Librerías:** *NodeJS* provee de mayor cantidad de librerías debido a su facilidad de desarrollo, mientras que *Go*, al tener un mercado mucho menor, provee de menos funcionalidades, aunque muchas librerías para añadir más funcionalidad a *NodeJS*, ya se encuentran de forma nativa en *Go*. No obstante, uno de los grandes atractivos de *Go* es poder utilizar casi sin dificultad librerías de *C*, permitiendo acceder a API's del mismo Sistema Operativo.
- **Manejo de JSON:** Sin lugar a dudas, este es el punto por el que *NodeJS* puede sobresalir frente a *Go*, debido a que el sistema de objetos de *NodeJS/JavaScript* es JSON, mientras que el de *Go* son las estructuras. A pesar de que *Go* dispone de librerías nativas para el marshall/unmarshall de JSON, su velocidad y gasto de memoria es bastante abusivo frente a *NodeJS*.

En resumen, *Go* es un lenguaje mucho más óptimo que *NodeJS*, aunque mucho más difícil de utilizar y con un menor mercado, pero provee de mayor eficiencia y escalabilidad frente a *NodeJS*.

## 5.2. Pruebas de tiempo y rendimiento

### 5.2.1. Cliente de prueba

A la hora de verificar el funcionamiento y rendimiento de cada servidor desarrollado, se ha utilizado el cliente de prueba descrito en el capítulo anterior es un portátil Asus con las siguientes especificaciones:

- Procesador Intel(R) Core(TM) i7-3537U CPU @2.00Ghz 2.50GHz
- Memoria instalada(RAM) de 8,00 GB
- Sistema Operativo Windows 8.1 Pro de 64 bits, procesador de x64.

Es importante destacar que algunos de los servidores ya contienen un uso intensivo de Base de Datos MySQL, lo que puede conllevar problemas de rendimiento al estar tanto el servidor, los clientes y la base de datos interactuando sobre una misma máquina sin ninguna distribución. Los servidores con Base de Datos en su lógica de negocio están marcados con un “\*”.

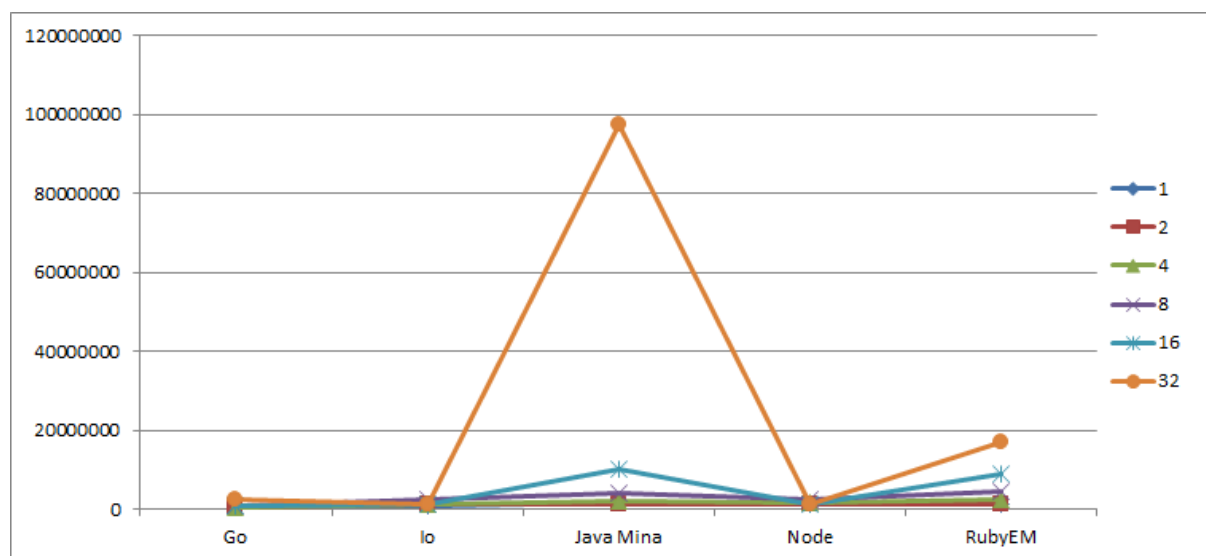
Dados los datos obtenidos tras pasar los tests con el cliente de prueba, recogidos en la tabla 6.1, podemos clasificar los resultados en servidores con o sin base de datos:

### Con Base de Datos

La interacción con Base de Datos requiere que no se pueda enviar el “ACK” hasta que no se ha terminado la consulta (en producción, las inserciones o actualizaciones se realizan de forma asíncrona utilizando hilos para agilizar el rendimiento). Hay que tener en cuenta que cada driver para *MySQL* tiene un rendimiento distinto, al igual que los pool de conexiones. Podemos ver estos resultados en 5.2

Los peores resultados los hemos obtenido con *Apache Mina*, obteniendo el mayor más alto de tiempo de ejecución con 32 clientes (en torno a los 97.595.312). Por otro lado, los mejores resultados se han obtenido con unos 2.405.786. Es por ello por lo que podemos determinar que *Go* es el lenguaje más idóneo en cuanto a rendimiento.

Desde un punto de vista de la codificación, la sintaxis de *Go* no es la más sencilla. Por ello, podemos seleccionar al combo *IOJS/NodeJS* como la mejor solución en cuanto a rendimiento (segundo puesto) y facilidad de desarrollo.



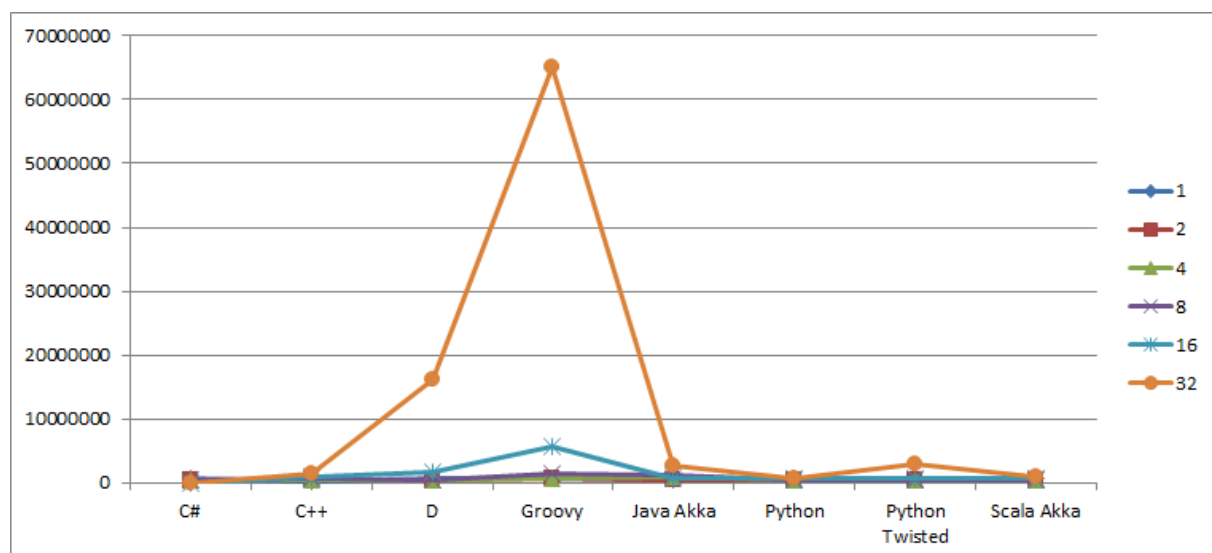
**Figura 5.2:** Gráfica de rendimiento de servidores con base de datos

### Sin Base de Datos

Debido a que en este caso no existe un pequeño delay por parte de los mensajes obtenidos desde los clientes, en estos tests los servidores no mandan el “ACK” hasta que ha finalizado las tareas de modificación (en estructuras globales dentro del servidor). Así mismo, el “ACK” de finalización requiere esperar a la eliminación del cliente del servidor (sin borrar el cliente en sí, solo su referencia en la estructura de datos correspondiente) para

luego mandar la respuesta para simular un delay como si de una petición a una base de datos se tratase. Podemos ver estos resultados en 5.3

El peor resultado lo hemos obtenido con *Groovy*. Si bien se cree que estos peores resultados se han podido obtener debido a necesitar del entorno *Eclipse*, tiene el peor rendimiento con creces (aproximadamente 65.191.410). Los mejores resultados se han obtenido con *Scala*. Como se comentó en capítulos anteriores, los actores permiten separar cada cliente de forma autónoma, sin necesidad de que el servidor sepa de su existencia. Esta separación permite que los propios actores puedan comunicarse con el resto y cambiar de estado como pequeños subprocesos o subhilos dentro del servidor. Si bien *Scala* no es un lenguaje muy utilizado en el mercado, provee de una sintaxis limpia y sencilla, lo cual facilita el desarrollo e implementación de sistemas. Esta facilidad junto a la separación de acciones en actores y estados permite separar cada componente de forma sencilla y con poco acoplamiento.

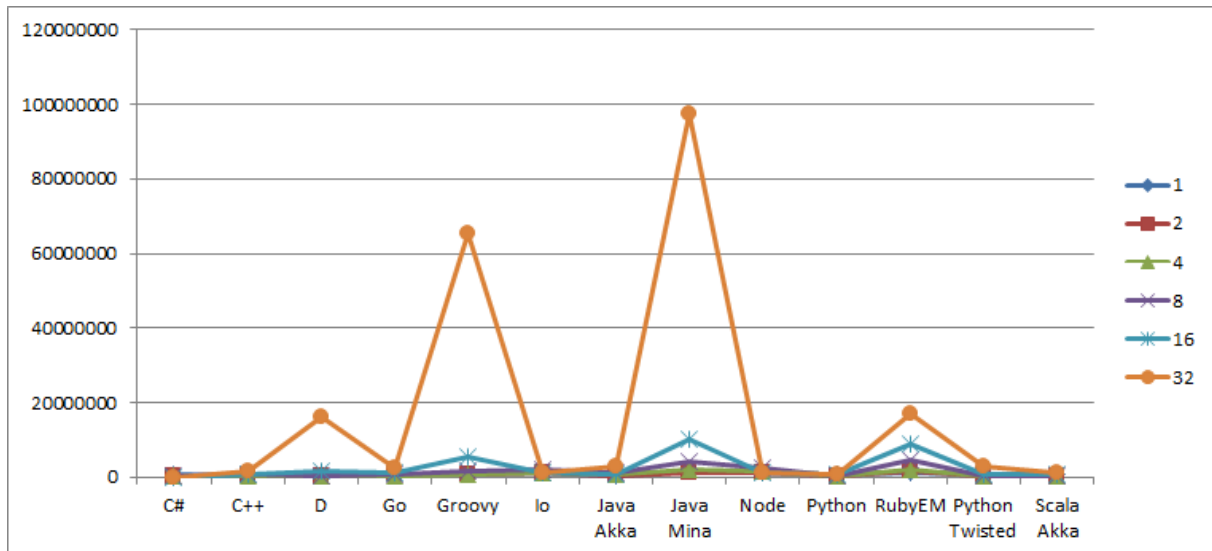


**Figura 5.3:** Gráfica de rendimiento de servidores sin base de datos

## Visión global

Desde una revisión global de todos los valores obtenidos en el experimento realizado (ver Figura 5.4), podemos considerar que *Go*, *NodeJS/IOJS*, *Python* y *Scala (Akka)* tienen los mejores resultados, aún a pesar de que los primeros ya tienen integrado acceso de base de datos.

Al igual que en los otros subapartados, los peores resultados se han obtenido con *Groovy* y *Apache Mina*.



**Figura 5.4:** Gráfica de rendimiento de servidores global

### 5.2.2. Comparativas JSON

Debido a que de forma teórica (y a la hora de desarrollar), se han tenido diversos problemas (o facilidades, según el caso) para la parsear o generar JSON con el fin de comunicarse entre servidores y clientes, en esta sección se incluirá una tabla comparando la facilidad de cada una de las librerías, tanto nativas como de terceros, para la gestión de este tipo de mensajes/objetos (ver Tabla 5.5).

Lenguaje	Marshall	Unmarshall
Node	JSON es la notación nativa para los objetos JavaScript.	JSON es la notación nativa para los objetos JavaScript.
Go	La API nativa de GO es lenta y complicada para JSON complejos. Uso de ffjson.	La propia API nos permite fácilmente el unmarshall.
C++	No incluye API de JSON. Uso de Jzon.	No incluye API de JSON. Uso de Jzon.
C#	Demasiadas librerías para tratamiento de JSON. Difícil de usar fuera de ASP	Demasiadas librerías para tratamiento de JSON. Difícil de usar fuera de ASP
D	La API nativa de D es complicada de utilizar para JSON complejos. Código de terceros.	D permite generar mensajes JSON fácilmente.
Ruby (Ruby, EventMachine)	La API nativa es muy completa.	La API nativa es muy completa.
Java (Apache Mina, Akka)	No incluye API de JSON. Uso de librerías de terceros.	No incluye API de JSON. Uso de librerías de terceros. Posibilidad de sobrescribir la generación de JSON de una clase.
Python (Python, Twisted)	La API nativa es complicada para JSON complejos.	La API nativa es complicada para JSON complejos.
Scala (Akka)	No incluye API de JSON. Uso de JSON4s.	La API nativa es complicada para JSON complejos.
Groovy	El propio lenguaje permite parsear objetos JSON.	La propia librería permite obtener JSON, aunque no devuelve unos resultados muy exactos.
Julia	El paquete JSON provee de buenos resultados.	El paquete JSON provee de buenos resultados.

**Figura 5.5:** Tabla comparativa de tratamiento de JSON en los servidores implementados





# Capítulo 6

## Conclusiones y trabajos futuros

En las secciones anteriores se han presentado diversas implementaciones que definen una arquitectura común utilizando diferentes lenguajes, librerías y paradigmas centrados en la programación en red. A su vez, a través de un cliente gráfico en forma de videojuego ha ayudado a determinar y realizar un desarrollo centralizado en pruebas, tanto de rendimiento como de carga.

A lo largo de esta sección se valorarán todos los resultados obtenidos, así como varias valoraciones personales sobre todo lo que se ha logrado.

### 6.1. Objetivos

Los dos grandes objetivos marcados desde el inicio del proyecto fueron el lenguaje Go y el desarrollo de aplicaciones basadas en Sistemas Distribuidos. Estos objetivos se han cumplido, obteniéndose muy buenos resultados, lo que conlleva un buen feedback frente a todo lo que se esperaba poder realizar.

El primer objetivo (Practicar con Sistemas Distribuidos), se ha podido completar con éxito gracias a una arquitectura bien planificada, que permite extender sin dificultad las diferentes acciones de los servidores a través de un modelo Cliente-Servidor.

El segundo objetivo (Desarrollo de videojuegos) se ha completado con éxito gracias a la librería *SFML*. Esta librería nos ha permitido encapsular toda la funcionalidad compleja de *OpenGL* de forma que el videojuego pueda ser portable. Además, como ya se mencionó en capítulos anteriores, *SFML* permite desarrollar en otros lenguajes distintos al lenguaje principal: *C++*.

El tercer objetivo (Trabajar con Go) ha sido uno de los puntos más difíciles de desarrollar y completar, debido a la gran cantidad de formas en las que los servidores se podían desarrollar. Al final se ha optado por una solución que une las dos formas más habituales de tratar la concurrencia y paralelismo en Go: canales y gorutinas.

En cuanto al cuarto objetivo (Búsqueda de escalabilidad), hemos obtenido distintos valo-

res en las gráficas de rendimiento en el capítulo anterior. ????

El último objetivo primario de este proyecto (Plataformas de programación), hemos utilizado diferentes lenguajes de programación junto a algunas librerías que extienden su funcionalidad primaria. Entre estas librerías cabe destacar, entre otras, *Java Mina* y *Akka*, que proveen al lenguaje *Java* de otros paradigmas de concurrencia.

Para finalizar, es importante destacar que no se han podido completar casi todos los objetivos secundarios generales (no todos los servidores tienen acceso a base de datos, pero si se han utilizado otros paradigmas o implementado soluciones de tolerancia a fallos), como tampoco los objetivos específicos del juego (solo algunos servidores devuelven los objetos al empezar y permite interactuar con ellos).

## 6.2. Resultados

Si bien se han realizado diversas pruebas con una cantidad variable de “usuarios” conectados, los resultados obtenidos, en parte, no han sido los que se suponían en un primer momento.

- Es posible que el simple hecho de utilizar una máquina virtual ralentice la comunicación entre el servidor desplegado bajo la máquina huésped (en este caso, *Windows*) y los propios clientes de la máquina virtual (en este caso, *Debian 8*).
- Muchas aplicaciones funcionan mucho mejor cuando actúan de forma colaborativa bajo un sistema de múltiples nodos. No obstante, esto no siempre es una ventaja.

## 6.3. Trabajos futuros

Sin lugar a dudas, este trabajo es el proyecto más grande que he desarrollado hasta la fecha, con grandes cantidades de código, librerías y lenguajes de programación diferentes. No obstante, este proyecto se puede extender para experimentar con otras funcionalidades no desarrolladas al completo, como puede ser la implementación de un sistema aleatorio que genere el final de los combates a través de un timeout, sin necesidad de una aceptación de finalización por parte de los dos usuarios combatientes. A continuación se incluye una lista de posibles extensiones del proyecto:

- Se han desarrollado gran cantidad de lenguajes de programación, pero pocos meses después de finalizar toda la parte de programación, han surgido varios nuevos

lenguajes, los cuales se podrían probar para sacar nuevas métricas de rendimiento, como es el caso de Nim<sup>1</sup> o Elixir<sup>2</sup>.

- Migración de código a otras plataformas para ver el rendimiento fuera del ámbito *Windows*. Principalmente sería necesario hacer pruebas con Sistemas Operativos *Unix*.
- Pocas aplicaciones actuales disponen de un sistema basado en envíos de ACK's en las comunicaciones, ni mucho menos son tolerantes a los fallos. Estos fallos podrían intentar implementarse como posibles mejoras en los servidores implementados.
- Completar los servidores que no se pudieron finalizar (*C* y *F#*), para añadirlos a nuevas listas de métricas de rendimiento, además de arreglar los errores encontrados en el servidor en *Julia* y en *C#*.
- Como se ha comentado anteriormente, ningún servidor tiene completamente todas las funcionalidades secundarias de los clientes (algunos tienen parte, otros nada), ya que estas funcionalidades no eran fáciles de usar para hacer métricas, por lo que se tuvieron que prescindir. Entre todas las funcionalidades secundarias, la principal a probar sería el sistema basado en combates con dados, cuyo resultado es ofrecido a los usuarios tras pasar un tiempo aleatorio.
- Refactorizar todos los códigos en busca de mejoras, así como actualizar los códigos y las librerías utilizadas a las diferentes versiones actuales (la versión del cliente en *C++* es la versión 2.1, frente a la 2.3 actual).
- Ninguna aplicación real gestiona todos los datos de sus usuarios conectados en memoria principal. A pesar de que algunos servidores han llegado a implementar una conexión a una base de datos, sería altamente interesante investigar cómo conectarse desde el resto de lenguajes, realizando a su vez una pequeña comparativa frente a los diferentes drivers, así como una comparativa en busca de la mejor base de datos para nuestro juego.
- Todos los tests se han realizado sobre sistemas montados bajo un único nodo. Sería interesante poder realizar estos mismos tests sobre servidores multinodo o clusters de servidores.

---

<sup>1</sup><http://nim-lang.org/>

<sup>2</sup><http://elixir-lang.org/>

## 6.4. Reflexiones finales

Gracias a este proyecto he podido llegar a experimentar un alto crecimiento tanto a nivel técnico (análisis de problemas, arquitecturas cliente-servidor, diseño del software) como a nivel teórico (ampliar conocimientos sobre redes, TCP, paradigmas orientados a redes, ...).

Junto a estos nuevos conocimientos, hay que añadir la gran cantidad de libros, artículos, informes o foros para solucionar todos los problemas que se han podido encontrar durante el desarrollo, tanto desde las consultas a bases de datos desde NodeJS pasando por errores de conceptos o errores de las propias librerías.

Hay que agradecer el gran papel del tutor de este proyecto, Carlos E. Cuesta por facilitarme toda la ayuda posible, además de incentivarme para aprender y conocer nuevas tecnologías no tan habituales en el ámbito académico, así como dejarme vía libre para la experimentación. Debido a mi propio nivel de exigencia personal para desarrollar este trabajo, he intentado contar lo menos posible con su ayuda como método para llegar a ser capaz de valerme por mi mismo.

Para finalizar, es importante destacar que gran parte de la información sobre este proyecto se puede encontrar en la siguiente dirección: <http://maldicion069.github.io/tfg-gis/>. En ella se encuentra, de manera resumida, cada uno de los servidores implementados.

# Anexos

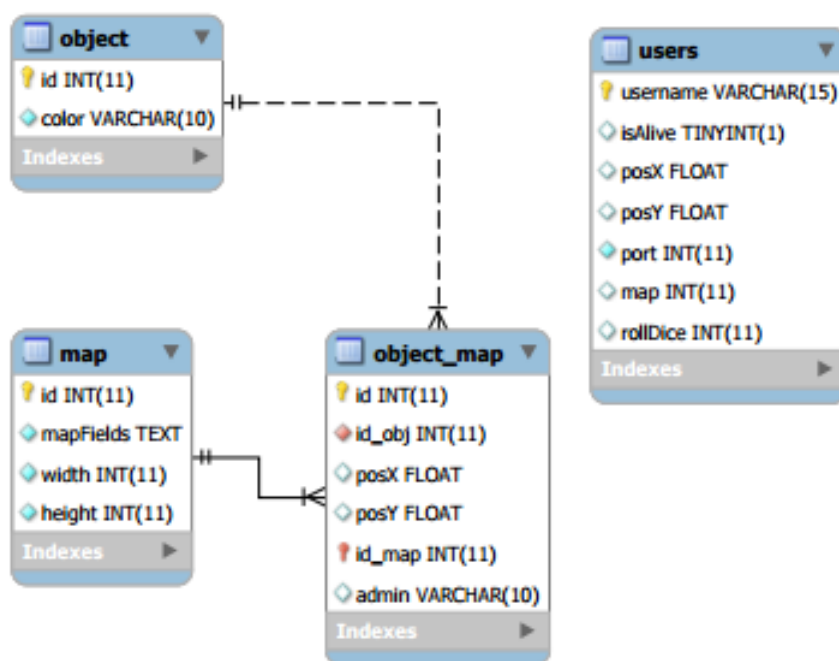
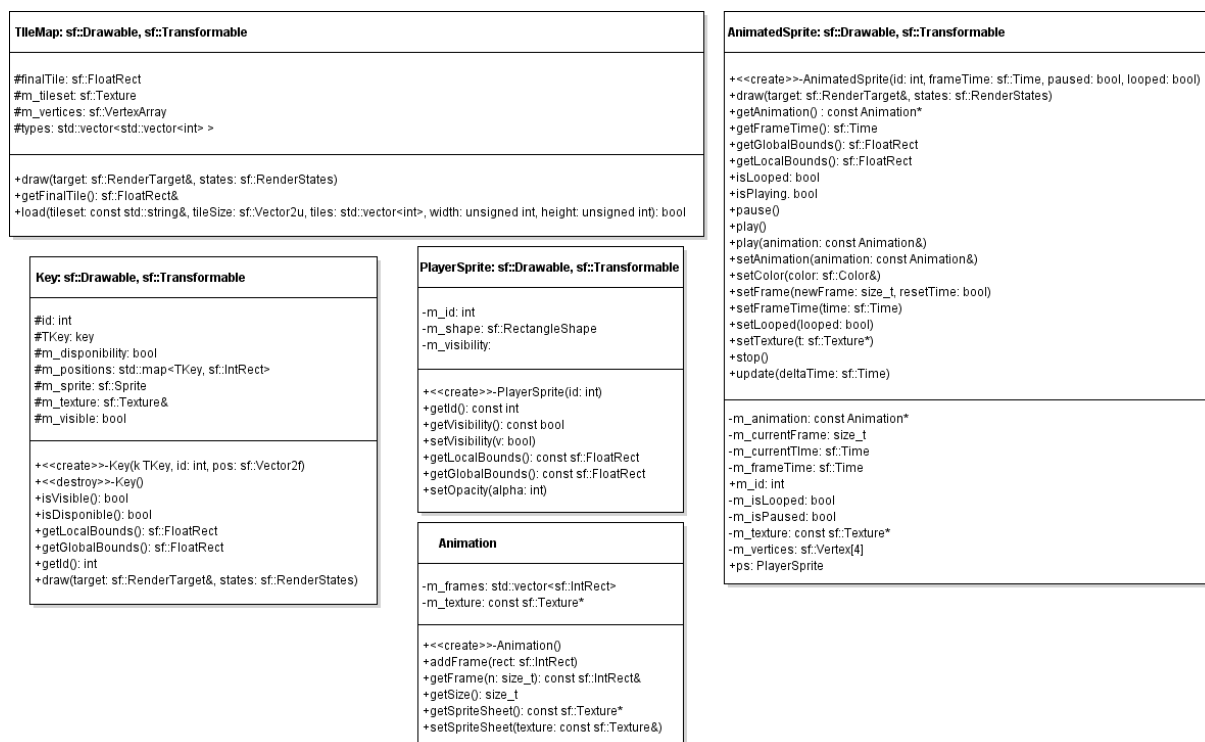


Figura 6.1: Esquema Entidad/Relación SQL

Lenguaje	1 Cliente	2 Clientes	4 Clientes	8 Clientes	16 Clientes	32 Clientes
NodeJS*	1113264	551509	406650.5	306666.25	67278.3125	43348.2813
	1113264	1103018	1626602	2453330	1076453	1387145
	997457.5	606070.5	321237.125	284586.75	68133.7813	32952.2969
IOJS*	997457.5	1212141	1284948.5	2276694	1090140.5	1054473.5
	667206	290625.75	138484.5	92637.375	63932.2813	75180.8281
	667206	581251.5	553938	741099	1022916.5	2405786.5
Go*	263269	271112.5	133945	74892.25	55438.0625	47763.5938
	263269	542225	535780	599138	8870009	1528435
	629981.5	247462.75	138906.25	67099.375	Dummy	Dummy
C#	629981.5	494925.5	555625	536795	Dummy	Dummy
	720895.5	273552.75	137734	70579.5	111748.875	505525.938
	720895.5	547105.5	550936	564636	1787982	16176830
D	1302875	724281.5	566723.25	573778.313	567131.813	527676.594
	1302875	1448563	2266893	4590226.5	9074109	16885651
	1581812.5	635151	529939.624	530975.563	643593.344	3049853.5
EventMachine*	1581812.5	1448563	2266893	4590226.5	9074109	16885651
Apache Mina*	535081	247466.75	241024.875	147255.688	41370.6878	86278.5156
	535081	494933.5	964099.5	1178045.5	661931	2760912.5
	595905	173835.75	94585.375	47222.25	38500.75	3326.8438
Scala Akka	595905	347671.5	378341.5	377778	616012	1063579
	692969.5	310874.5	179944.5	192879.5	355065.844	2037231.56
	692969.5	621749	719778	1543036	5681053.5	65191410
Groovy	593959.5	236344.25	125546.5	64301.8125	38034.1563	18751.0469
	593959.5	472688.5	502186	514414.5	608546.5	600033.5
	581540.5	231087	130458.25	65712.0625	48764.5625	89044.6406
Python Twisted	581540.5	462174	521833	525696.5	780233	2849428.5

Cuadro 6.1: Comparación Servidores contra N Clientes



**Figura 6.2:** UML clases principales del cliente gráfico (no incluidas estructuras)

Requests Started By	Description of Request	Requests Ended By
BeginAccept()	Aceptar nuevas conexiones.	EndAccept()
BeginConnect()	Conectarse a un host remoto.	EndConnect()
BeginReceive()	Recibir datos de un socket.	EndReceive()
BeginReceiveFrom()	Recibir datos de un host remoto.	EndReceiveFrom()
BeginSend()	Mandar datos a un socket.	EndSend()
BeginSendTo()	Mandar datos a un host remoto.	EndSendTo()

**Figura 6.3:** Acciones asíncronas en C# [Blu]





# Bibliografía

- [Ale] Andrei Alexandrescu. *The D Programming Language*.
- [Asy] Using Asynchronous Sockets. <http://codeidol.com/csharp/csharp-network/Asynchronous-Sockets/Using-Asynchronous-Sockets/>.
- [Bal] Ivo Balbaert. *Getting Started with Julia*.
- [Ber] Berkeley sockets. [https://en.wikipedia.org/wiki/Berkeley\\_sockets](https://en.wikipedia.org/wiki/Berkeley_sockets).
- [Blu] Richard Blum. *C# Network Programming*.
- [Com] Relative Speed: F# v C# v VB.NET. <https://jamesdixon.wordpress.com/2013/10/01/relative-speed-f-v-c-v-vb-net/>.
- [CSh] How to C# Chat Server. <http://csharp.net-informations.com/communications/csharp-chat-server.htm>.
- [Des] Desarrollo ágil de software. [https://es.wikipedia.org/wiki/Desarrollo\\_%C3%A1gil\\_de\\_software](https://es.wikipedia.org/wiki/Desarrollo_%C3%A1gil_de_software).
- [Dla] D Programming Language. [https://en.wikibooks.org/wiki/Category:D\\_\(The\\_Programming\\_Language\)](https://en.wikibooks.org/wiki/Category:D_(The_Programming_Language)).
- [Don] Alan A. A. Donovan. *The Go Programming Language (Addison-Wesley Professional Computing Series)*.
- [EMR] Building a socket server with EventMachine, Ruby, and Flash CS5. <http://www.giantflyingsaucer.com/blog/?p=1784>.
- [Gen] Manejar la concurrencia con Actores. <http://www.genbetadev.com/paradigmas-de-programacion/manejar-la-concurrencia-con-actores>.
- [GoC] Concurrency, Goroutines and GOMAXPROCS. <http://www.goinggo.net/2014/01/concurrency-goroutines-and-gomaxprocs.html>.

- [GoM] Go by Example: Mutexes. <https://gobyexample.com/mutexes>.
- [GPa] Gpars Documentation. <http://gpars.org/0.12/guide/guide/2.%20Getting%20Started.html>.
- [Gro] Curso de Groovy II (Diferencias con Java). <http://basallo.es/blog/2012/09/23/curso-de-groovy-ii-diferencias-con-java/>.
- [IBM] Sockets programming in ruby de ibm. <https://www6.software.ibm.com/developerworks/education/1-rubysocks/1-rubysocks-a4.pdf>.
- [Inta] Introducción a la programación asíncrona con Twisted. <http://www.genbetadev.com/frameworks/introduccion-a-la-programacion-asincrona-con-twisted>.
- [Intb] Introducing Akka - Simpler Scalability, Fault-Tolerance, Concurrency & Remoting through Actors. <http://jonasboner.com/2010/01/04/introducing-akka/>.
- [Java] Java 7 vs Groovy 2.1 Performance Comparison. <http://java.dzone.com/articles/java-7-vs-groovy-21>.
- [Javb] Java AkkaIO Documentation. <http://doc.akka.io/docs/akka/2.2.0/java/io-tcp.html>.
- [JRu] Going Dynamic on JVM: JRuby vs. Groovy. <http://www.sitepoint.com/going-dynamic-on-jvm-jruby-vs-groovy/>.
- [Lam] Using C++ Lambdas. <http://www.umich.edu/~eecs381/handouts/Lambda.pdf>.
- [Mul] A multiclient TCP server for Ruby. [http://apps.jcns.fz-juelich.de/doku/sc/tcp\\_server\\_ruby](http://apps.jcns.fz-juelich.de/doku/sc/tcp_server_ruby).
- [Noda] GameFromScratch. <http://www.gamefromscratch.com/post/2012/01/31/Network-programming-with-SFML-and-Nodejs-Part-1.aspx>.
- [Nodb] How does Nodejs compare to Scala and Akka. <http://www.quora.com/How-does-Node-js-compare-to-Scala-and-Akka>.
- [Nodec] Node.js v0.12.6 Manual & Documentation - Timers. <http://nodejs.org/api/timers.html>.

- [Pre] Se presenta públicamente el lenguaje de programación Ruby. <http://timerime.com/es/evento/768876/Se+presenta+pblicamente+el+lenguaje+de+programacin+Ruby/>.
- [Pro] Proceso Unificado. [https://es.wikipedia.org/wiki/Proceso\\_unificado](https://es.wikipedia.org/wiki/Proceso_unificado).
- [Quo] Scala vs Go - Could people help compare/contrast these on relative merits/demerits? <http://www.quora.com/Scala-vs-Go-Could-people-help-compare-contrast-these-on-relative-merits-demerits/>
- [REv] EventMachine: scalable non-blocking i/o in Ruby. <https://es.scribd.com/doc/28253878/EventMachine-scalable-non-blocking-i-o-in-ruby>.
- [Rob] Robust Networking in Multiplayer Games. <http://courses.cms.caltech.edu/cs145/2011/robustgames.pdf>.
- [Rub] Comparison of programming languages: Ruby, Groovy, Python and PHP. <http://blog.websitesframeworks.com/2013/11/comparison-of-programming-languages-ruby-groovy-python-and-php-353/>.
- [Scaa] Scala AkkaIO Documentation. <http://doc.akka.io/docs/akka/snapshot/scala/io-tcp.html>.
- [Scab] Scala and Go: A Comparison of Concurrency Features. <http://www.cs.colorado.edu/~kena/classes/5828/s12/presentation-materials/smithbrentgibsonleon.pdf>.
- [Scac] Scala vs. Node.js as a RESTful backend server. <http://developerblog.redhat.com/2015/04/22/scala-vs-node-js-as-a-restful-backend-server>.
- [Scad] The Difference Between Scala and Go. <http://boldradius.com/blog-post/U62N9C0AAC8AVoav/the-difference-between-scala-and-go>.
- [Scae] The Scala Actors Migration Guide. <http://docs.scala-lang.org/overviews/core/actors-migration-guide.html>.
- [SFM] Threads SFML. <http://www.sfml-dev.org/tutorials/2.0/system-thread.php>.
- [Thr] Thread Synchronization Mechanisms in Python. <http://effbot.org/zone/thread-synchronization.htm>.

- [Twi] Writing Servers. <https://twistedmatrix.com/documents/current/core/howto/servers.html>.