

# Maelström

Andrés Ortiz Corrales



# Índice general

<b>1. Introducción</b>	<b>5</b>
Videojuegos MMO . . . . .	6
Multijugador vs MMO . . . . .	6
<b>2. Objetivos y alcance</b>	<b>7</b>
Objetivos principales . . . . .	7
Objetivos secundarios . . . . .	8
<b>3. Planificación</b>	<b>9</b>
Principios ágiles . . . . .	9
Planificación temporal . . . . .	10
Cambios en planificación . . . . .	12
Seguimiento . . . . .	12
Documentación generada . . . . .	13
Desarrollo abierto . . . . .	14
<b>4. Análisis del problema</b>	<b>15</b>
Descripción inicial . . . . .	15
Antecedentes e influencias . . . . .	16
Mecánica de juego . . . . .	16
Diagramas de estado . . . . .	19
Especificación de requisitos . . . . .	21
Requisitos funcionales . . . . .	21
Requisitos no funcionales . . . . .	22
Descripción de casos de uso . . . . .	23
Descripción de historias de usuario . . . . .	25
Historias de usuario de baja prioridad . . . . .	30

Análisis de HU . . . . .	31
<b>5. Diseño</b>	<b>33</b>
Introducción . . . . .	33
Arquitectura . . . . .	34
Arquitectura del servidor . . . . .	34
Arquitectura del cliente . . . . .	41
Comunicaciones cliente-servidor . . . . .	41
Seguridad . . . . .	42
Diagramas de clase . . . . .	42
Modelo de datos . . . . .	44
Servicio de usuarios . . . . .	44
Servicio de juego . . . . .	45
<b>6. Implementación</b>	<b>46</b>
Estado actual del desarrollo . . . . .	46
Servidor . . . . .	46
Cliente . . . . .	48
Bots del juego . . . . .	49
Lenguajes y herramientas . . . . .	52
Servidor . . . . .	52
Cliente . . . . .	53
Comunicación . . . . .	53
Herramientas de desarrollo . . . . .	54
Pruebas . . . . .	55
Pruebas dinámicas . . . . .	56
Pruebas estáticas . . . . .	59
Despliegue . . . . .	61
<b>7. Conclusiones y trabajos futuros</b>	<b>63</b>
Problemas encontrados . . . . .	63
Conocimientos adquiridos . . . . .	64
Asignaturas relevantes . . . . .	65
Trabajos futuros . . . . .	65
Posible comercialización . . . . .	66
<b>8. Apéndices</b>	<b>68</b>
Glosario . . . . .	68

Licencias . . . . .	70
Preámbulo de la licencia . . . . .	71
Especificación de la API . . . . .	73
Identificación . . . . .	73
Maelström user . . . . .	74
Maelström world . . . . .	75
Códigos de estado . . . . .	80
Manual de usuario . . . . .	81
Introducción . . . . .	81
Mundo . . . . .	81
Jugadores . . . . .	83
Versiones . . . . .	85
Servicio de usuarios . . . . .	86
Servicio del mundo . . . . .	86
Servicio web . . . . .	87
Bot . . . . .	88
Diagrama de burndown . . . . .	90
<b>Bibliografía</b>	<b>91</b>

# Índice de figuras

4.1. Diagrama de estado de <i>barco</i> . . . . .	19
4.2. Diagrama de estado del <i>jugador</i> . . . . .	20
5.1. Arquitectura Cliente-Servidor . . . . .	33
5.2. Monolítico vs Microservicio[17] . . . . .	35
5.3. Arquitectura del servidor . . . . .	36
5.4. Arquitectura del servidor (prototipo) . . . . .	38
5.5. Arquitectura propuesta del servicio de juego . . . . .	40
5.6. Diagrama de clases del servicio del mundo . . . . .	43
5.7. Diagrama de clases del servicio de usuarios . . . . .	44
5.8. Diagrama del modelo de datos del servicio de juego . . . . .	45
6.1. Análisis de cobertura del servicio de usuarios . . . . .	58
8.1. Evolución del desarrollo del servicio del mundo . . . . .	87
8.2. Evolución del desarrollo de los servicios . . . . .	89
8.3. Diagrama de Burndown . . . . .	90

# Capítulo 1

## Introducción

Desde el comienzo del desarrollo y comercialización de los videojuegos ha existido el concepto de *Videojuego multijugador*, en referencia a aquellos juegos en los que múltiples jugadores (humanos) pueden jugar al mismo tiempo[1].

Las redes de ordenadores permitieron aumentar la capacidad de los videojuegos multijugador, pasando de múltiples mandos conectados a una misma máquina, a distintas máquinas conectadas, pudiendo aumentar el número de jugadores de 2~4 hasta 64 jugadores simultáneos.

Este proyecto, sin embargo, se orienta al estudio de los *Videojuegos Multijugador Masivo Online* (MMO), estos juegos buscan, como su nombre indica, permitir a cientos o miles de jugadores participar simultáneamente en una partida e interactuar entre ellos a través de una plataforma online[2].

Estos productos suponen, aún hoy en día, un reto tecnológico al necesitar gestionar tal cantidad de clientes simultáneos en un mundo persistente, debido a esto, resultan en proyectos de gran magnitud, con una base tecnológica que requiere de una enorme infraestructura y recursos. Esto resulta en la aparición de pocos juegos con estas características, con poca variedad, y generalmente con unos resultados insatisfactorios debido a la escasez de tecnologías y software desarrollado en este ámbito.

A lo largo de esta memoria, se profundizará en las dificultades técnicas, las necesidades de estos productos para ser considerados viables, las soluciones técnicas desarrolladas y las aplicaciones de estas.

## Videojuegos MMO

Un juego se podría considerar MMO cuando un gran número (indeterminado) de jugadores pueden jugar simultáneamente. Generalmente, además, esto implica la existencia de un **mundo persistente**, es decir, que las partidas seguirán desarrollándose estén o no conectados los jugadores, manteniendo sus datos entre distintas conexiones.

Aunque un juego podría ser de cualquier género, la mayoría de videojuegos se encuentran en uno de estos dos géneros:

- **MMORPG**: Juegos de *rol* o *acción* en los que los jugadores interactúan en tiempo real en un entorno 2D o 3D. Por ejemplo *World of Warcraft* (<http://eu.blizzard.com/games/wow>) o *Guild Wars* (<https://www.guildwars.com>) corresponden a este género
- **MMO-RTS**: Comúnmente juegos basados en navegador o móvil, con gráficos simples, en el que el jugador interactuará con los elementos del juego a tiempo real y deberá esperar un determinado tiempo a que se completen las tareas que asigne, ejemplos de este tipo son *Ogame* (<https://es.ogame.gameforge.com>) y *Clash of Clans* (<https://play.google.com/store/apps/details?id=com.supercell.clashofclans>)

El prototipo propuesto para este proyecto consistirá en un MMO-RTS.

## Multijugador vs MMO

Debido a la ambigüedad de la definición, gran cantidad de videojuegos reciben el calificativo *multijugador masivo* aunque su funcionamiento y mecánica básica no difiera de la de juegos multijugador estándar (partidas cortas con pocos jugadores), por ejemplo, juegos con integración social (chats, logros, mensajería) o juegos multijugador con diversas salas.

Este proyecto se centrará en soluciones MMO completas, que permitan una interacción real entre un número indeterminado de jugadores en una misma partida.

## Capítulo 2

# Objetivos y alcance

En este apartado desarrollamos los objetivos a completar durante el desarrollo de este proyecto, tanto los objetivos prioritarios como los requisitos secundarios que se plantea alcanzar.

### Objetivos principales

- Analizar y estudiar el estado actual de los videojuegos MMO, problemas y soluciones técnicas: La escasez de productos de estas características suponen un reto al tratar de analizar los aspectos técnicos de estos productos y encontrar soluciones.
- Desarrollar un prototipo de *videojuego multijugador masivo online de estrategia*: Aprovechando los datos obtenidos y las soluciones propuestas, poner en práctica el desarrollo de un prototipo funcional mínimo de un MMO-RTS que permita recoger datos, definir nuevos requisitos y analizar la viabilidad del producto. El prototipo constará mínimo de dos partes:
  - Servidor: servidor de juego con funcionalidad mínima y API
  - Cliente: Al menos un cliente funcional comunicado con el servidor



## Objetivos secundarios

Además de analizar y trabajar en este género de videojuegos, se busca generar una serie de avances técnicos en este ámbito, reflejados en los objetivos secundarios que se esperan alcanzar en el desarrollo del prototipo y que proporcionan un valor añadido:

- Aprovechamiento de tecnologías modernas: Por la complejidad técnica e incertidumbre en las nuevas tecnologías, la mayoría de productos MMO comerciales optan por tecnologías ya asentadas. En este proyecto se tratará de comprobar si las tecnologías aparecidas en los últimos años pueden resultar en una mejora significativa en el rendimiento y funcionamiento de estos juegos.
- Desarrollo libre del prototipo: Se espera realizar un desarrollo abierto del proyecto y de su documentación (Véase apéndice *Licencia*) proporcionando una solución completamente *open-source*[3] basada únicamente en tecnología libre.
- Analizar la viabilidad del desarrollo de un framework para facilitar la posterior implementación de productos basados en la misma solución software.
- Proponer una metodología de trabajo viable para el desarrollo completo del producto. Analizar dicha propuesta con el desarrollo realizado

# Capítulo 3

## Planificación

Para el desarrollo del prototipo. Se aplicarán los principios de metodologías de desarrollo ágil[4][5], en concreto, se han adaptado los principios y prácticas de la metodología **Extreme Programming**[6] (XP).

### Principios ágiles

Se aplicarán la mayoría de principios y técnicas de la metodología XP[7][8], obviando aquellos referentes al trabajo en equipo o que no posean relevancia en este proyecto:

#### Planificación

- Historias de usuario[9]: Además de los casos de uso, se incorporarán HU a la documentación y planificación del prototipo para realizar una estimación y como *roadmap*[10] inicial del proyecto.
- Desarrollo iterativo: El proyecto se dividirá en iteraciones, cada una de estas se planificará con una serie de objetivos y tareas a cumplir, y se procederá a un análisis en retrospectiva de dicha iteración.

## Diseño

- Mantener la funcionalidad al mínimo: Se busca un desarrollo rápido y un producto funcional temprano, aun con sólo las funcionalidades mínimas (**prototipado**).
- Desarrollo de *spikes*[11][12]: Se crearán productos mínimos (*spikes*) en conjunto con los prototipos para comprobar las posibles soluciones a los problemas de carácter técnico.
- Refactorización continua: Todo el código desarrollado será refactorizado iterativamente, procurando, no sólo ir añadiendo funcionalidad, sino actualizar y mejorar la calidad del código existente.

## Desarrollo

- Desarrollo de tests: Se crearán tests unitarios y de integración que deberán validar la funcionalidad de todo el código antes de ser puesto en producción.
- Repositorio común de código: Todo el software desarrollado será añadido a un repositorio común.
- Integración continua: Todo el código subido al repositorio será desplegado y testeado automáticamente con cada cambio realizado.
- Codificación estándar: Se siguen unos patrones y nomenclaturas de programación estandarizadas a lo largo del proyecto para evitar inconsistencias.

## Planificación temporal

La planificación temporal planteada inicialmente propone iteraciones mensuales, correspondiendo la iteración 0 al mes de **Octubre**, siendo **Julio** la novena iteración y la correspondiente a una versión estable.

Versión	Iteración	Descripción
0.0.x	0	Descripción del ámbito del proyecto y análisis inicial

Versión	Iteración	Descripción
0.1.x	1	Prototipo inicial de la arquitectura general del sistema, soluciones documentadas y prototipo del servicio de usuarios
0.2.x	2	Desarrollo del servicio de usuarios y prototipo del mundo
0.3.x	3	Prototipo <i>funcional</i> del sistema, desarrollo de primer prototipo de cliente
0.4.x	4	Análisis, prueba y corrección de errores, reinspección de los requisitos y desarrollo de tests, despliegue en web, versión <i>alpha</i>
0.5.x	5	Incorporación de Funcionalidades básicas restantes y características secundarias
0.6.x	6	Testeo, análisis de requisitos, desarrollo de tests, refactorización
0.7.x	7	Incorporación de características secundarias, corrección de errores, diseño de documentación final
0.x.y	8	Incorporación de características secundarias, corrección iterativa de errores, test del sistema completo
1.x.y	9	Actualización del sistema, documentación, requisitos. Refinamiento del cliente, optimización y despliegue final del sistema

Cada iteración además, considera el siguiente ciclo:

1. **Análisis:** Se analizará las historias de usuario a completar en la iteración actualizar
2. **Implementación:** Implementación de las HU prioritarias y funcionalidad básicas
3. **Testing:** Desarrollo de tests para comprobar la funcionalidad
4. **Refactorización:** Refactorización de código e Implementación de requisitos secundarios

5. **Análisis retrospectivo:** Análisis retrospectivo de la iteración y generación de documentación parcial

La planificación descrita corresponde a la segunda versión de esta, propuesta en *Septiembre* tras realizar el análisis del proyecto.

## Cambios en planificación

Debido a evolución del proyecto, la acumulación de retrasos y el trabajo irregular, se procedió a una serie de cambios en la planificación durante la iteración 5:

- Reducción del tiempo de las iteraciones: Se procedió a planificar iteraciones de entre 1 semana y medio mes
- Análisis del estado del proyecto, prioridad de las tareas y objetivos a cumplir
- División del proyecto y planificación en diversos sistemas: Se comenzó a planificar cada sistema y a desarrollarlo de forma independiente al resto.

Estos cambios, junto con algunos cambios en el desarrollo, mejoraron notablemente la evolución del proyecto, mejorando la velocidad de desarrollo y reduciendo el número de tareas de baja prioridad desarrolladas.

## Seguimiento

Aunque el desarrollo ha seguido la planificación descrita, para el seguimiento diario se hizo uso de las herramientas de desarrollo descritas en el capítulo 6, obteniendo diversas métricas útiles:

- **Commits:** Al haber usado un *sistema de control de versiones*, todos los cambios realizados son almacenados en forma de *commits*, permitiendo analizar los cambios en el código, llevar un historial y revertir cambios. Se han realizado más de **450 commits** en el repositorio a lo largo del desarrollo de este proyecto
- **Tests de Integración:** Para cada *commit*, la herramienta de integración continua realiza los test implementados, garantizando que el código

ya implementado y probado no se ha visto afectado por los nuevos cambios.

- **Tests de Cobertura:** Además, se analizará la cobertura de código de los tests con cada commit, para poder verificar que los tests siguen siendo válidos.
- **Branches:** Para mantener la estabilidad del proyecto, el desarrollo de los módulos principales se separará en dos o más versiones (*Ramas*) en el sistema de control de versiones:
  - *Master:* Es la rama principal del proyecto, corresponde a la versión estable
  - *Dev:* Corresponde al desarrollo actual del proyecto, es una rama inestable (cambios constantes), es necesario que todos los tests en esta rama sean correctos antes de llevar los cambios a *master* (proceso denominado **pull request**)
- **Issues:** Aunque las Historias de Usuario son la principal referencia en el desarrollo, se han generado *tareas* de menor granularidad para gestionar y priorizar el desarrollo a corto plazo, así como medio para reportar *bugs*, el proyecto completo consta de más de **250 issues**, con más de 200 completados.
- **Milestones:** Los issues se agrupan en *hitos*, representando las versiones estables que se han ido lanzando. Generalmente corresponden a una iteración y una versión nueva (más información en el apéndice Versiones). Aunque no poseen una estructura fija y son determinados en cada iteración para proporcionar un seguimiento flexible de acuerdo a los principios de las metodologías ágiles. El proyecto cuenta con más de 20 milestones, incluyendo milestones actualmente abiertos.

Este seguimiento, además, se realiza de forma independiente para cada *servicio* (Descritos en el Capítulo 5) como parte de los cambios de planificación realizados

## Documentación generada

Además de la documentación presentada, se fue generando documentación desechable para mejorar el seguimiento en cada iteración y proporcionar información general sobre el proyecto:

- **README.md:** Cada Servicio posee un archivo README en formato

*markdown* con una breve descripción del servicio en cuestión, una guía para instalarlo, desplegarlo y testarlo, así como información sobre el estado del servicio y licencia.

- **LICENSE:** Todos los Repositorios se encuentra con una copia de la Licencia *AGPL-3.0* (Véase Apéndice Licencia).
- **Manual de la API:** Todas las apis de los servicios se encuentran documentadas para facilitar su uso. El apéndice *Especificación de la API* recoge dicha documentación.
- **Manual de juego:** Con el fin de analizar y facilitar el uso del prototipo final, se redactó un manual de usuario (Apéndice Manual de Usuario).
- **Coverage report:** Los tests generan un análisis de cobertura automático para comprobar su validez.
- **Análisis de versiones:** Se proporcionará un índice de las versiones y algunas métricas para analizar la evolución del proyecto (Véase apéndice *Versiones*)

## Desarrollo abierto

De acuerdo a los objetivos del proyecto, todo el desarrollo será abierto, con todas las versiones, código y documentación publicadas bajo licencias de código abierto (Véase apéndice Licencias)

Se optará por la licencia **AGPL v3** pues posee las mismas características que una licencia GPL normal, en la que se obliga a una distribución libre de las obras derivadas, pero además incluye las obras en las que se haga uso del proyecto como servicio.

Para los recursos que no sean código (logotipo, documentos, etc...) se usará licencia **CC-by-sa**

Estas licencias se usarán en todo el proyecto salvo que se especifique lo contrario.

# Capítulo 4

## Análisis del problema

En este apartado realizaremos un análisis del problema a resolver y del prototipo a implementar

### Descripción inicial

Este proyecto desarrollará **Maelström**, un videojuego del género *multijugador masivo online de estrategia y simulador comercial* en el que los jugadores tratarán de aumentar un capital inicial comerciando entre diversas ciudades y entre ellos.

En este género de videojuegos, consideramos un *mundo* dinámico y persistente, continuamente activo en un servidor, accesible en cualquier momento por cualquiera de los jugadores. Aunque estos influyen en el mundo y en otros jugadores, tanto el estado del mundo como el de los propios jugadores cambia con independencia de que estén o no activos.

No es imprescindible por tanto la presencia de un jugador en el juego, sin embargo, este deberá conectarse a intervalos para realizar acciones con el objetivo de progresar. Generalmente estas acciones tardarán un lapso de tiempo real en cumplirse (minutos u horas), el jugador podrá seguir conectado o conectarse tras el paso de este tiempo para recibir los resultados de dichas acciones y comandar nuevas acciones.



Ejemplos de acciones serían:

- Construir un edificio
- Viajar de un sitio a otro
- Realizar una investigación o mejora
- Recolección de recursos

Además, la capacidad de interacción entre diversos usuarios permitirá la realización de acciones cooperativas o competitivas para alcanzar ventajas respecto otros jugadores (alianzas, guerras, . . . )

Para obtener más información acerca de la descripción del juego consultar el apéndice **Manual de Juego**

## Antecedentes e influencias

El proyecto Maelström guarda relación con dos géneros principales (MMO de estrategia y simulador económico), en concreto, recibe influencias de dos exitosos títulos:

- **Patrician III**[13]: Principal influencia en la mecánica de juego, Patrician III es un exitoso videojuego de simulación comercial para un jugador.
- **Ogame**[14]: Videojuego MMO de estrategia orientado a web, principal influencia en estética y patrones de interacción con los usuarios.

## Mecánica de juego

Comenzaremos describiendo las mecánicas básicas del videojuego[15] a desarrollar para aclarar el problema a resolver

- **Objetivo**

El objetivo de los jugadores es aumentar su **capital** (dinero y recursos) en un ambiente competitivo para superar al resto de jugadores, para ello podrá tomar estrategias cooperativas o competitivas con estos además de aprovechar las reglas básicas de juego.

## ■ Recursos

- *Dinero*: Recurso principal del juego, válido para intercambiar por otros recursos o comerciar con otros jugadores.
- *Productos*: Los productos poseen un valor dependiente de la ciudad y cantidad de productos en esta, los jugadores podrán comprar y vender productos.
- *Barcos*: Los jugadores podrán adquirir barcos, que les permitirá almacenar y transportar una cantidad limitada de productos entre ciudades. Cada barco tendrá una serie de atributos dependientes de su tipo
  - Nombre: dado por el jugador
  - Velocidad: velocidad a la que viaja el barco
  - Cargamento: Carga máxima de productos que puede almacenar
  - Precio: Coste de construir un nuevo recursos de este tipo

## ■ Roles

- *Jugador*: Jugador (humano) que poseerá una serie de recursos. Un jugador puede realizar una serie de acciones e interactuar con otros jugadores.
  - Construir barco: Intercambiará dinero por un recurso de tipo barco.
  - Comprar producto: Compra un producto de una ciudad a un barco.
  - Vender producto: Vende un producto de un barco a una ciudad.
  - Mover barco: Inicia el transporte de un barco de una ciudad a otra (acción que consumirá tiempo), el jugador podrá retornar el barco mientras esté en movimiento.
- *Bot*: Poseerá las mismas reglas que un jugador, sin embargo, tendrá un comportamiento automatizado que añadirá un factor aleatorio al juego con independencia de los jugadores reales activos, no podrá interactuar directamente con otros jugadores.
- *Ciudad*: Existen diversas ciudades en el mundo, en posiciones  $[x,y]$  cada una poseerá una serie de productos cuyos valores se actualizarán periódicamente. Los jugadores podrán interactuar con la ciudad. Cada producto de la ciudad posee unos parámetros:
  - Cantidad: Define la cantidad de un determinado producto en una ciudad (el máximo que un jugador podrá comprar), debe

- ser mayor o igual que 0.
- Producción: Define la variación de cantidad de un producto, puede ser positivo o negativo.
- Precio: Define el coste al que un jugador podrá comprar o vender un producto, depende del precio base de un producto, la cantidad y la producción en esa ciudad de dicho producto.

## ■ Reglas

- Los jugadores podrán comprar barcos si poseen el dinero suficiente.
  - El barco podrá ser creado en cualquier ciudad.
  - La construcción del barco requerirá tiempo.
- Los jugadores podrán mover sus barcos de una ciudad a cualquier otra, para ello el barco deberá estar en dicha ciudad y no en movimiento.
  - Una vez el barco esté en movimiento, se encontrará en dicho estado hasta que el barco alcance su destino o el jugador elija cancelar el viaje.
  - El tiempo que tarde el barco en alcanzar su destino dependerá de la velocidad de este y la distancia con la ciudad origen.
  - El viaje no podrá ser realizado a la ciudad origen
  - Si el viaje es cancelado, el barco tardará el mismo tiempo que el tiempo de viaje transcurrido en volver a la ciudad, no se podrá realizar ninguna acción adicional hasta que el barco alcance el puerto origen.
- Los jugadores podrán comprar productos de una ciudad si poseen el dinero suficiente, la ciudad posee cantidad suficiente y tienen al menos un barco en dicha ciudad con suficiente capacidad libre para almacenarlos
- Los jugadores podrán vender un producto si poseen dicha cantidad en un barco en la ciudad seleccionada
- Los jugadores no podrán comprar ni vender ni mirar precios de las ciudades en las que no haya al menos un barco amarrado
- Los jugadores podrán comprar y vender productos entre ellos si al menos hay dos barcos en la misma ciudad

## Diagramas de estado

Los siguientes diagramas definen el comportamiento de los barcos (Figura 4.1) y del jugador (Figura 4.2) representando los posibles estados y transiciones entre estos a partir de las reglas antes definidas.

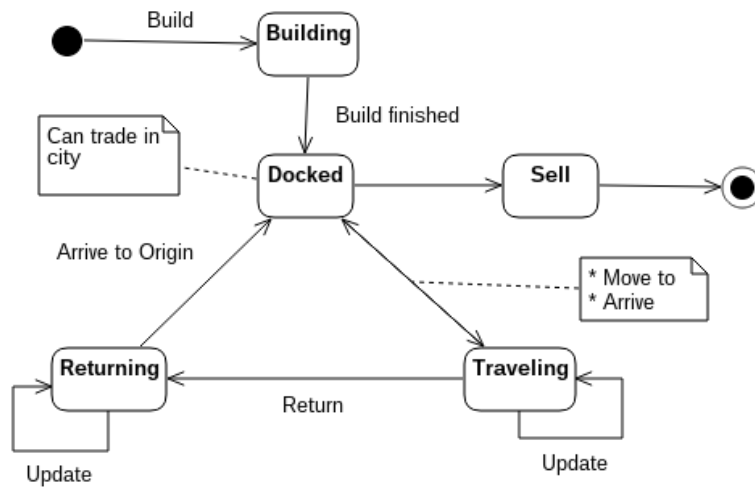


Figura 4.1: Diagrama de estado de *barco*

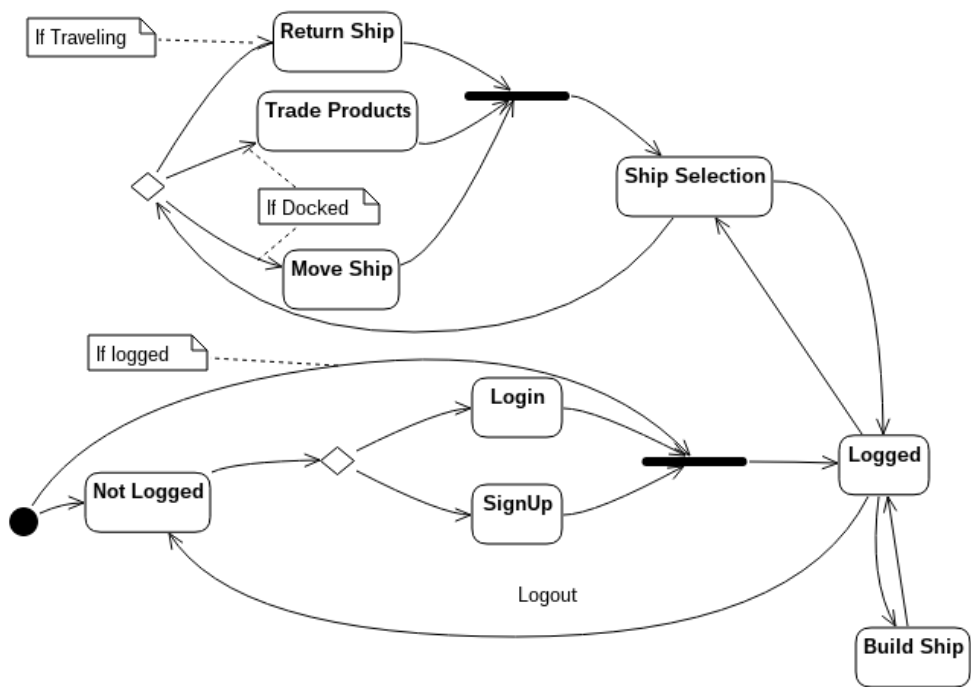


Figura 4.2: Diagrama de estado del *jugador*

## Especificación de requisitos

Teniendo en cuenta el carácter experimental del proyecto, los requisitos obtenidos se basan en el análisis de proyectos similares a los comentados anteriormente, así como una serie de necesidades añadidas al prototipo deseado, por tanto, estos componen una lista preliminar de las características de este.

### Requisitos funcionales

Aquí exponemos los requisitos principales que definen la funcionalidad del sistema:

1. **Gestión de usuarios (jugadores):** El sistema permitirá una gestión de usuarios básica (*login, signup, logout*)
2. **Acción de usuarios:** El sistema aceptará la realización de diversas acciones de los usuarios registrados desde un software cliente
  1. **Construir barco:** El jugador podrá añadir un barco al sistema
  2. **Comerciar:** El jugador podrá comprar y vender productos entre un barco y una ciudad
  3. **Mover barco:** El jugador podrá mover un barco de una ciudad a otra si este se encuentra *atracado*
  4. **Cancelar viaje:** El barco volverá al puerto origen
3. **Acceso a los datos de juego:** El jugador deberá ser capaz de acceder a los datos actuales del juego de acuerdo a lo que necesite (recibir información a petición del usuario)
  1. **Información de eventos:** El jugador deberá recibir información sobre eventos a tiempo real
4. **Actualización de datos de juego:** Los datos de juego se deberán actualizar periódicamente de acuerdo a las reglas del juego
  1. Los barcos en movimiento deberán actualizar su estado
  2. La cantidad de productos de las ciudades deberá actualizarse de acuerdo a su producción y consumo
5. **Cliente web:** Un cliente web para acceder a todos los sistemas del juego

## Requisitos secundarios

1. **Servicio de mensajería entre jugadores:** Los jugadores podrán mandarse mensajes entre ellos
2. **Bots automáticos:** Bots automáticos que sean capaces de interactuar con el sistema de forma autónoma de forma realista
3. **Cliente móvil:** Prototipo de cliente móvil

## Requisitos no funcionales

Aquí exponemos como requisitos no funcionales aquellas limitaciones impuestas por el tipo de problema a tratar y objetivos a conseguir en el prototipo

1. El sistema será accesible por clientes con independencia de la plataforma (por ejemplo móvil o web).
2. Algunos servicios del sistema requerirán identificación por parte de los usuarios registrados
  - La información de los usuarios será almacenada de forma segura mediante una contraseña encriptada.
3. La interacción deberá ser rápida y tener poca latencia. Se espera una reacción del sistema inferior a 500 ms para conseguir una interacción fluida. Minimizar la latencia es prioritario.
4. El sistema deberá ser **escalable** con el aumento de clientes. Se debe permitir más de 100 jugadores simultáneos, sin límite en el número de jugadores futuros.
5. El sistema deberá tener **tests** unitarios, de integración y de cobertura asociados:
  - Todo el sistema se encontrará en una plataforma de **integración continua** durante el desarrollo.
  - Los sistemas principales deberán poseer una **cobertura** de al menos un 70 %.
6. Toda la tecnología aplicada deberá ser **open-source**, el prototipo igualmente deberá ser licenciado bajo una licencia libre.
7. El sistema deberá ser **portable**, pudiendo ejecutar instancias del servidor con independencia de la infraestructura.
8. El sistema debe evitar trampas por parte de los usuarios, garantizando que la información no puede ser vista ni modificada sin permiso.

## Descripción de casos de uso

Se exponen los casos de uso principales del sistema, correspondientes a las acciones del usuario con el sistema, definiendo el comportamiento normal y alternativo de cada acción. (Véase Diagramas de estado para más información sobre las posibles acciones de los usuarios en el juego).

### Sign Up

- Descripción: El usuario se registrará con sus datos en el sistema (nombre de usuario, email y contraseña).
- Precondición: El usuario no se encuentra registrado en el sistema.
- Postcondición: El usuario será registrado en el sistema con sus datos y se le asignará una id única.

Jugador	Sistema
1. Cliente inicia acción de registro	2a. Registra al usuario en el sistema 3. Crea nuevo jugador

*Curso alterno:*

- 2b: Si el usuario se encuentra registrado o los datos son incorrectos no se continúa y se notifica del error al usuario.

### Login

- Descripción: El usuario iniciará una sesión en el sistema.
- Precondición: El usuario se encuentra registrado en el sistema.

Jugador	Sistema
1. Cliente inicia sesión	2a. Genera token de sesión y lo envía al usuario

*Curso alterno:*

- 2b: Si el usuario no se encuentra registrado o los datos son incorrectos no se continúa y se notifica del error al usuario.



### Consulta de datos de juego

- Descripción: El cliente consulta datos del juego.
- Precondición: Usuario registrado si los datos son privados.

Jugador	Sistema
1. Cliente inicia acción de consulta	
	2a. El sistema devuelve los datos
3. Cliente actualiza los datos	

*Curso alterno:*

- 2b: Si la acción requerida es incorrecta se notificará al cliente.
- 2c: Si los datos están restringidos y el usuario no se encuentra identificado no se devolverán datos
- 2d: En caso de error, se notificará al usuario.

### Acción del juego

- Descripción: El cliente realiza una acción en el juego (consultar *Diagramas de estado*).
- Precondición
  - Usuario registrado e identificado
  - Estado interno acorde a la acción requerida
- Postcondición: Estado interno del juego actualizado con los resultados de dicha acción.

Jugador	Sistema
1. Cliente envía petición	
	2a. Comprueba el estado interno del juego
	3a. Actualiza el estado interno
	4. El sistema notifica al cliente
5. Cliente actualiza su estado	

*Curso alterno:*

- 2b: Si la acción requerida es incorrecta o el usuario no se encuentra identificado se notificará al cliente y cancelará el curso de la acción.

- 2c: Si la acción es inválida, se cancelará y se notificará al cliente.
- 3b: Si ocurre algún error, se revertirán los cambios para volver al estado anterior, se notificará al cliente y se registrará la incidencia en un log.

### Actualización de datos

- Descripción: El cliente se registra en el servidor para obtener unos datos a tiempo real.
- Precondición: Usuario registrado.

Jugador	Sistema
1. Cliente inicia subscripción	2a. El sistema registra al cliente 3. El sistema envía datos actualizados
4. Cliente actualiza los datos	5a. Se repite 3. cuando los datos se actualizan

*Curso alterno:*

- 2b: Si no es posible establecer conexión, se rechaza la subscripción.
- 5b: Si los datos no se actualizan, espera a una actualización
- 5c: Si el cliente se desconecta, no se envían más datos

## Descripción de historias de usuario

El desarrollo de este prototipo contempla la siguiente serie de Historias de usuario (HU) a completar, agrupadas en 11 funcionalidades principales. Cada HU se dividirá en tareas con unos **Puntos de Historia** asignados[9].

Cada punto de historia (**PH**) equivale a 1 hora de trabajo aproximada de un desarrollador *junior* más tests, refactorización y solución de errores. Calcularemos un 25 % extras para el desarrollo de tests y arreglo de errores más un margen de 10 %, resultando en 1.35 horas aproximadas por PH.

1. Sistema de autenticación de usuarios
  1. Los usuarios se podrán registrar en el sistema con un **nombre de usuario, email y contraseña**

- Base de datos: 4 PH
  - API CRUD: 3 PH
- 2. Los usuarios podrán usar su nombre de usuario o email y su contraseña para iniciar sesión.
  - Base de datos: 2 PH
  - API: 2 PH
- 3. La identificación de usuarios será persistente entre sesiones.
  - Sistema de sesiones: 8 PH
  - Actualización de la API: 2 PH
- 4. Las contraseñas se almacenarán encriptadas
  - Seguridad de contraseñas: 4 PH
- 5. El registro de usuarios será independiente de la plataforma usada (cliente web, móvil)
  - Actualización de la API: 1 PH
  - Activación de protocolos CORS: 1 PH
- 6. Los usuarios podrán modificar y eliminar sus datos.
  - API: 1 PH
  - Actualización de la funcionalidad: 2 PH
- 7. Cada usuario poseerá una **id** única.
  - API: 1 PH
  - Sincronización de las Bases de datos: 4 PH
- 2. Ciudades
  1. Cada ciudad poseerá un **nombre** único y una lista de **productos**.
    - Base de datos: 6 PH
    - API: 4 PH
  2. Cada ciudad se encontrará en una **posición** determinada (coordenadas [x,y])
    - Base de datos: 2 PH
    - Cálculo de posiciones: 2 PH
    - API: 1 PH
  3. La ciudad poseerá una **cantidad** determinada de cada producto (mayor de 0), que se podrá modificar.
    - Base de datos: 8 PH
    - API: 3 PH
  4. La ciudad tendrá un valor de **producción/consumo** constante de cada producto, que modificará su cantidad periódicamente
    - Base de datos: 2 PH
    - API: 2 PH

- Actualización de los datos: 10 PH
- 3. Jugadores
  1. Los jugadores se podrán identificar mediante una id de usuario (Relacionado con HU 1.2)
    - Base de datos: 2 PH
    - API: 1 PH
  2. Los jugadores poseerán una cantidad de **dinero** y una lista de **barcos**
    - Base de datos: 3 PH
    - API: 1 PH
    - Comunicación al cliente: 3 PH
  3. Los jugadores sólo podrán realizar acciones en el mundo del juego si se encuentran identificados
    - Identificación mediante token: 3 PH
- 4. Barcos del usuario
  1. Cada barco tendrá unas características básicas de acuerdo a su modelo de barco (HU 6.2)
    - Base de datos: 4 PH
    - Funcionalidad básica: 2 PH
    - API: 4 PH
  2. Cada barco tendrá, además, un **nombre** único para el usuario (un mismo usuario no puede tener 2 barcos con el mismo nombre) y una lista de **productos** (cada producto asociado a una **cantidad**).
    - Base de datos: 8 PH
    - Actualización de la funcionalidad: 3 PH
    - API: 2 PH
  3. La cantidad total de productos no podrá superar el valor de **carga máximo** del barco.
    - Actualización de la base de datos: 1 PH
    - Funcionalidad y validación de datos: 3 PH
  4. Cada barco se encontrará asociado a una **ciudad**.
    - Base de datos: 2 PH
  5. Un barco tendrá un **estado** asociado (amarrado, viajando) de acuerdo al diagrama de estado antes descrito con las reglas descritas.
    - Funcionalidad del objeto: 6 PH
    - Actualización de la base de datos: 2 PH
  6. Un barco podrá moverse entre las distintas ciudades si se encuentra

amarrado. el tiempo del viaje dependerá de la posición de las ciudades. El **estado** del barco deberá actualizarse debidamente.

- Funcionalidad básica del barco 2 PH
- Actualización de los datos en tiempo real 2 PH

5. Modelos de barcos

1. Cada modelo de barco poseerá un **nombre** único.
  - Base de datos: 1 PH
2. El modelo asociará al barco unas características definidas (**vida, velocidad, carga máxima**).
  - Base de datos: 2 PH
  - API: 3 PH
3. El modelo de barco tendrá un precio asociado.
  - Base de datos y API asociada: 2 PH

6. Implementación de compra/venta de productos

1. Un jugador podrá comprar productos a una ciudad desde un barco, comprobando que el jugador posee dinero suficiente, hay suficiente cantidad de dicho producto en la ciudad, el barco se encuentra amarrado en la ciudad y hay espacio de carga suficiente en el barco
  - API: 1 PH
  - Funcionalidad básica de compra: 6 PH
  - Actualización de la BBDD: 5 PH
  - Validación de datos: 4 PH
2. Un jugador podrá vender productos de un barco a una ciudad, validando que hay suficiente cantidad de dicho producto en el barco y el barco se encuentra amarrado en la ciudad.
  - API: 1 PH
  - Funcionalidad básica: 4 PH
  - Actualización de la BBDD: 5 PH
  - Validación de datos: 2 PH

7. Construcción de barcos

1. El jugador crea un nuevo barco, validando que posee suficiente dinero (*precio* del modelo)
  - Validación de datos: 2 PH
  - Actualización de base de datos: 2 PH
  - Funcionalidad básica: 5 PH
2. El barco será creado en una ciudad
  - Base de datos y funcionalidad: 2 PH

8. Implementación de interfaz web

1. La API (HTTP) será accesible para la consulta de los datos del juego y realización de acciones desde cualquier cliente válido.
  - API CRUD básica: 5 PH
  - Especificación de las peticiones: 4 PH
  - Actualización de seguridad (CORS): 2 PH
  - Especificación de las respuestas: 6 PH
2. La API permitirá una comunicación bidireccional (*websockets*) entre cliente y servidor.
  - Especificación de la API: 5 PH
  - Gestión de eventos y mensajes: 5 PH
3. Todas las operaciones de la API deberán ser aceptadas y sus datos validados
  - Identificación de las operaciones: 7 PH
  - Identificación de los datos: 6 PH
9. Implementación de cliente web
  1. El cliente web permitirá acceder a la funcionalidad del juego desde una interfaz
    - Interfaz básica del usuario: 5 PH
    - Proveedor de cliente: 4 PH
    - Comunicación con la API: 4 PH
    - Identificación de acciones: 6 PH
    - Sincronización de los datos: 8 PH
  2. Los usuarios podrán ver los barcos, acceder a los datos de cada uno y realizar acciones sobre estos
    - Interfaz de lista de barcos: 5 PH
    - Datos del barco: 4 PH
    - Acciones del barco: 6 PH
  3. El usuario podrá ver un mapa y seleccionar ciudades para ver los datos o para seleccionar un destino
    - Interfaz del mapa: 8 PH
    - Acciones del mapa: 4 PH
  4. La interfaz del cliente permitirá recibir notificaciones y realizar acciones visuales sobre el juego, así como recibir datos de otros jugadores
    - Notificaciones: 3 PH
    - Diseño de la web: 6 PH
    - Datos de otros jugadores: 8 PH
  5. Los clientes podrán identificarse y registrarse desde el cliente web

- Interfaz de registro: 4 PH
- Interfaz de Identificación: 4 PH

## Historias de usuario de baja prioridad

1. Sistema de mensajería
  1. El sistema permitirá el envío de mensajes entre jugadores
    - Servidor: 4 PH
    - Funcionalidad básica: 3 PH
    - API: 2 PH
  2. Cada mensaje consta de un **autor**, **destinatarios**, **cabecera** y **cuerpo**
    - Base de datos: 5 PH
    - Validación de datos: 3 PH
2. Bot de juego
  1. Agentes reactivos capaces de acceder al sistema y jugar con el resto de jugadores mediante una IA básica.
    - Implementación básica del bot: 2 PH
    - Login al sistema: 2 PH
    - Interacción con el sistema: 6 PH
    - Sincronización de datos: 4 PH
  2. Las acciones de los agentes deberá ir acorde a las reglas de cualquier jugador, y su medio de acceso será similar al de resto de jugadores
    - Heurística del juego: 8 PH
    - Validación de datos: 3 PH

## Análisis de HU

### Historias de Usuario de Alta Prioridad

Funcionalidad	HUs	PH	Tests	Horas totales
1	7	35	9	47
2	4	40	10	54
3	3	13	3	18
4	6	42	10	57
5	3	8	2	11
6	2	28	7	38
7	2	11	3	15
8	3	40	10	54
9	5	79	20	107
Total	35	296	74	$296 \cdot 1.35 = 400$

### Historias de Usuario de Baja Prioridad

Funcionalidad	HUs	PH	Tests	Horas totales
1	2	17	4	23
2	2	25	6	34
Total	4	42	10	$42 \cdot 1.35 = 57$

- **Historias de Usuario totales:** 39
- **PH totales:** 338
- **PH por HU media:** 8.6 PH
- **Horas Totales:**  $338 \text{PH} \cdot 1.35 = 456$  horas aproximadas

Las Historias de Usuario planteadas y las tareas generadas a partir de ellas se usarán como guía del desarrollo del proyecto, el alto número PH resultante obligará a priorizar las funcionalidades principales pues se estima un número de horas de trabajo superior al tiempo disponible.

En concreto, se estiman 8 meses de desarrollo (**Noviembre a Junio**), lo que supone una carga media de 57 PH por mes (aunque se prevé un progreso no



homogéneo, con mayor avance en los últimos meses). Si consideramos una carga de 8 horas semanales los primeros 4 meses y un aumento a 14 horas en los 4 siguientes, obtenemos un tiempo disponible de 320 horas para el desarrollo (aproximadamente un 70 % de los estimado).

Para suplir esta falta de tiempo, se aplicarán los cambios en la planificación antes comentados, se simplificará en algunos aspectos el prototipo y se relajarán los tests a desarrollar (reduciendo el tiempo de implementación de estos). En caso de tener tiempo suficiente, se desarrollarán las funcionalidades secundarias y todos los tests. Con estas medidas, se espera obtener el prototipo requerido en el plazo disponible.

A esta estimación hay que añadirle aproximadamente 40~50 horas para redactar debidamente la documentación y gestión del proyecto, lo que supone una carga extra aproximada de 6 horas al mes fuera de la estimación de desarrollo.

# Capítulo 5

## Diseño

### Introducción

El problema a tratar, como recordamos, se trata de un videojuego multi-jugador. Por las necesidades de persistencia y escalabilidad será imposible realizar una arquitectura P2P o mediante conexión directa entre los usuarios, solución común entre estos productos, por tanto, será necesario desarrollar una **aplicación web**, con una arquitectura similar a una **cliente-servidor** (Figura 5.1).

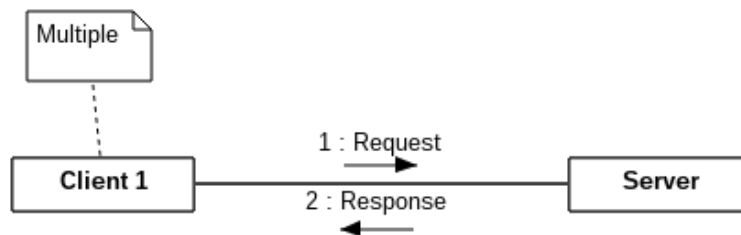


Figura 5.1: Arquitectura Cliente-Servidor

Sin embargo, las necesidades particulares de latencia y la relativa complejidad del sistema dificulta la implementación sobre un sistema cliente-servidor común. En concreto, necesitaremos una comunicación bidireccional entre

cliente y servidor, así como un mecanismo para actualizar periódicamente los datos del servidor.

## Arquitectura

Partiendo de la base cliente-servidor existente en cualquier aplicación web, adaptaremos cada parte de la arquitectura en base a las necesidades específicas del problema.

### Arquitectura del servidor

Dado el coste de las operaciones a realizar en un videojuego, es imprescindible tener en cuenta la distribución carga entre diversos servidores (*game servers*)[16]. Considerando, además, la complejidad del sistema optaremos por una **arquitectura orientada a servicios** (SOA).

Una SOA, divide el sistema completo en una serie de sistemas independientes (servicios) con bajo acoplamiento, de forma que un usuario u otro servicio pueda hacer uso de estos mediante una interfaz.

En una primera aproximación, se partió de un modelo de **microservicios**(Figura 5.2), en el que una aplicación es dividida en diversos servicios de forma transparente al cliente[17][18]. Un desarrollo orientado a esta arquitectura favorece diversos factores:

- Modularidad: Una relativa independencia entre los servicios favorece un bajo acoplamiento del sistema, haciéndolo más fácil de desarrollar y mantener.
- Escalabilidad: Cada servicio resuelve una parte del problema de forma independiente, dividiendo el sistema en varias máquinas (escalabilidad horizontal), además, cada servicio puede ser desplegado y escalado con independencia del resto (escalabilidad vertical).
- Extensibilidad: Es posible añadir, eliminar o modificar cada microservicio con un impacto mínimo en el resto

- Especialización de los servicios: Cada servicio desarrolla una tarea concreta, haciendo uso de las herramientas mínimas necesarias para realizarla, con independencia de las herramientas usadas en otros servicios

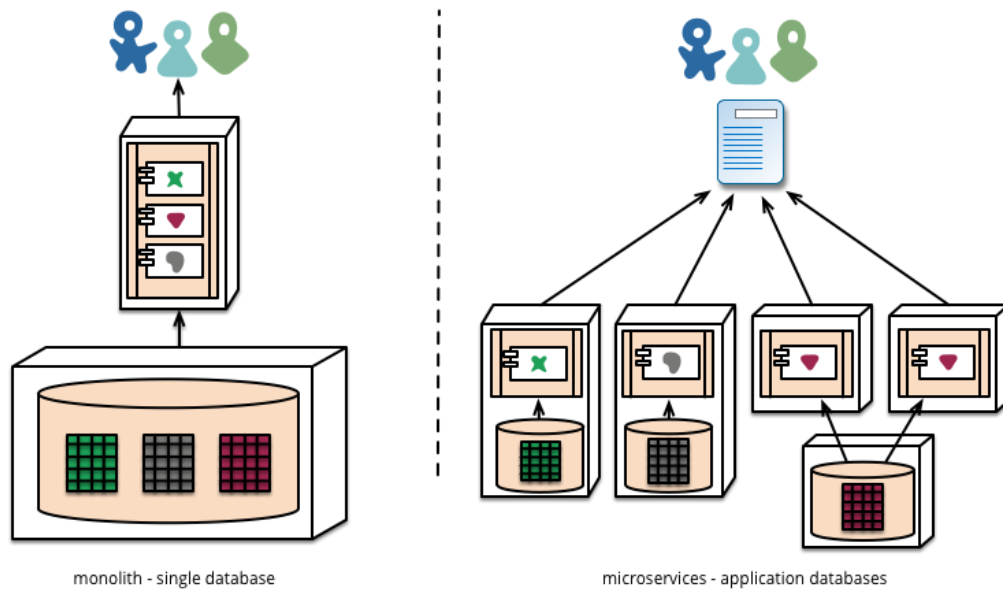


Figura 5.2: Monolítico vs Microservicio[17]

Esta arquitectura, sin embargo, puede tener un impacto negativo en la **latencia** al realizar peticiones entre servicios. Por ello, en el sistema a desarrollar se va a tratar de eliminar servicios intermediarios, a costa de reducir la transparencia de la arquitectura al cliente, eliminando interfaces generales sobre los microservicios.

Cada microservicio poseerá su propia instancia de servidor, base de datos y podrá desplegarse y funcionar de forma independiente al resto. Se comunicarán entre ellos mediante un *middleware* si fuera necesario.

Los distintos servidores (microservicios) se dividirán en dos zonas diferenciadas, aquellos que actúan directamente con el cliente (**front-end**) y aquellas que actúan únicamente con otros servicios ocultos al cliente (**back-end**), obtenemos así una arquitectura similar a una **DMZ**(Demilitarized Zone) (Figura 5.3).

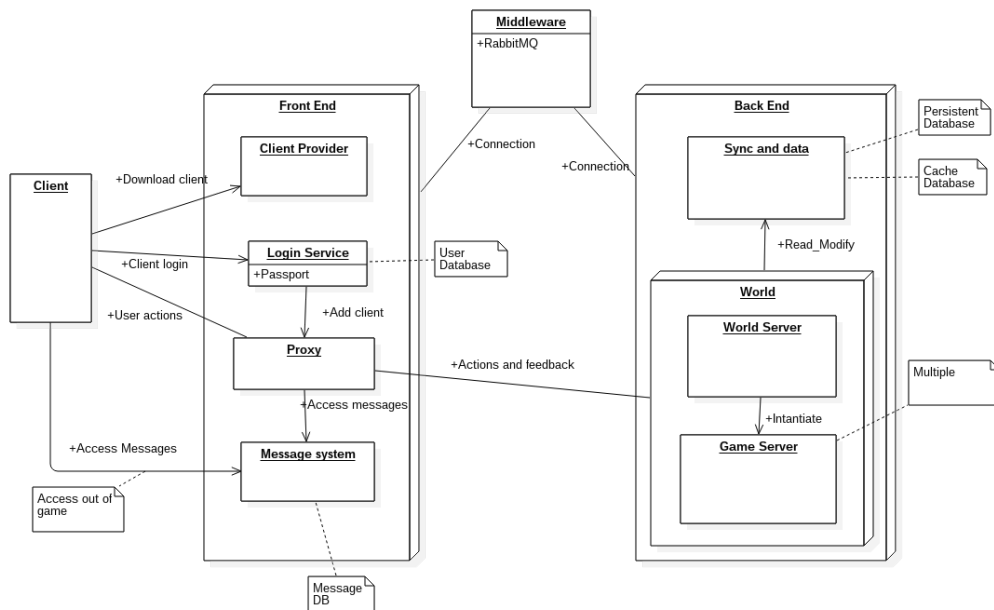


Figura 5.3: Arquitectura del servidor

## Servicios

La alta extensibilidad de esta arquitectura propicia futuros cambios en los servicios implementados, por tanto es previsible una variación en la lista de servicios a implementar, podemos dividirlos en servicios **front-end** y **back-end** dependiendo de si interactúan directamente con los clientes o no así como servicios **middleware** que proveerán la infraestructura necesaria para la comunicación entre los distintos servicios:

### Front-End

Con el término front-end nos referimos a la parte “visible” del sistema, con el que realizará conexión directa desde el cliente:

- **Servicio de Login y autenticación:** Se encargará de controlar la entrada de usuarios y su autenticación mediante distintos procedimientos (contraseña, redes sociales...) y el almacenamiento seguro de los usuarios en su base de datos.

- **Proveedor de cliente web:** Proporciona el cliente web y los recursos al usuario con el que conectarse al sistema. No poseerá ninguna lógica del sistema ni almacenamiento de datos
- **Interfaz de sockets/proxy:** Conexión directa con los clientes y enrutamiento a los servicios back-end apropiados, incorpora la principal capa de seguridad del sistema y almacenamiento en cache.
- **Servicio de mensajería:** Proporciona un servicio independiente de mensajería y notificaciones en el sistema, tanto entre usuarios como sistema-usuario.

## Back-End

Nos referimos a los servicios *ocultos* al cliente, comunicados únicamente con el resto de servicios del sistema:

- **Servidor de datos y sincronización:** Gestiona los datos del juego, tanto permanentes como dinámicos, gestionando el caché necesario y los back-ups a la base de datos.
- **Servidor de mundo:** Actualización y gestión del mundo y lógica del juego
  - **Servidores de juego:** En caso de un posible aumento en la envergadura prevista del servidor del mundo, se plantea la opción de crear además una serie de servidores de comunicados con el servidor de mundo que sincronizará con el servicio de datos y sincronización.

## Middleware

La comunicación interna entre servidores se realizará mediante un sistema de cola de mensajes ligeros por su menor carga de red y mayor velocidad respecto a peticiones HTTP.

Aunque más pesada, algunos servicios pueden implementar una comunicación HTTP mediante sistemas de una API REST, bien sustituyendo o complementando el sistema AMQP, principalmente en aquellos casos en los que

el servicio proporcionado pueda realizar tareas puntuales tanto con otros servicios del sistema como con clientes/servicios externos.

Este middleware se encontrará en un servicio externo, a través del cual se comunicarán las capas de *front-end* y *back-end*.

## Arquitectura del prototipo

Siguiendo la arquitectura anteriormente planteada, para el desarrollo del prototipo se procedió a la simplificación de los servicios desarrollados para obtener un producto funcional lo antes posible de acuerdo a la planificación propuesta anteriormente (Figura 5.4). Esta decisión no impide, posteriormente, aumentar el número de servicios dividiendo los existentes (una práctica muy común en el desarrollo orientado a microservicios).

En esta simplificación, se optó unificar los servicios del mundo en un único servicio y eliminar el proxy.

Además, de acuerdo al planteamiento inicial de la arquitectura, se optó por eliminar el *middleware*, reduciendo las comunicaciones entre servicios y usando directamente las API REST

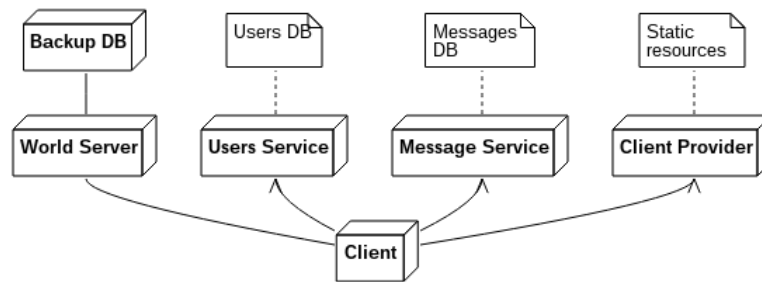


Figura 5.4: Arquitectura del servidor (prototipo)

## Propuesta de trabajo futuro en el servicio de juego

Como parte de la arquitectura en servicios, se propone como evolución en el desarrollo implementar el servidor de juego como una arquitectura de

microservicios compatible con el resto de servicios.

Esta propuesta trata de solventar el problema de escalar el sistema, siendo este servicio el principal cuello de botella del sistema. Al ser un sistema con gran cantidad de accesos y actualizaciones a la base de datos, una simple duplicación del servicio o una BD distribuida no permitirá escalarlo con al mismo nivel que el resto de servicios.

Por ello, se propone, adaptando la arquitectura clásica de los MMO, distribuir los datos del juego en múltiples instancias (concretamente, en ciudades), con una base de datos independiente. El servicio tendría un servidor principal, encargado de instanciar estos servicios y como punto de acceso (Figura 5.5). A diferencia del resto de arquitecturas MMO, los clientes, en lugar de acceder a un único servidor, podrán acceder directamente a múltiples servidores de juego (o ciudades) a la vez, permitiendo intercambiar datos y trabajar con la totalidad del sistema. Los propios servidores (u otros servidores especializados) podrán dedicarse a manejar los datos comunes entre los servidores de juego.

Con esta arquitectura, se prevé conseguir los mismos beneficios en escalabilidad que en el resto del sistema sobre los servicios principales, a la vez de reducir la latencia y el consumo de recursos en la actualización de los datos de juego (respecto a los recursos totales). También proporcionará una capa más de seguridad y fiabilidad sobre los propios servidores de juego y sus datos.

Esta arquitectura es producto de la división de la arquitectura actual, diseñada con el fin de facilitar este proceso. Sin embargo, esta arquitectura dificultaría el despliegue del sistema y no es adecuada para un prototipo.



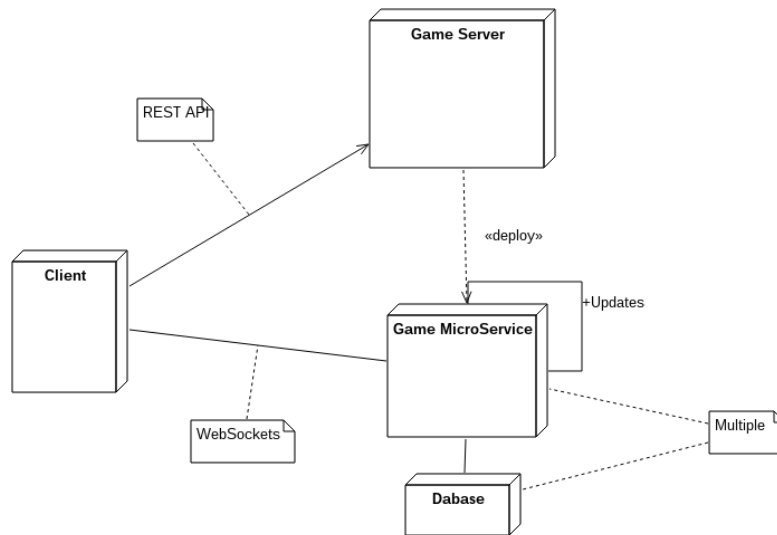


Figura 5.5: Arquitectura propuesta del servicio de juego

### Comentarios sobre la arquitectura del servidor

Se ha elegido esta arquitectura de microservicios en lugar de la llamada “*arquitectura monolítica*” de un servidor para proporcionar mayor escalabilidad y modularidad al sistema, una mejor integración con las tecnologías elegidas y con la metodología a desarrollar.

La gran cantidad de funcionalidades distintas a implementar y el uso de herramientas relativamente modernas y con poca experiencia en su aplicación es decisivo a la hora de la elección de esta arquitectura, pues será posible realizar cada funcionalidad de una forma sencilla y con las herramientas mejor preparadas para ella, además, permitirá incorporar otras tecnologías ya conocidas en donde sea posible. Mejorando enormemente la mantenibilidad del proyecto, extensibilidad futura y escalabilidad.

El uso de esta arquitectura, sin embargo, dificulta en gran medida el despliegue del servidor, aumenta la complejidad de este en conjunto, y puede afectar negativamente al rendimiento en conjunto.

## Arquitectura del cliente

El cliente actuará simplemente como una interfaz entre el usuario final y el sistema, por tanto, no resultará en un sistema complejo. La gran variedad de clientes distintos que es posible desarrollar hace imposible definir una arquitectura o diseño concretos, sin embargo, todos actuarán de acuerdo a un diseño común de interfaz y unas comunicaciones HTTP y websocket comunes (Véase Comunicaciones cliente-servidor), se definen a priori tres prototipos de clientes:

- **Cliente Web:** Ejecutado desde el servicio proveedor de cliente web, se compondrá de una página web dinámica, comunicada mediante peticiones HTTP (*AJAX*) y websockets con los distintos servicios.
- **Cliente escritorio:** Se compondrá de una interfaz completa (independiente del sistema o navegador web) que actuará de forma similar al cliente web, aunque eliminando algunas de sus limitaciones en cuestiones de rendimiento gráfico o almacenamiento permanente de datos.
- **Cliente móvil (Android):** Aunque actúe de forma similar a los otros, el cliente móvil presentará una mejor optimización en rendimiento y una interfaz adaptada a las capacidades de un móvil moderno, además, se propone una mejor integración con el sistema (notificaciones, sonidos etc...) propias de una aplicación de este tipo.

## Comunicaciones cliente-servidor

Debido a la variedad de clientes, y la posible incorporación de nuevos servicios, la comunicación entre los distintos servidores front-end y clientes se realizará mediante protocolos estándar. Concretamente se realizarán comunicaciones de dos formas:

- **Websocket:** Se usarán para comunicación bidireccional ligera entre cliente y servidor para acciones y actualización de datos en tiempo real. Se implementarán mediante la librería *Socket.io*[19]
- **API-REST:** Algunos servicios implementarán una API-REST a la que un usuario o otro servicio se podrá conectar mediante peticiones HTTP estándar para hacer uso exclusivamente de dicho servicio, dentro o fuera de las del juego.

- **JWT**: El cliente almacenará un token con la estructura de un *JSON Web Token*[20] donde se almacenará su id de usuario, lo que garantizará su acceso a los distintos servicios con su id única. Este token se podrá usar tanto con websockets como API-REST. (Véase Apéndice 3: Especificación de la API)

## Seguridad

Aunque la seguridad no es la prioridad del prototipo, cualquier servicio online debe poseer una capa de seguridad. Como resultado de la arquitectura diseñada, los puntos más vulnerables del sistema son los servicios comunicados con el cliente.

Los sistemas no comunicados directamente con clientes abordarán su seguridad mediante la implementación de *cortafuegos* basados en *whitelist* permitiendo únicamente comunicarse con los otros servicios.

El sistema de usuarios posee todos los datos sensibles de los usuarios, por tanto, su acceso a estos se realizarán mediante usuarios autenticados. La contraseña de los usuarios se almacenará encriptada mediante un *hash*.

La comunicación será mediante conexiones HTTPS y WSS durante la autenticación de usuarios, así como en cualquier acceso que requiera el envío del token *JWT* para la identificación en los servicios.

Una prioridad en este tipo de juegos es evitar las trampas de los jugadores, aún con clientes no oficiales. Por ellos, todos los datos del juego son gestionados y comprobados únicamente por el servidor impidiendo realizar acciones ilegales.

## Diagramas de clase

A continuación, se exponen los diagramas de clases UML[21] de los módulos principales del sistema. Estos diagramas representan la estructura de objetos de los servicios de **usuarios** (Figura 5.6) y del sistema del **mundo** (Figura 5.7). La arquitectura de objetos desarrollada finalmente en *JavaScript* no es exacta a los diagramas aquí descritos pues dicho lenguaje no implementa *clases* sino *prototipos*[22].

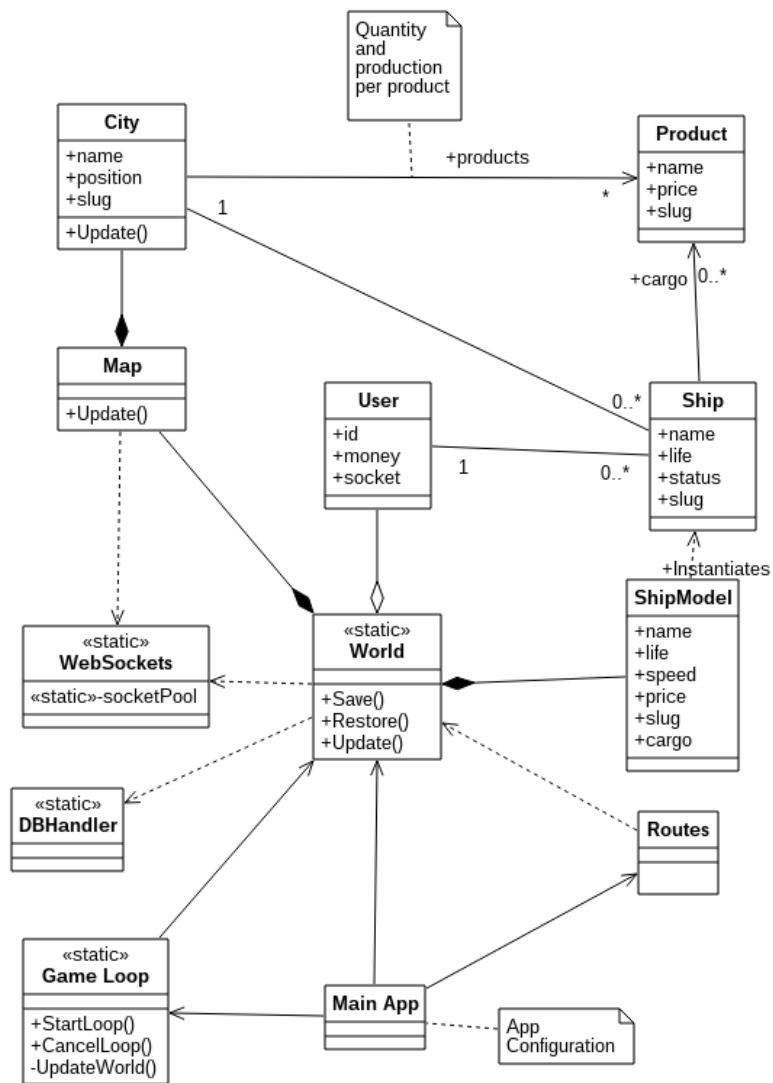


Figura 5.6: Diagrama de clases del servicio del mundo

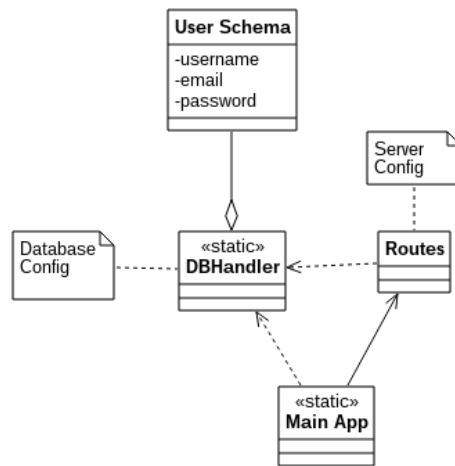


Figura 5.7: Diagrama de clases del servicio de usuarios

## Modelo de datos

Cada servicio poseerá su propio modelo de datos. Los servicios no compartirán ningún dato entre ellos, con la excepción de la **id de usuario**, de forma que se garantice una id común entre todos los servicios para un usuario concreto.

## Servicio de usuarios

El servicio de usuarios posee un modelo de datos sencillo, únicamente con la entidad **usuarios** con los siguientes atributos:

- Id: Id única del usuario común a todos los servicios.
- Nombre: Nombre único de usuario
- Email: Correo electrónico único de usuario
- Contraseña: Hash+salt[23] de la contraseña del usuario

Todos los usuarios tienen una id,nombre y email únicos

## Servicio de juego

El modelo de datos del servicio de juego (Figura 5.8) consiste en los datos necesarios para modelar la lógica del juego (Capítulo 3: Diagramas de estado). Las entidades coinciden con los *roles* y *recursos* anteriormente definidos:

- **Usuario:** Define los datos del usuario en el juego, con una id coincidente con la del *servicio de usuarios*
- **Producto:** Id y datos básicos de cada Producto
- **Ciudad:** Datos de cada ciudad, relacionados con los productos de dicha ciudad
- **Barco:** Datos del barco, relacionados con el usuario propietario y los productos que posee
  - **Modelo del barco:** Datos del tipo de barco, relacionados con cada barco de dicho tipo

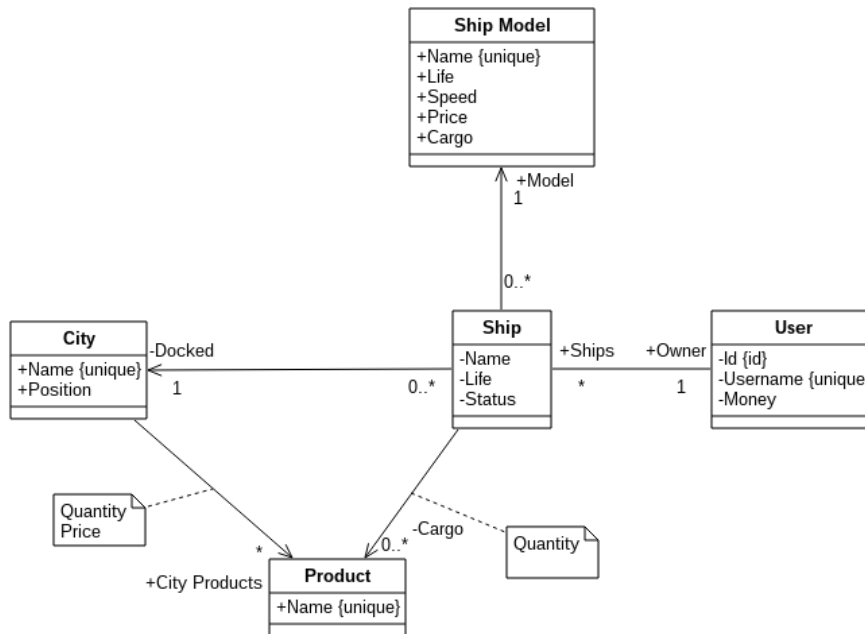


Figura 5.8: Diagrama del modelo de datos del servicio de juego

# Capítulo 6

## Implementación

La primera etapa de la implementación consistió en un extenso estudio de las herramientas apropiadas para el desarrollo e implementación del prototipo. La arquitectura diseñada permite el uso de tecnologías específicas para cada componente del proyecto, sin embargo, es necesario además tener en cuenta las tecnologías que permitan implementar dicha infraestructura.

Debido a la arquitectura orientada a servicios, la implementación de cada módulo se ha realizado independiente del resto, pudiendo realizar un mejor seguimiento de cada módulo.

### Estado actual del desarrollo

Siguiendo la planificación y arquitectura comentadas, el sistema actual consta de tres servicios principales desarrollados además del bot y el cliente de juego.

#### Servidor

El lado servidor consta en la actualidad con 3 servicios, cada uno implementado como un servidor independiente.

## Usuarios

- Repositorio: <https://github.com/demiurgosoft/maelstrom-users>
- Versión: 1.0.0
- Estado: Estable, en mantenimiento

El sistema de usuarios gestiona los datos de usuario así como los procesos de alta, baja e identificación de usuarios. El sistema almacena un *hash* de las contraseñas mediante el algoritmo *bcrypt* basado en el cifrado *Blowfish*.

El acceso al sistema se realiza mediante una *REST API*. El sistema proporcionará al cliente un *token* basado en la tecnología *JSON Web Token* a modo de identificación en el sistema, permitiendo el acceso de un usuario identificado al resto de servicios del sistema.

La arquitectura modular y el bajo acoplamiento con el resto de servicios permiten una completa reutilización del servicio con cualquier arquitectura similar.

## Mundo

- Repositorio: <https://github.com/demiurgosoft/maelstrom-world>
- Versión: 0.8.0
- Estado: En Pruebas

El sistema de mundo posee la lógica del juego principal, así como los datos del juego. Es posible acceder a los datos y realizar acciones mediante la API. Además, una implementación paralela de comunicación mediante *websockets* permite la actualización a tiempo real de algunos datos del juego.

Los datos de cada jugador solo serán accesibles por este. El sistema funciona completamente por su API, con independencia del cliente usado. El propio servidor del mundo implementa el bucle principal de juego, que actualizará los datos a intervalos de 1 segundo.

También este servicio implementa la cache de los datos para reducir la latencia y la comunicación con la base de datos redundante para almacenar los datos del juego en caso de fallo en el servidor.

Este sistema es con diferencia el más complejo y de mayor tamaño. Se



prevé dividir el sistema en diversos servicios mediante una arquitectura de *microservicios* en posteriores iteraciones (Capítulo 5).

## Servidor web

- Repositorio: <https://github.com/demiurgosoft/maelstrom-web>
- Versión: 0.6.2
- Estado: En Desarrollo y pruebas

El servidor web proporciona el cliente web, así como todos los recursos del juego. No contiene base de datos ni implementa ninguna lógica. Su función es proporcionar los recursos para reducir la carga del resto de recursos.

Este servidor actualmente sólo proporciona el cliente web, pero podría usarse como punto de actualización al resto de clientes, compartiendo los recursos comunes entre todos los clientes. También es usado para proporcionar un punto de acceso al sistema único, de forma que el acceso a toda la arquitectura de servicios subyacente sea transparente al usuario final.

## Cliente

El sistema permite el uso de cualquier tipo de cliente con diversas tecnologías, para el prototipo se desarrolla un *cliente web* y un *bot* que actúa como cliente automatizado.

El sistema no prevé ninguna lógica en el cliente, y todos los datos son actualizados desde los servidores. Sin embargo, el sistema cliente puede contener cierto control sobre las acciones del usuario para evitar peticiones inválidas al servidor, reduciendo las peticiones totales y mejorando la experiencia del usuario.

## Cliente web

El cliente web consiste en una página web desarrollada con Html 5, CSS y JavaScript. Además de una serie de frameworks (Bootstrap, React, EJS, JQuery,...) con el objetivo de concentrar toda la funcionalidad de una página

única, evitando recargas de página innecesarias reduciendo la latencia en las acciones del jugador.

Al ser un cliente que no debe recargar páginas, para actualiza su estado se hace uso de peticiones *AJAX* a la API del servidor y *websockets* para recibir datos a tiempo real. El resultado es un cliente que no sobrecarga el servicio de juego al no necesitar ni recursos ni realizar peticiones *AJAX* innecesarias proporcionando una latencia mínima para un servicio web.

## Bots del juego

- Repositorio: <https://github.com/demiurgosoft/maelstrom-bot>
- Versión: 0.1.1
- Estado: En Desarrollo y pruebas

Maelström implementa bots a modo de clientes del sistema, haciendo uso de sistema bajo las mismas reglas que cualquier cliente. El servidor no diferencia entre humanos y bots.

El objetivo de estos bots es proporcionar unos agentes de juego fuera del control humano y probar el sistema bajo condiciones de uso realistas.

Un bot de juego consiste en un agente automático que realizará acciones con el juego principal haciendo uso de la API de este. Toda la información será recibida a través de esta misma API. Por tanto, toda la información del bot y su rango de acciones se encuentran en el mismo plano que las de los jugadores normales.

El bot será capaz de actuar de acuerdo a las reglas del juego de una forma razonable para intentar ganar, aunque **no** óptima, pues el objetivo es emular un jugador humano, por lo que se ha añadido una cierta aleatoriedad a sus acciones, aunque siempre dentro de las operaciones que le proporcionarán ventaja en el juego.

Por simplicidad en la implementación, se ha implementado con **JavaScript** y **Node.js** acorde con el resto del proyecto. El bot implementa además su propio cliente para acceder a la API del juego, por lo que no necesitará el *cliente web* de Maelström.

El bot es capaz de consultar y actualizar todos los datos de juego relevantes,

es decir, los datos “públicos” del juego (ciudades y modelos de barco) así como sus datos de jugador (barcos, productos y dinero). Igualmente, es capaz de identificarse automáticamente en el sistema con el usuario indicado en su fichero de configuración. No posee la capacidad de crear nuevos usuarios para evitar un uso indebido del bot.

```
{  
    username: "marvin",  
    password: "dontpanic42",  
    email: "marvin@bot.com"  
}
```

#### *Configuración del usuario para el bot*

El bot se comunica únicamente mediante peticiones HTTP a la API de los servicios a intervalos regulares (por defecto *3 segundos*) para actualizar sus datos internos y realizar acciones. En caso de un error en la conexión el bot intentará actualizar su estado interno y dejará de realizar acciones hasta conseguirlo.

El bot además, tratará de realizar únicamente acciones válidas, aunque no garantizará que lo sean, pues es el servidor el que validará estas acciones (al igual que con el resto de jugadores).

El servidor no diferenciará entre bots y jugadores humanos, por tanto, es posible entrar al sistema con los datos de un bot y realizar acciones con dicha cuenta. Igualmente, un bot puede tomar el relevo a un jugador humano en cualquier momento. Los datos del bot se actualizarán siempre con los datos actuales del servidor, permitiendo al bot jugar en cualquier momento con cualquier usuario, incluso a la vez. También es posible usar múltiples bots distintos a la vez.

## **Heurística**

El bot consiste en un *agente reactivo* que actuará de acuerdo al estado que reciba del servidor. El agente actualizará su estado, comprobará las acciones posibles a realizar y las ejecutará a intervalos regulares.

El bot puede realizar las siguientes acciones en caso de cumplir las condiciones:

- **Login:** Identificarse en el sistema con el usuario proporcionado al comienzo de la ejecución.
- **Actualizar mapa y modelos de barco:** El agente pedirá la información general del juego al servidor.
- **Actualizar barcos del jugador y dinero:** El agente puede pedir información al servidor del su estado como jugador si se encuentra identificado en el sistema.
- **Construir barco:** El agente construirá un barco de un **modelo aleatorio** válido (con un precio menor al dinero que posee el jugador) si:
  - El agente tiene más dinero que lo que cuesta el barco más barato.
  - El agente no tiene ningún barco o tiene 3 veces más de dinero que el barco más barato.
- **Comprar productos:** El agente comprará 10 unidades de cada producto de una ciudad si:
  - El barco se encuentra amarrado.
  - Hay más de 20 unidades de dicho producto y la producción es positiva en la ciudad.
  - El agente **no** tiene en cuenta ninguna limitación en las reglas del juego para la compra de productos (simplemente el servidor no las ejecutará).
  - El agente **no** comprará ni más ni menos de 10 unidades de un producto determinado de golpe, para favorecer la compra de diversos productos en lugar de uno solo.
- **Vender productos:** El agente venderá 10 unidades de cada producto desde un barco a una ciudad si:
  - El barco se encuentra amarrado.
  - Hay menos de 2 unidades de dicho producto en la ciudad y la producción en negativa.
  - Hay al menos 10 unidades de dicho producto en el barco.
- **Mover barco:** El agente moverá un barco **después** de vender y comprar productos a una ciudad aleatoria (distinta a la actual) si el barco se encuentra amarrado.

Las acciones de comprar y vender productos se realizan para cada barco amarrado y cada producto. Mover barco se realizará después para cada barco amarrado. Para los barcos en movimiento no se realiza ninguna acción.

## Lenguajes y herramientas

Las siguientes tecnologías se eligieron como principales tecnologías para el desarrollo del sistema.

### Servidor

- **Node.js:**[24][25] Entorno de ejecución JavaScript orientado al desarrollo de servidores web.
  - **Alternativas:** Se planteó el uso de **Python** como principal alternativa, sin embargo, se optó por Node.js al usar un lenguaje conocido (*JavaScript*), su eficiencia y su orientación a servicios de tiempo real. Se tuvo en consideración Python+Django para el servicio proveedor de web, al ser una herramienta adecuada, pero al tener experiencia con Node.js se optó por realizar todos los servicios con dicho lenguaje.
- **Express:** Principal framework para el desarrollo de servidores en Node.js
- **MongoDB:** Por velocidad e integración con *JavaScript* se optó por MongoDB como principal base de datos del sistema, aunque esto no limita el uso de otras bases de datos en ciertos servicios
  - **Mongoose:** Herramienta de modelado de objetos para MongoDB, proporcionando una capa extra de abstracción e integridad de la base de datos mongo
  - **MySQL:** Opción planteada para aquellos servicios que requieran una base de datos relacional
- **RabbitMQ:** Framework del protocolo de colas de mensajes *AMQP*, planteado para las comunicaciones entre servicios, permitiendo diversos patrones (balanceo de carga, publish-subscribe,...) de comunicación eficiente y fiable.
- **Redis:** Almacenamiento estructurado de datos en memoria, útil como cache en memoria de la base de datos para reducir latencia.
  - **Alternativas:** **LokiJS** proporciona un mecanismo simple de base de datos en memoria con backup a un archivo **JSON**, sin embargo, no parece una opción apropiada para un sistema complejo que pueda requerir escalar.

## Cliente

- **Html 5:** Se aprovecharán las especificaciones de Html 5 para desarrollar el *cliente web* principal
- **JavaScript:** Como es habitual, la lógica del cliente web se desarrollará sobre JavaScript
  - **JQuery:** Se usará JQuery como framework de JavaScript para el cliente
  - **AJAX:** Se usará AJAX para realizar peticiones HTTP asíncronas.
- **EJS:** Lenguaje de plantillas basado en Html y JavaScript. Usado para generar el código Html en el servidor
- **React:** Se aprovecharán los componentes de React para generar código dinámico y reutilizable en el cliente.
  - **JSX:** Se hace uso de la extensión JSX de JavaScript para generar el código de React.
    - **Babel:** Se usará el intérprete babel para generar código js a partir del código JSX de React mediante Browserify.
- **Bootstrap:** Framework de Html, CSS y js para crear páginas *responsive* adaptadas a móvil.
- **Otras Alternativas:** Al ser un servicio sin un cliente definido, es posible desarrollar clientes en diversas plataformas, en concreto se plantearon otras tecnologías como clientes:
  - **Android:** Una aplicación móvil nativa (desarrollada con **Java**) y conectada por HTTP y Websockets.
  - **Unity:** Motor de desarrollo de videojuegos, permitiría el desarrollo de una aplicación cliente tanto para móvil como para escritorio.

## Comunicación

- **Api Rest:** A partir del protocolo HTTP estándar, se crearán Apis *Restful* para conectar el cliente con el servidor mediante **AJAX**, **supertest** o cualquier cliente HTTP.
- **Socket.io:** Framework para el uso de **Websockets** que proporciona una capa de abstracción sobre esta tecnología, permitiendo una comunicación bidireccional entre clientes y servidor a tiempo real
  - **Alternativas:** Se eligen Websockets sobre **WebRTC** o **UDP** al ser un protocolo orientado a la comunicación fiable en tiempo real

de mensajes planos.

- **Json:** Se usará Json como estructura de datos para mensajes planos entre cliente y servidor, al ser relativamente eficiente y tener fácil integración con JavaScript tanto en cliente como servidor).
  - **Alternativas: XML** ofrece una estructura más compleja, a costa de mayor tamaño en los mensajes y necesitar código *parser* extra tanto en cliente como servidor.
- **JWT:** Se usarán los *Json Web Tokens* como token entre cliente y servidor, que garantice la identidad del primero al hacer uso de los servicios[20].
- **CORS:** Cross-Origin Resource Sharing, el uso de servicios independientes requerirá una configuración en el sistema que permita la conexión con distintos dominios de origen[26].

## Herramientas de desarrollo

Para ayudar al desarrollo de proyecto se eligió un conjunto de herramientas. Estas herramientas favorecen un desarrollo ágil, abierto, y de acuerdo a la metodología especificada en el Capítulo 4. Aunque se hayan seleccionado estas herramientas, el desarrollo no se encontrará limitado únicamente a estas.

- **GitHub:**[27] Repositorio basado en **Git**[28][29] donde se alojará el código y se realizará un seguimiento del desarrollo usando sus herramientas:
  - **Issues & Milestones:** Se aprovecharán los issues y milestones de GitHub para llevar a cabo la planificación a corto y medio plazo del proyecto mediante una metodología basada en *tareas*
  - **GitHub Pages:** La información extra del proyecto se creará en una página alojada en GitHub para proporcionar información actualizada al público.
  - **Alternativas:** GitLab, repositorio basado en git similar a GitHub, aunque menos conocido. Se elige GitHub por ser un *estándar de facto* en proyectos libres.
- **Travis CI:**[30] Servicio de integración continua conectado con GitHub. Automatiza los procesos de integración y testeo del código.
  - *Alternativas:* Jenkins.
- **Npm:** Gestor de paquetes de Node.js, además de servir de herramienta principal de despliegue.

- **Atom:** Editor libre de texto usado preferentemente para el desarrollo del código aunque no exclusivamente
  - Se ha optado por el uso de un entorno simple y libre, en lugar de un IDE completo que pueda limitar la portabilidad del proyecto.
- **Markdown:** Lenguaje de marcado usado para generar la documentación y recursos de texto.
- **Browserify:** Herramienta para generar paquetes de código JavaScript para cliente.
- **Js-Beautify:** Herramienta para indentación automática del código en JavaScript, garantizando un estilo limpio y coherente.

## Herramientas de tests

Las herramientas a continuación fueron usadas para el proceso de testeo del software (Capítulo 7) con el objetivo de garantizar su calidad y funcionalidad.

- **Mocha:**[31] Módulo para desarrollo de tests unitarios y de integración para Node.js.
  - **Chai:** Módulo de aserciones usado junto con mocha para los tests.
  - **Supertest:** Módulo de tests de peticiones HTTP, usado para probar la API
- **Istanbul:** Módulo para realización de tests de cobertura.
- **Coveralls:** Servicio automático de análisis de cobertura, con integración con GitHub y Travis. Este servicio permite un análisis continuo de la cobertura de los tests, como parte de la integración continua del software.
- **JsHint:** Herramienta para detección de errores y análisis de la calidad del código en JavaScript.
- **BitHound:** Servicio online para comprobar calidad de código y análisis de dependencias de un repositorio.
  - **Alternativas:** Code Climate, Gemnasium.
- **Cloc:** Análisis de las líneas de código del proyecto.

## Pruebas

Siguiendo las prácticas planteadas en el Capítulo 3, se desarrollaron un conjunto de tests y comprobaciones automatizadas en el proceso de integración



y despliegue. Estos tests no solo comprueban la funcionalidad en un momento dado, sino que son reescritos, mejorados y ejecutados a lo largo de todo el proyecto.

## Pruebas dinámicas

Las pruebas dinámicas consisten en pruebas ejecutando el código. Todos los tests dinámicos son implementados en la carpeta `test` de cada servicio y pueden ser ejecutados con `npm test` o las indicaciones dadas en el servicio en cuestión. Todos los tests dinámicos han sido desarrollados para no interferir con los datos o ejecución normal del sistema, trabajando con direcciones, datos y servicios de prueba. Esta configuración se define en los *archivos JSON* en la carpeta `test/config`.

Los tests dinámicos han sido desarrollados con la suite de tests *mocha*[31], y las librerías *supertest* y *chai*. Cada test posee una estructura dividida en:

- **Test Case:** Definidas con la palabra `it`, definen un test independiente, que no debe afectar al resto del sistema ni tests. La configuración se reinicia con cada caso.
- **Suite:** Definidas con una palabra `describe` cada suite representa un conjunto de *tests cases* sobre un módulo o funcionalidad. Cada suite cargará los módulos y dependencias necesarias e inicializará los parámetros de los tests, así como gestionar su ejecución.

```
describe('Test Suite', function() {
  beforeEach(function(){
    //configuración inicial de cada test case
  });
  afterEach(function(){
    //Limpieza del test anterior
  });
  it('Test Case',function(){
    //Test
  });
  it.skip('Test Case 2',function(){
    //Test no ejecutado
  })
})
```

```
});
```

### *Estructura de una suite de test en Mocha*

La ejecución de cada *test case* puede producir 3 resultados: \* **Pass:** Test superado correctamente. \* **Fail:** Test fallado. \* **Skip:** Test no ejecutado. Estos tests, marcados con la palabra **skip** en el código, representan tests planificados u obsoletos que deben ser reprogramados.

## Tests unitarios

Cada módulo de código se desarrolla junto con una serie de pruebas que garanticen su funcionamiento con independencia del resto del sistema. Estos tests se desarrollan en conjunto con el módulo en cuestión, y es preciso que el software apruebe los tests antes de proceder a la integración de dicho módulo en el control de versiones.

Para el desarrollo de los módulos de servidor principales, se ha desarrollado una *suite* de tests unitarios para cada módulo. Consideramos por módulo un bloque de código relevante que es gestionado mediante dependencias de acuerdo al estándar en Node.js y npm[32].

El objetivo de estos tests es garantizar códigos sin *bugs* y probar los casos extremos de dicho código. Así como facilitar rastrear los fallos y los cambios en el código.

## Tests de integración

Se desarrollarán además *suites* orientadas a probar funcionalidad completa del sistema. Generalmente ejecutando múltiples módulos de código o levantando el subsistema completo (**Pruebas de Sistema**). Estos tests se desarrollan con las mismas herramientas que los tests unitarios. Los tests de integración deberán desarrollarse después de los tests unitarios con módulos ya probados individualmente. El objetivos de estos tests, sin embargo, se orienta a garantizar un correcto acoplamiento de los distintos módulos y un funcionamiento correcto en casos de uso reales.

## Pruebas de cobertura

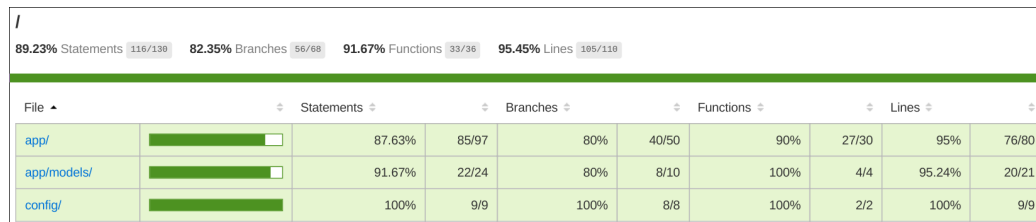
Al desarrollar tests automatizados, es imprescindible analizar si estos tests realmente representan el código que están probando, y si lo siguen representando tras varias iteraciones.

Los tests de cobertura, son tests automáticos generados con la herramienta *Istanbul*[33] al ejecutar los tests unitarios y de integración. Los tests de cobertura analizan la cantidad de código probado (*cobertura del código*[34]).

Para analizar la cobertura se toman 4 parámetros:

- Líneas: Porcentaje de líneas del programa ejecutadas en los tests, es el índice principal usado para analizar la cobertura.
- Funciones: Porcentaje de funciones ejecutadas.
- Ramas: Porcentaje de posibles flujos de del programa ejecutados (por ejemplo, si un condicional `if` a sido verdadero y falso).

Este análisis resulta en un informe detallado de la cobertura del código (Figura 6.1).



The screenshot shows the Istanbul.js coverage report. At the top, it displays overall statistics: 89.23% Statements (116/130), 82.35% Branches (56/68), 91.67% Functions (33/36), and 95.45% Lines (105/110). Below this is a table with columns for File, Statements, Branches, Functions, and Lines. The table lists three files: app/, app/models/, and config/, each with its respective coverage percentages and counts.

File	Statements	Branches	Functions	Lines
app/	87.63% (85/97)	80% (40/50)	90% (40/50)	95% (76/80)
app/models/	91.67% (22/24)	80% (8/10)	100% (4/4)	95.24% (20/21)
config/	100% (9/9)	100% (8/8)	100% (2/2)	100% (9/9)

Figura 6.1: Análisis de cobertura del servicio de usuarios

Para considerar *estable* un módulo de código, las pruebas automáticas deben tener un mínimo de un 75 % de cobertura durante el desarrollo. Los cambios en el código afectarán a la cobertura resultante en los tests, indicando si los tests deberán ser actualizados.

## Integración continua

Para obtener el mayor rendimiento de los tests y favorecer un flujo de trabajo automatizado, se aplicarán técnicas de *Integración Continua* (CI)[35][36] sobre cada servicio a implementar:

- Mantener un repositorio único: Todos los servicios poseen un repositorio en **GitHub**[27] donde se aloja la totalidad del proyecto y donde se integran todos los cambios.
- Construcción automatizada: Toda la instalación y despliegue del proyecto es automática y no requiere intervención humana. Mediante el servicio de integración continua **Travis CI**[30] cada *commit* realizado al repositorio será automáticamente descargado y desplegado.
- Tests automáticos: Los tests funcionales descritos anteriormente son igualmente cargados en el servicio de integración, y serán ejecutados para garantizar que los cambios nuevos no afectan a la funcionalidad ya probada.

De esta forma, con cada cambio en el repositorio será probado y desplegado automáticamente, mediante el servicio **Coveralls** se realizarán también las pruebas de cobertura garantizando que los tests siguen siendo válidos.

El objetivo de la integración continua es agilizar el flujo de trabajo y una detección temprana de los errores, así como realizar un seguimiento de los cambios y garantizar la funcionalidad total del sistema en todo momento.

## Resultados

Las pruebas dinámicas se han implementado para el sistema de usuarios y el sistema de mundo, los principales sistemas del proyecto:

Sistema	Numero de tests	Cobertura (líneas)
Mundo	32	79 %
Usuarios	14	95 %
Total	46	82 %

## Pruebas estáticas

Las pruebas estáticas consisten en una serie de análisis sobre el software que no implican su ejecución. En general nos centramos en herramientas automatizadas que se usan como parte del desarrollo y se integran con la integración continua. Aunque en el proceso iterativo durante la fase de

**refactorización** se realizan revisiones de código y documentación como parte de las pruebas estáticas.

El objetivo de estas pruebas, es conseguir mejorar la calidad del código, proyecto y documentación, generalmente como parte de la refactorización.

## Pruebas de calidad del código

La calidad del código viene dada principalmente por el análisis y diseño inicial, la habilidad de los programadores y, finalmente, el proceso de refactorización. Sin embargo, se aplican diversas herramientas para agilizar estos procesos.

- **JsHint**: Detecta errores y malas practicas en código JavaScript garantizando un código de acuerdo a los estándares js.
- **Js-Beautify**: Programa para indentación automática del código, garantiza un estilo de código común en todo el proyecto, con independencia del desarrollador.
- **BitHound**: Automatiza el proceso de revisión de código como parte del flujo de desarrollo, indicando posibles errores y malas prácticas. Proporciona una nota entre 0 y 100 para indicar la calidad del código.

Es importante tratar de mantener un código de alta calidad y seguir unos estándares en el desarrollo para facilitar las tareas de extensión y mantenibilidad posteriores. Además, un mal código puede tener repercusiones en rendimiento o errores.

## Análisis de dependencias

Las herramientas elegidas para el desarrollo del sistema, así como el ecosistema web en general, exigen el uso de gran cantidad de dependencias y mantenerlas actualizadas pudiendo superar la decena de paquetes de los que depende un sistema (además de las dependencias de estos).

Debido a los riesgos de seguridad, rendimiento y compatibilidad, es preciso mantener todas las dependencias actualizadas, para ello se aprovechará el propio gestor de paquetes **npm** que permite actualizar dependencias obsoletas. Además como parte del flujo de desarrollo, el servicio BitHound también

permite mantener un análisis continuo de dependencias en el repositorio, avisando en caso de que se desactualicen.

Aunque algunos proyectos optan por un servicio de actualización automático de dependencias, una actualización no supervisada puede conllevar una rotura en el proyecto debido a incompatibilidad, por tanto la actualización de dependencias se mantiene como una tarea independiente del flujo automatizado.

## Análisis de commits

Como parte de la revisión manual de código, cada *commit* al repositorio es analizado manualmente para revisar sus cambios, se ejecutan todos los tests pertinentes y se envía a la rama de desarrollo del repositorio.

Para integrar un conjunto del proyecto con la rama *master* es imprescindible que los commits realizados cumplan con todos los tests y no realicen cambios imprevistos.

## Despliegue

A mitad de desarrollo aproximadamente, se procedió al despliegue del sistema en servicios **PaaS** (Platform as a Service). A partir de ese despliegue se procedió a incorporar la fase de despliegue a las operaciones automáticas como parte del proceso de integración continua, desplegando cada servicio tras pasar las pruebas en una versión estable (rama *master*).

Aprovechando la arquitectura distribuida, se procedió a un despliegue en distintos PaaS para cada servicio de acuerdo a las necesidades del servicio:

- Servidor de usuarios: Desplegado en **Openshift**, servicio de RedHat. Permite el uso de base de datos mongo y escalado automático de la aplicación.
- Servidor web: Desplegado en **Heroku**, al no necesitar base de datos, se decidió desplegarlo en heroku, que aunque no permite usar mongoDB, es más fácil de integrar y más rápido en desplegar que Openshift.
- Servidor del mundo: Actualmente desplegado en **Heroku**, con una Base de datos en **Mlab**.

Este despliegue, además, trata de probar el comportamiento del sistema bajo un entorno completamente distribuido, haciendo uso simultáneo de diversos servidores por parte de un único cliente.

## Capítulo 7

# Conclusiones y trabajos futuros

Al finalizar el proyecto, el producto resultante fue un *prototipo funcional* de un videojuego multijugador masivo online, así como una memoria sobre este tipo de proyectos, las dificultades en su implementación y posibles soluciones. Además, se han estudiado temas relacionados con la programación web y sistemas distribuidos así como en la gestión de un proyecto de tamaño medio.

El prototipo desarrollado demuestra una arquitectura y un conjunto de tecnologías *viable* para resolver el problema, permitiendo un desarrollo posterior. Finalmente, las primeras pruebas en despliegue demuestran un prototipo que cumple con los requisitos no funcionales del sistema, proporcionando un servicio eficiente y con una latencia mínima.

## Problemas encontrados

La escasa documentación y tecnologías existentes en el problema de los juegos multijugador masivos, además del desconocimiento de las tecnologías adecuadas para su desarrollo supuso un retraso en el tiempo total de desarrollo, al tener que solucionar *spikes* y probar diversas opciones en prácticamente todos los ámbitos del proyecto. En general, los las principales dificultades radican en la arquitectura y la tecnología a aplicar:

- Arquitectura del servidor: Una arquitectura estándar no era suficiente para desarrollar este sistema y entre las arquitecturas distribuidas



existían diversas opciones. Finalmente se optó por el híbrido entre una arquitectura orientada a servicios estándar y una arquitectura de microservicios especificada en el Capítulo 5.

- Tecnología del servidor: Era preciso una tecnología eficiente, ligera y que se adecuara correctamente a una arquitectura distribuida y a los estándares web modernos.
- Estructura de datos: La cantidad de modificaciones y actualizaciones sobre los datos del juego requerían de diversas capas de datos para mejorar la latencia y escalabilidad del sistema.
- Tecnologías del cliente: Existen diversas opciones para cliente del sistema, finalmente se optó por un cliente web, sin embargo, esto requería aprovechar las tecnologías existentes para desarrollar un cliente capaz de trabajar de forma asíncrona y con baja latencia.
- Técnicas de desarrollo: El desarrollo de un sistema con un cierto tamaño y complejidad requiere la aplicación de técnicas de gestión de proyecto (metodología ágil, testeo, repositorio único, etc.) aplicadas al problema especificado.
- Sesiones en un sistema distribuido: Fue necesario estudiar los mecanismos para mantener sesiones persistentes en un servidor distribuido.

Las soluciones expuestas en el desarrollo del prototipo resuelven o mitigan la mayor parte de estos problemas, lo que permitiría un desarrollo con un menor coste e incertidumbre al aplicarlas al desarrollo o mejora de nuevos productos.

## Conocimientos adquiridos

Durante el análisis y desarrollo de este proyecto, se han adquirido conocimientos en diversos ámbitos del desarrollo software aplicando tanto tecnologías ya conocidas como experimentando con opciones novedosas:

- Programación de aplicaciones web: Se han implementado un stack completo de aplicaciones web.
  - *Front\_End*: Se han desarrollado interfaces de las aplicaciones mediante un conjunto de tecnologías web modernas para el desarrollo de clientes web como *Bootstrap* y *React*.
  - *Back\_End*: Implementación del lado servidor de una aplicación

web, con especial énfasis en conseguir un sistema eficiente y con baja latencia mediante la implementación de servicios y APIs.

- Programación de sistemas distribuidos: Implementación y despliegue de un servidor distribuido y comunicación a tiempo real entre cliente y servidor mediante *websockets*.
- Bases de Datos: Desarrollo de bases de datos tanto relacionales (*MySQL*) como no relacionales (*MongoDB*) y análisis de diversas opciones para almacenamiento de datos.
- Gestión de proyectos software: Gestión de un proyecto de tamaño medio aplicando técnicas de desarrollo software y metodología ágil.
- Pruebas de Software: Desarrollo de tests automatizados sobre el software tanto estáticos como dinámicos.
- *DevOps*: Aplicación de un flujo de desarrollo, testeo y despliegue automatizado junto con integración continua.

## Asignaturas relevantes

Este proyecto aplica los conocimientos adquiridos en diversas asignaturas tanto generales como de especialidad:

- Asignaturas de programación (fundamentos) y programación orientada a objetos.
- Bases de datos, estructuras de datos, diseño y desarrollo de sistemas de información.
- Desarrollo software, metodologías de desarrollo ágil y programación lúdica.
- Sistemas de información basados en web y desarrollo de aplicaciones de internet.
- Desarrollo de sistemas distribuidos, desarrollo basado en agentes, sistemas concurrentes y distribuidos.

## Trabajos futuros

Este prototipo, si bien resuelve los problemas antes mencionados, aún dista de ser un software listo para producción. Además, abre múltiples líneas de

trabajo orientadas al desarrollo del videojuego completo y del framework final para su aprovechamiento en otros proyectos:

- Desarrollo de un microservicio para el servicio de juego: Cómo se indica en la arquitectura del servidor (Capítulo 5), se propone aplicar la arquitectura de microservicios específicamente sobre el servicio de juego, para mejorar la escalabilidad manteniendo la baja latencia actual.
- Mejora del aspecto visual del videojuego: El prototipo de cliente actual requiere de un mejor diseño gráfico e interfaz.
- Implementación de cliente para móvil y otras plataformas: El servidor actual posee la capacidad de gestionar el juego con independencia de la plataforma cliente, por lo que se recomienda desarrollar clientes específicos para móviles y escritorio.
- Encapsulamiento y generalización de la plataforma: Es preciso facilitar la reutilización de la plataforma, principalmente el servicio de juego, facilitando la adaptación de las reglas de juego.
- Implementación de otros servicios para extender funcionalidad: La arquitectura actual contempla el desarrollo de nuevos servicios (por ejemplo un servicio de chat o mensajería) para extender las capacidades del juego.

Una estimación preliminar indica un tiempo de desarrollo aproximado de 6 meses con un equipo de entre 3 a 5 desarrolladores para obtener una versión completa del producto, lista para producción.

## **Posible comercialización**

Tras un análisis del producto y las necesidades que resuelve, se encuentran dos posibles líneas comerciales del producto:

- Desarrollo y venta de videojuegos MMO: El sistema y tecnología actual permitiría obtener una fuente de ingresos vendiendo juegos (el modelo de negocio específico puede variar entre juegos). Esto requeriría un equipo de desarrollo de videojuegos permanente para continuar el desarrollo de la plataforma y el desarrollo de los videojuegos específicos.
- Desarrollo y venta del framework: Un modelo de negocio, posiblemente más lucrativo, aunque requeriría más tiempo para obtener ingresos, sería el desarrollo de un framework genérico de desarrollo, con estilo

similar a frameworks cómo Unity (<https://unity3d.com>) o Unreal (<https://www.unrealengine.com>), con modelos de negocio similares, aunque orientado a videojuegos MMO y compatibles con estos frameworks.

Debido al elevado coste de desarrollo estimado, sería necesario aprovechar ambos modelos de negocio para obtener un negocio viable, además de aprovechar los videojuegos desarrollados como publicidad para el framework, se propone el desarrollo de un *Producto Mínimo Viable* (MVP) a partir del prototipo ya realizado para reducir la incertidumbre en relación a la comercialización de este producto.

# Capítulo 8

## Apéndices

### Glosario

Aquí se indexan una serie de términos en orden alfabético.

- **AGPL** Affero GPL, licencia *open source* usada en este proyecto[37].
- **CORS** Cross-Origin resource sharing, mecanismo para permitir el envío de contenido a aplicaciones web de distinto origen[26].
- **Dev-Ops** Técnicas y prácticas que tienen por objetivo automatizar y agilizar el proceso de despliegue y entrega software.
- **Framework** Software genérico que proporciona funcionalidades para su implantación en sistemas específicos a partir de modificaciones y adiciones particulares.
- **HU** Siglas de *Historia de Usuario*[9] (*User Story* en inglés). Breves descripciones de las necesidades de los clientes referentes al sistema, útiles para estimaciones.
- **Integración Continua** Práctica de unir todas las copias de un proyecto de software frecuentemente, evitando los posteriores *problemas de integración* al unir el trabajo de todos los desarrolladores.
- **Issue** Define una tarea de desarrollo (mejora, error a arreglar,...) para gestionar el desarrollo software a corto plazo.

- **JWT**[20] Siglas de *JSON Web Token*, tecnología de tokens para verificación de identidad usada en maelström
- **LOC** *Lines Of Code* (Líneas de código), métrica para indicar el tamaño aproximado de un proyecto, cuenta las líneas de código (eliminando líneas en blanco y comentarios).
- **Microservicio** Servicio mínimo e independiente dentro de un mismo sistema[17]. Los microservicios son un caso particular de las Arquitecturas orientadas a servicios.
- **MMO** Siglas en ingles de *Massively Multiplayer Online* referentes a los juegos multijugador masivo
- **MVP** Siglas de *Minimum Viable Product*, un producto con las suficientes características como para permitir realizar un análisis y recolectar datos validados sobre el futuro de dicho producto
- **Open Source** Código abierto, referente al desarrollo libre de un proyecto de software.
- **PaaS** *Platform as a Service* (Plataforma como servicio), servicio de computación en la nube que proporciona una plataforma para ejecutar y desplegar una aplicación sin necesidad de gestionar la infraestructura de un servidor, proporcionando una capa de abstracción entre la máquina, el sistema operativo y la aplicación.
- **Roadmap**[10] Planificación a corto y largo plazo de los objetivos de un producto
- **RTS** Siglas de *Real Time Strategy* referente a juegos del género de estrategia en tiempo real
- **SOA** *Service Oriented Architecture* (Arquitectura orientada a servicios), es una arquitectura software en la que el sistema se compone de distintos componentes independientes (servicios) que se comunican entre ellos.
- **Spike** Programa simple que busca una posible solución a una dificultad técnica
- **XP** Siglas de *eXtreme Programming*[6][7], metodología de desarrollo software

## Licencias

Todo el proyecto ha sido realizado con tecnología libre, y su código ha sido liberado bajo licencia **GNU AFFERO GENERAL PUBLIC LICENSE, Version 3**[37] salvo que se indique lo contrario en los siguientes repositorios:

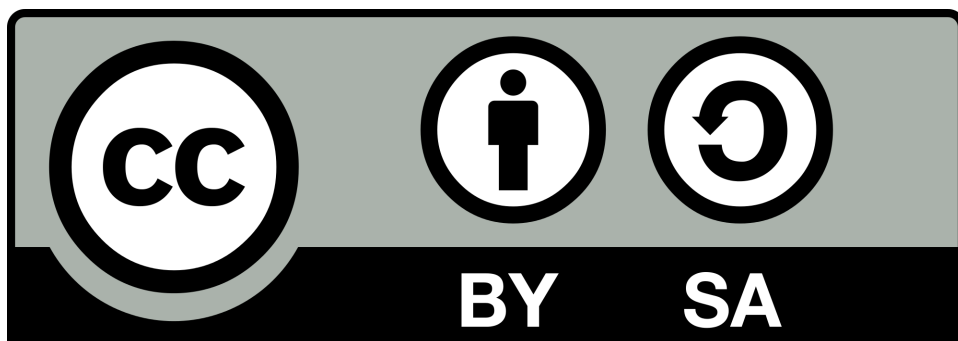
- <https://github.com/demiurgosoft/maelstrom>
- <https://github.com/demiurgosoft/maelstrom-users>
- <https://github.com/demiurgosoft/maelstrom-web>
- <https://github.com/demiurgosoft/maelstrom-world>
- <https://github.com/demiurgosoft/maelstrom-messages>
- <https://github.com/demiurgosoft/maelstrom-bot>

Todos los recursos visuales del juego desarrollados han sido liberados bajo copyright **Creative Commons** incluyendo:

- **Logotipo de Maelström** de Israel Blancas bajo licencia **CC-BY-SA**  
[https://commons.wikimedia.org/wiki/File:Maelstr%C3%B6m\\_Logo.png](https://commons.wikimedia.org/wiki/File:Maelstr%C3%B6m_Logo.png)



- **Memoria del Proyecto CC-BY-SA**



## Preámbulo de la licencia

### GNU AFFERO GENERAL PUBLIC LICENSE

Version 3, 19 November 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/> Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

#### Preamble

The GNU Affero General Public License is a free, copyleft license for software and other kinds of works, specifically designed to ensure cooperation with the community in the case of network server software.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, our General Public Licenses are intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

Developers that use our General Public Licenses protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License which



gives you legal permission to copy, distribute and/or modify the software.

A secondary benefit of defending all users' freedom is that improvements made in alternate versions of the program, if they receive widespread use, become available for other developers to incorporate. Many developers of free software are heartened and encouraged by the resulting cooperation. However, in the case of software used on network servers, this result may fail to come about. The GNU General Public License permits making a modified version and letting the public access it on a server without ever releasing its source code to the public.

The GNU Affero General Public License is designed specifically to ensure that, in such cases, the modified source code becomes available to the community. It requires the operator of a network server to provide the source code of the modified version running there to the users of that server. Therefore, public use of a modified version, on a publicly accessible server, gives the public access to the source code of the modified version.

An older license, called the Affero General Public License and published by Affero, was designed to accomplish similar goals. This is a different license, not a version of the Affero GPL, but Affero has released a new version of the Affero GPL which permits relicensing under this license.

The precise terms and conditions for copying, distribution and modification follow.

## Especificación de la API

La API de Maelström sigue las normas básicas de una **REST API**:

- Peticiones **HTTP**
- Uso de los métodos HTTP:
  - **GET**
  - **POST**
  - **UPDATE**
  - **DELETE**
- Todas las peticiones devolverán una respuesta **JSON** indicando el resultado de la operación
- Todas las peticiones devolverán un código de estado (especificado más adelante)
- Las peticiones GET no tendrán ninguna carga en el *body* del mensaje, mientras que el resto si
- Todos los datos en el *body* han de ser provistos en formato **JSON**
  - Ejemplo:

```
{
  "username": "arthur",
  "password": "dontpanic42"
}
```

## Identificación

La verificación del usuario entre los diversos servicios se realizará mediante un *JSON Web Token*[20] (JWT) con una id común para cada usuario.

El payload del token poseerá la siguiente estructura

```
{
  "id": "56d96ce3a5e8cf4c28e1a4a4",
  "username": "my user",
  ...
}
```

Este token podrá ser almacenado por el cliente (una cookie, en memoria o en

un archivo) y se podrá usar como token de acceso a los servicios y acciones que requieran identificación. Para ellos, se enviará el *header* de autenticación `Bearer [token jwt]`

## Maelström user

- **POST /login**
  - Da de alta al usuario
  - **Respuesta:** token de acceso {"token"}
  - **Argumentos**
    - **username:** Nombre de usuario en el sistema
    - **password:** contraseña del usuario
  - **Status**
    - 200: Operación correcta
    - 403: Password inválido
    - 404: Usuario no registrado
    - 500: En caso de error interno en servidor
- **POST /signup**
  - Crea un nuevo usuario en el servidor si o existe
  - **Respuesta:** token de acceso {"token"}
  - **Argumentos**
    - **username:** Nombre del nuevo usuario
    - **password:** Contraseña del nuevo usuario
    - **email:** Correo electrónico del usuario
  - **Status**
    - 201: Usuario se ha creado correctamente
    - 400: Algún formulario se encuentra vacío
    - 500: Error interno, formulario incorrecto o usuario ya existente
- **PUT /restricted/update**
  - Actualiza los datos de usuario con nuevos datos si estos datos no coinciden con otro usuario
  - **Respuesta:** Mensaje de error si hubiera
  - **Identificación necesaria**
  - **Argumentos**
    - **username:** Nuevo nombre de usuario
    - **email:** Nuevo email
    - **password:** Nuevo password

- **Status**
  - 204: Operación correcta
  - 400: Operación inválida (usuario no existe o nuevos datos no válidos)
  - 401: No autorizado (token inválido)
- **DELETE /restricted/remove**
  - Elimina el usuario (el token no será válido después de esta operación)
  - **Respuesta:** Vacío si la operación es correcta, error en otro caso
  - **Identificación necesaria**
  - **Argumentos:** Sin Argumentos
  - **Status**
    - 204: Operación correcta
    - 400: Operación inválida (usuario no existe o nuevos datos no válidos)
    - 401: No autorizado (token inválido)
- **GET /restricted/dash**
  - Obtiene la información del usuario
  - **Respuesta**

```
{
  "_id": "Id del usuario",
  "username": "Nombre del usuario",
  "email": "Correo del usuario"
}
```
  - **Identificación necesaria**
  - **Status**
    - 200: Operación correcta
    - 400: Operación inválida (usuario no encontrado)
    - 401: No autorizado (token inválido)

## Maelström world

- **GET /map**
  - Obtiene información del mapa, lista de todas las ciudades
  - **Respuesta:** Array de las ciudades [city1,city2,...], mensaje de error si lo hubiera
  - **Status**

- 200: Operación correcta
  - 500: Error
- GET /city/:city\_name
  - Obtiene información de la ciudad city\_name
  - **Respuesta:** Datos de la ciudad en formato JSON
 

```
{
  "name": "Nombre de la ciudad",
  "position": "posición de la ciudad en formato [x,y]",
  "products": "Lista de productos (nombre{quantity,production})",
  "slug": "nombre-de-la-ciudad"
}
```
  - **Status**
    - 200: Operación correcta
    - 500: Error
- GET /city/products/:city\_name
  - Obtiene información de todos los productos de una ciudad (cantidad, producción y precio)
  - **Respuesta**

```
{
  "product1": {
    "quantity": "Cantidad de producto en ciudad",
    "production": "Producción de la ciudad",
    "price": "Precio actual del producto"
  },
  "product2": {...}
}
```
  - **Status**
    - 200: Operación correcta
    - 500: Error
- GET /ship\_models
  - Obtiene un array de los modelos de barcos
  - **Respuesta**

```
[
  {
    "name": "Nombre del modelo",
    "life": "Vida del barco",
    "speed": "Velocidad del barco",
    "price": "Precio de construcción",
```

```

        "cargo": "Capacidad de carga del barco",
        "slug": "nombre-del-modelo"
    },
    {
        ...
    }
]

```

- **Status**

- 200: Operación correcta
- 500: Error

- GET /products

- Obtiene una lista de todos los productos del juego
- **Respuesta:** Array de productos [product1,product2,...]
- **Status**
  - 200: Operación correcta
  - 500: Error

- GET /user/ships

- Devuelve la lista de barcos del usuario autenticado
- **Identificación necesaria**
- **Respuesta:** Un array con la información básica de los barcos del usuario

```

[
    {
        "name": "Nombre del barco",
        "model": "Modelo del barco",
        "slug": "nombre-del-barco",
        "life": "Vida del barco",
        "status": "Estado actual del barco"
    },
    {
        ...
    }
]

```

- **Status**

- 200: Operación correcta
- 401: No autorizado (token inválido)
- 500: Error

- GET /user/ship/:ship\_id

- Devuelve los datos del barco del usuario con id dada
- **Identificación necesaria**
- **Argumentos**
  - :ship\_id id del barco
- **Respuesta**

```
{
  "name": "Nombre del barco",
  "owner": "Nombre del propietario (jugador)",
  "model": "Modelo del barco",
  "life": "Vida del barco",
  "city": "Ciudad actual del barco",
  "status": "Estado actual del barco",
  "carga": {"producto1":"cantidad1","producto2":"cantidad2",...},
  "slug": "nombre-del-barco",
}
```
- **Status**
  - 200: Operación correcta
  - 401: No autorizado (token inválido)
  - 500: Error
- GET /user/data
  - Devuelve información del usuario
  - **Identificación necesaria**
  - **Respuesta**

```
{
  "id":"Id del usuario",
  "money": "Dinero del usuario"
}
```
  - **Status**
    - 200: Operación correcta
    - 401: No autorizado (token inválido)
    - 500: Error
- POST /user/signup
  - Crea un nuevo usuario en el servicio si no existe
    - **Identificación necesaria**
    - **Status**
      - ◊ 201: Operación correcta
      - ◊ 401: No autorizado (token inválido)
      - ◊ 500: Error

- PUT /user/build/ship
  - Realiza la acción de construir un barco
  - **Identificación necesaria**
  - **Argumentos**

```
{
  "model": "Modelo del barco",
  "ship_name": "Nombre del barco",
  "city": "Ciudad de origen"
}
```
  - **Respuesta:** Datos del barco
  - **Status**
    - 201: Operación correcta
    - 400: Formato inválido
    - 401: No autorizado (token inválido)
    - 500: Error
- PUT /user/move/ship
  - Acción de mover barco
  - **Identificación necesaria**
  - **Argumentos**

```
{
  "ship": "Id del barco",
  "city": "Ciudad destino"
}
```
  - **Status**
    - 200: Operación correcta
    - 400: Formato inválido
    - 401: No autorizado (token inválido)
    - 500: Error
- PUT /user/buy
  - Acción de comprar productos
  - **Identificación necesaria**
  - **Argumentos**

```
{
  "ship": "Id del barco",
  "product": "Producto a comprar",
  "quantity": "Cantidad a comprar"
}
```
  - **Status**



- 200: Operación correcta
  - 400: Formato inválido
  - 401: No autorizado (token inválido)
  - 500: Error
- PUT /user/sell
  - Acción de vender productos
  - **Identificación necesaria**
  - **Argumentos**

```
{
  "ship": "Id del barco",
  "product": "Producto a vender",
  "quantity": "Cantidad a vender"
}
```
  - **Status**
    - 200: Operación correcta
    - 400: Formato inválido
    - 401: No autorizado (token inválido)
    - 500: Error

## Eventos de sockets

## Códigos de estado

Code	Significado
200	Operación correcta
201	Operación de añadido correcta
204	Operación de modificación correcta
400	Petición inválida
401	No autorizado (token inválido)
403	Prohibido (error en contraseña)
404	Datos no encontrados
500	Error interno en servidor

# Manual de usuario

## Introducción

En la Europa renacentista, el comercio marítimo hace florecer las distintas capitales del continente. En Maelström tomarás el papel de un mercader e intentarás hacer fortuna con tu flota para expandirte y cooperar o competir con otros jugadores.

## Mundo

El juego transcurre en diversas Ciudades de la Europa renacentista que los jugadores podrán visitar. Cada ciudad se encuentra a una cierta distancia de otra, los viajes pueden ser muy largos!



*Mapa de Maelström*

## Ciudades

Cada una de las ciudades que el jugador puede visitar tiene un mercado, donde se podrá **comprar** y **vender productos** si hay un barco atracado.

Cada ciudad es distinta, producirá y consumirá unos productos distintos, y por tanto, tendrán un precio distinto.

Para tener éxito comerciando, es importante tener en cuenta los precios de los productos en cada ciudad, las acciones de los jugadores también pueden alterar los precios de los productos.

Product	Ship	Actions	City	Price
rice	0	<input type="text"/> Buy Sell	0	40
beer	0	<input type="text"/> Buy Sell	0	60
timber	0	<input type="text"/> Buy Sell	500	8
wheat	10	<input type="text"/> Buy Sell	0	36
bread	10	<input type="text"/> Buy Sell	500	12
clothes	10	<input type="text"/> Buy Sell	500	34

*Ventana de compra y venta de productos*

En Maelström encontrarás las siguientes ciudades:

- Algiers
- Barcelona

- Cádiz
- Lisboa
- Londres
- Riga

## Productos

En Maelström podrás comerciar con múltiples productos, cada uno posee un precio y se produce y consume en distintas ciudades. Los precios cambian constantemente en el mercado, es importante estar atento del precio de cada producto en las ciudades para conseguir beneficio al llevarlo de una ciudad a otra.

## Jugadores

Los distintos jugadores tendrán libertad para construir barcos y comerciar entre las ciudades, sin embargo, necesitarán **dinero** para ello. Cada jugador comienza con una cantidad de dinero inicial, con la que podrá construir barcos y comprar productos. Para seguir creciendo tendrá que obtener beneficios de sus transacciones. Los jugadores también podrán tomar actitudes competitivas o cooperativas entre ellos.

## Barcos

Los barcos son la fuente principal de ingresos de los jugadores, los jugadores podrán construir distintos tipos de barcos (**goletas**, **carabelas** y **galeones**), cada barco tendrá una **velocidad** y una **capacidad de carga** distinta, el jugador deberá aprovechar sus barcos y sus características para obtener el mayor beneficio posible.



*Lista de barcos del jugador*

## Versiones

Para poder gestionar el proyecto, se ha seguido una notación de versiones basadas *semantic versioning 2.0.0*[38] de tres números (major.minor.patch), cada servicio posee su propias versiones al tener un desarrollo independiente.

Al ser prototipos, todos los servicios se comenzaron a desarrollar bajo una versión 0, cada funcionalidad o hito generalmente corresponde a una versión menor (0.1), el número de patch se reserva para issues resueltos, bugs arreglados o cambios relevantes en el código dentro de la misma versión. A continuación se enumeran las principales versiones de cada servicio junto con algunas métricas del desarrollo:

- **LOC:** *Lines Of Code*, número de líneas de código (sin contar líneas en blanco, comentarios ni código de los tests)
- **Módulos:** Número de módulos (archivos JS en este caso)
- **Tests:** Número total de tests implementados
- **Cobertura:** Cobertura (en líneas) de los tests
- **Estado:** Estado del proyecto en esa versión (basado en funcionalidad implementada y tests)
  - No Funcional: El código no supera los tests o no funciona correctamente
  - Funcional: El código funciona, pero se encuentra con funcionalidad limitada o sin probar
  - En Pruebas: El código posee funcionalidad completa y se están desarrollando nuevos tests.
  - Estable: El código posee funcionalidad completa y probada, sin errores graves durante múltiples cambios
  - Inestable: Se encontraron errores o los test resultaron obsoletos en un código estable
  - Release: Versión estable apta para despliegue

## Servicio de usuarios

Última versión estable: **1.0.0**

Versión	Fecha	LOC	Módulos	Tests	Cobertura	Estado
0.0.1	13/10/2015	72	3	0	0 %	No Funcional
0.0.5	26/10/2015	337	6	12	87 %	No Funcional
0.1.0	28/10/2015	339	6	13	93 %	No Funcional
0.1.5	15/11/2015	340	6	13	96 %	Funcional
0.1.6	04/03/2016	357	6	13	96 %	Funcional
0.1.9	28/04/2016	371	6	13	93 %	En Pruebas
0.2.0	03/05/2016	375	6	13	92 %	Estable
0.2.1	12/05/2016	375	6	14	95 %	Estable
0.2.2	11/06/2016	385	6	14	95 %	Estable
1.0.0	30/06/2016	385	6	14	95 %	Release

## Servicio del mundo

Última versión estable: **0.8.1**

Versión	Fecha	LOC	Módulos	Tests	Cobertura	Estado
0.0.1	28/10/2015	36	1	0	0 %	No Funcional
0.0.17	20/11/2015	626	11	11	58 %	No Funcional
0.1.0	20/11/2015	626	11	1	58 %	En Pruebas
0.1.4	28/11/2015	645	6	29	87 %	En Pruebas
0.2.0	08/12/2015	645	6	29	38 %	No Funcional
0.2.9	06/03/2016	624	15	1	29 %	No Funcional
0.3.0	10/03/2016	675	15	2	40 %	No Funcional
0.3.6	18/04/2016	897	16	22	76 %	Funcional
0.4.0	02/05/2016	997	17	24	73 %	En Pruebas
0.4.4	05/05/2016	1038	17	25	73 %	En Pruebas
0.5.0	10/05/2016	1116	18	20	64 %	Inestable
0.5.4	15/05/2016	1174	18	29	80 %	Inestable
0.6.0	15/05/2016	1175	18	30	80 %	En Pruebas
0.6.4	21/05/2016	1183	18	31	80 %	Estable
0.7.0	04/06/2016	1201	18	31	75 %	En Pruebas

Versión	Fecha	LOC	Módulos	Tests	Cobertura	Estado
0.7.2	09/06/2016	1211	18	31	79 %	Estable
0.8.0	11/06/2016	1226	17	31	79 %	Release
0.8.1	09/06/2016	1257	17	32	79 %	Release

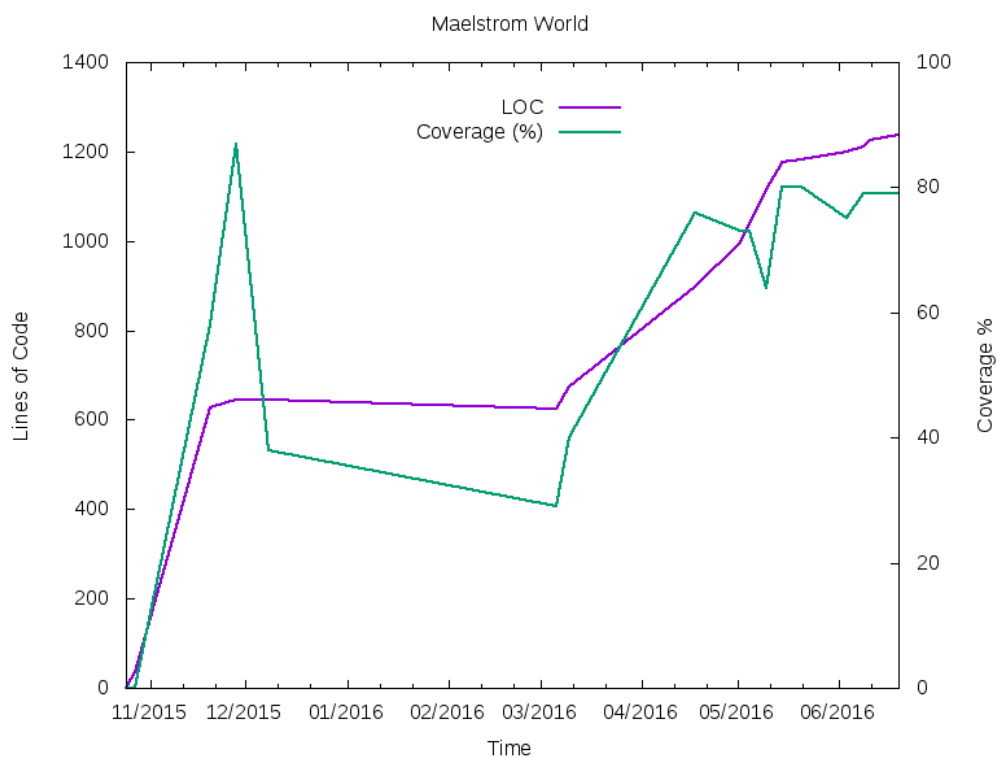


Figura 8.1: Evolución del desarrollo del servicio del mundo

## Servicio web

Última versión estable: **0.6.2**

Versión	Fecha	LOC	Módulos (JS)	Estado
0.1.0	27/03/2016	240	2	No Funcional



Versión	Fecha	LOC	Módulos (JS)	Estado
0.2.0	10/04/2016	967	13	Funcional
0.2.3	02/05/2016	973	15	En Pruebas
0.3.0	03/05/2016	976	15	En Pruebas
0.3.5	15/05/2016	1164	18	Estable
0.4.0	15/05/2016	1204	18	En Pruebas
0.4.7	21/05/2016	1206	18	Estable
0.5.0	04/06/2016	1241	18	Estable
0.5.2	09/06/2016	1289	19	Estable
0.6.0	11/06/2016	1291	19	Estable
0.6.2	26/06/2016	1330	19	Release
0.6.3	08/06/2016	1371	20	Release

## Bot

Para el desarrollo del bot se siguió la misma notación de versiones. Última versión estable: **0.1.1**

Versión	Fecha	LOC	Módulos	Estado
0.0.1	19/05/2016	82	4	No Funcional
0.0.4	02/06/2016	339	9	Funcional
0.1.0	08/06/2016	356	9	Funcional
0.1.1	08/06/2016	353	9	Estable

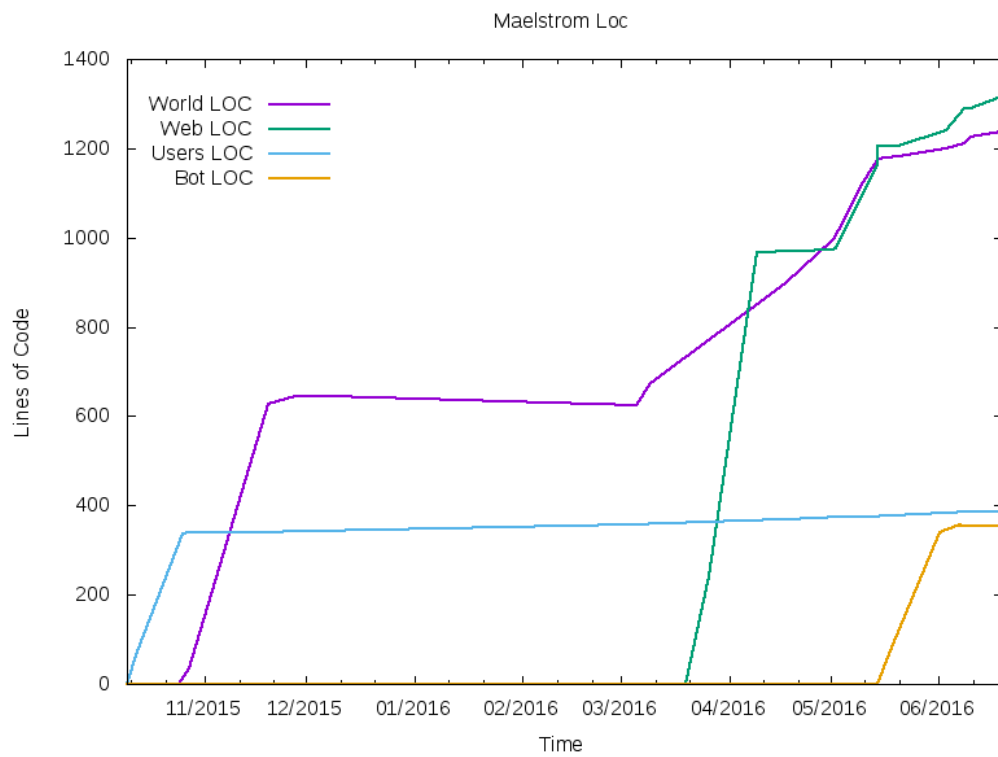


Figura 8.2: Evolución del desarrollo de los servicios

## Diagrama de burndown

El diagrama de Burndown (Figura 8.3) del proyecto representa la cantidad de *Puntos de historia* resueltas a lo largo del desarrollo de cada módulo. Este diagrama permite realizar un seguimiento en el desarrollo y comprobar la relación con las LOC (Figuras 8.1 y 8.2).

Como puede observarse en el diagrama, debido a los cambios en la planificación y la priorización de las tareas a completar (Véase Capítulo 3). Algunos servicios no se encuentran completados al 100 % respecto a las Historias de usuario inicialmente planificadas.

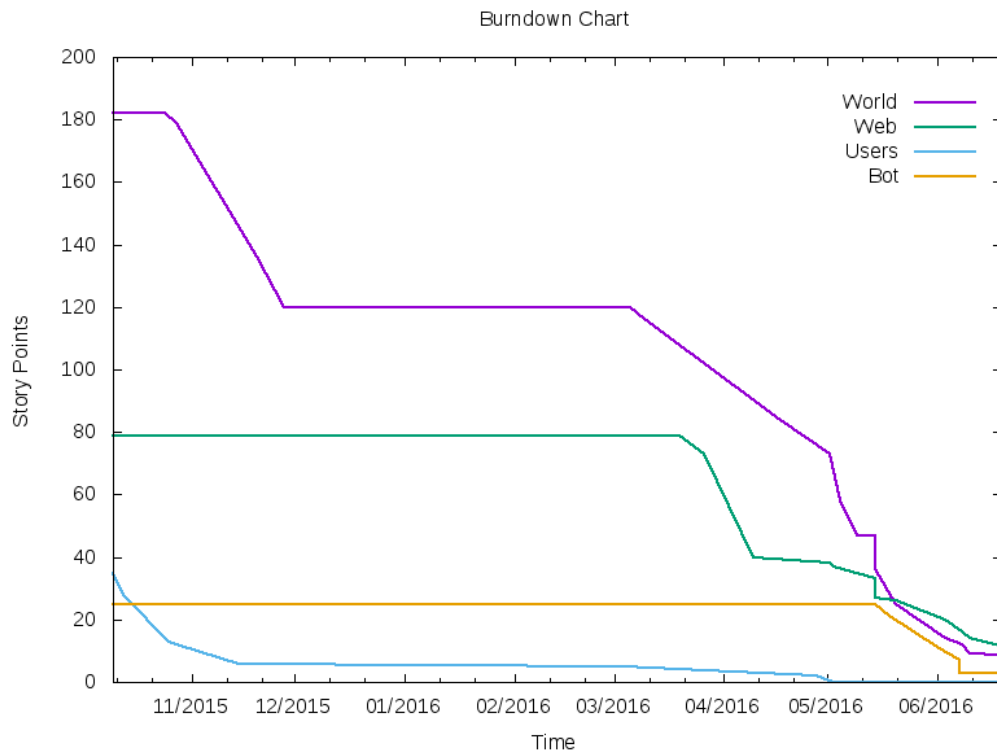


Figura 8.3: Diagrama de Burndown

# Bibliografía

- [1] *Multiplayer video game* - *Wikipedia*. Available: [https://en.wikipedia.org/wiki/Multiplayer\\_video\\_game](https://en.wikipedia.org/wiki/Multiplayer_video_game). Accessed 08 May 2016.
- [2] *Massively multiplayer online game* - *Wikipedia*. Available: [https://en.wikipedia.org/wiki/Massively\\_multiplayer\\_online\\_game](https://en.wikipedia.org/wiki/Massively_multiplayer_online_game). Accessed 08 May 2016.
- [3] *The Open Source Definition*. Available: <https://opensource.org/osd>. Accessed 2007.
- [4] *Manifesto for Agile Software Development*, 2001. Available: <http://www.agilemanifesto.org>.
- [5] *Agile Methodologies for Software Development*. Available: <https://www.versionone.com/agile-101/agile-methodologies>. Accessed 10 May 2016.
- [6] D. Wells, *Extreme Programming Introduction*, 1999. Available: <http://www.extremeprogramming.org>. Accessed 18 May 2016.
- [7] D. Wells, *Extreme Programming Rules*, 1999. Available: <http://www.extremeprogramming.org/rules.html>. Accessed 18 May 2016.
- [8] R. E. Jeffries, *What is Extreme Programming?*, 2011. Available: <http://ronjeffries.com/xprog/what-is-extreme-programming>.
- [9] D. Wells, *User Stories - Extreme Programming*, 1999. Available: <http://www.extremeprogramming.org/rules/userstories.html>. Accessed 18 May 2016.
- [10] D. Radigan, *Agile roadmaps: build, share, use, evolve*. Available: <https://>

- [//www.atlassian.com/agile/roadmaps](http://www.atlassian.com/agile/roadmaps). Accessed 12 Jun 2016.
- [11] D. Wells, *Spikes - Extreme Programming*, 1999. Available: <http://www.extremeprogramming.org/rules/spike.html>. Accessed 19 May 2016.
- [12] Agile Learning Labs, *The Agile Dictionary: Spike*, 2010. Available: <http://agiledictionary.com/209/spike>.
- [13] *Patrician III: Rise of the Hanse - Wikipedia*. Available: [https://en.wikipedia.org/wiki/Patrician\\_III:\\_Rise\\_of\\_the\\_Hanse](https://en.wikipedia.org/wiki/Patrician_III:_Rise_of_the_Hanse). Accessed 09 May 2016.
- [14] *Ogame*. Available: <https://es.ogame.gameforge.com>. Accessed 09 May 2016.
- [15] E. Adams and J. Dormans, *Game Mechanics: Advanced Game Design*. Pearson, 2012.
- [16] H. S. Chu, *IBM developers - Building a MMO game architecture*, 2008. Available: <https://www.ibm.com/developerworks/library/ar-powerup1>.
- [17] J. Lewis and M. Fowler, *Microservices, a definition of this new architectural term*, 2014. Available: <http://martinfowler.com/articles/microservices.html>.
- [18] P. J. Molina, *Microservicios sobre MEAN stack*, 2016. Available: <http://es.slideshare.net/pjmolina/opensouthcode-microservicios-sobre-mean-stack>.
- [19] *Socket.io*. Available: <http://socket.io>. Accessed 26 May 2016.
- [20] *JSON Web Tokens (JWT) official webpage*. Available: <http://jwt.io>. Accessed 27 May 2016.
- [21] S. W. Ambler, *UML 2 Class Diagrams: An Agile Introduction*. Available: <http://www.agilemodeling.com/artifacts/classDiagram.htm>. Accessed 28 Jun 2016.
- [22] S. Stefanov, *3 ways to define a JavaScript class*, 2006. Available: <http://www.phpied.com/3-ways-to-define-a-javascript-class>.
- [23] *Salted Password Hashing - Doing it Right*, 2016. Available: <https://crackstation.net/hashing-security.htm>.
- [24] Node.js Foundation, *Node.js official webpage*, 2015. Available: <https://nodejs.org/en/>.

[//nodejs.org/en](http://nodejs.org/en).

- [25] T. Hughes-Croucher and M. Wilson, *Node: Up and Running*. O'Reilly Media, 2012.
- [26] *enable cross-origin resource sharing*, 2006. Available: <http://enable-cors.org>.
- [27] GitHub, Inc., *GitHub Official Site*, 2008. Available: <https://github.com>.
- [28] *Git Official Webpage*. Available: <https://git-scm.com>.
- [29] S. Chacon and B. Straub, *Pro Git*. Apress, 2014.
- [30] Travis CI GmbH, *Travis CI*. Available: <https://travis-ci.org>.
- [31] T. Holowaychuk, *Mocha Official Site*, 2011. Available: <https://mochajs.org>.
- [32] *How npm Works - What is a module?* Available: <https://docs.npmjs.com/how-npm-works/packages#what-is-a-module>. Accessed 10 Jul 2016.
- [33] K. Anantheswaran, *Istanbul Js*. Available: <https://gotwarlost.github.io/istanbul>. Accessed 10 Jul 2016.
- [34] M. Fowler, *Test Coverage*, 2012. Available: <http://martinfowler.com/bliki/TestCoverage.html>.
- [35] D. Wells, *Integrate Often - Extreme Programming*, 1999. Available: <http://www.extremeprogramming.org/rules/integrateoften.html>.
- [36] M. Fowler, *Continuous Integration*, 2006. Available: <http://martinfowler.com/articles/continuousIntegration.html>.
- [37] Free Software Foundation, Inc, *GNU Affero General Public License*, 2007. Available: <https://www.gnu.org/licenses/agpl-3.0.html>.
- [38] T. Preston-Werner, *Semantic Versioning 2.0.0*. Available: <http://semver.org>.