Universidad de Valladolid

Máster universitario

Ingeniería Informática

Trabajo Fin de Máster

Optimización, Refactorización y Rearquitectura de Crossdroads 2.0

Realizado por **ROBERTO BAHILLO ORTEGO**

✠✠✠

Universidad de Valladolid

19 de septiembre de 2025

Tutor: YANIA CRESPO GONZÁLEZ-CARVAJAL

DAVID ESCUDERO MANCEBO

Universidad de Valladolid

Máster universitario en Ingeniería Informática

Dra. YANIA CRESPO GONZÁLEZ-CARVAJAL y Dr. DAVID ESCUDERO MANCEBO, profesores del departamento de DEPARTAMENTO DE INFORMÁTICA, áreas de LENGUAJES Y SISTEMAS INFORMÁTICOS y CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL.

Exponen:

Que el alumno D. ROBERTO BAHILLO ORTEGO, ha realizado el Trabajo final de Máster en Ingeniería Informática titulado "Optimización, Refactorización y Rearquitectura de Crossdroads 2.0".

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección del que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Valladolid, a 19 de septiembre de 2025

Vº. Bº. del Tutor:	Vº. Bº. del co-tutor:
Dra. YANIA CRESPO	Dr. DAVID ESCUDERO MANCEBO

Resumen

El cambio climático y el desarrollo sostenible de las economías siguen siendo temas muy importantes a día de hoy. Como respuesta a estas preocupaciones se creó la aplicación Crossroads 2.0, una aplicación web educativa y gamificada para ayudar a los ciudadanos a concienciarse de las decisiones políticas en los temas previamente mencionados.

El desarrollo inicial de dicha aplicación tuvo lugar durante trabajos previos de otros estudiantes; si bien éstos concluyeron dejando algunas carencias. El objetivo principal de este trabajo fin de máster es solucionar los problemas de sobrecarga del servidor, producidos por la arquitectura cliente-servidor en algunas llamadas, intentando modificar lo menos posible la funcionalidad percibida por los usuarios.

Cumplidos estos objetivos, se obtiene una nueva versión del juego que requiere un menor uso del servidor por cliente conectado, mejorando la eficiencia de la misma y permitiendo que aumente el número de usuarios concurrentes.

Descriptores

Crossroads 2.0, evaluación de rendimiento, aplicación web educativa, rearquitectura, websocket, cliente-servidor

Abstract

Climate change and sustainable development of the economy are still crucial topics nowadays. To answer these concerns, Crossroads 2.0 was created, an educational gamified web application built to help citizens get an understanding about how political decisions shape these previously mentioned subjects.

The initial development of this application took place during previous works by other students. These concluded, however, leaving some gaps. The main objective of this master's thesis is to solve the server overload, due to the client-server architecture calls, trying not to change the fuctionality perceived by the user.

Once these objectives have been achieved, a new game version is obtain that lessens the load on the server per connected client, enhancing the server efficiency and allowing for additional concurrent users.

Keywords

Crossroads 2.0, performance testing, educational web application, rearchitecture, websocket, client-server

Agradecimientos

Agradezco a mis amigos haberme apoyado a lo largo de todo este proyecto y de haberme prestado su tiempo y sus ideas para haber podido darle forma.

Índice general

Índice de figuras

Índice de tablas

Indice de fuentes de código

1: Introducción

1.1. Contexto

Resulta innegable que, en la sociedad actual, el cambio climático es uno de los asuntos más relevantes. Según los diversos Eurobarómetros, donde más de tres cuartas partes de los ciudadanos europeos pensaban que es un problema muy serio en el 2023 [4]; el 78 % de los encuestados en el 2024 pensaban que los problemas ambientales tienen un efecto directo en su salud y vida diaria; o entre los jóvenes este tema se encuentra en la cuarta posición, siendo lo que más les preocupa a un 26 % de los encuestados en el 2025 [5]. Teniendo en cuenta estos datos, tanto gobiernos como empresas privadas están buscando medidas para concienciar a la población y clientes, y paliar los efectos que este podría tener en el futuro.

Como parte de este conjunto de acciones de mitigación y concienciación, la Unión Europea seleccionó para su financiación el proyecto LOCOMOTION [18] como parte del programa Horizonte 2020.Una de las propuestas de este proyecto consiste en el desarrollo de la aplicación web educativa gamificada **Crossroads 2.0**, con el fin de enseñar en las escuelas el impacto que tienen diversas decisiones políticas en el ámbito del cambio climático y en el crecimiento económico. Mediante un conjunto de IAMs (*Integrated Assessment Models*, Modelos de Evaluación Integrados) se pretende mostrar a los usuarios cómo sí es posible tener un crecimiento económico, medido en la aplicación a través del PIB, sin tener que sacrificar en el ámbito climático, evaluado como aumento de la temperatura.

Tras la realización de los desarrollos previos, y teniendo en cuenta el TFM de Manuel Alda Peñafiel, titulado "Definición y automatización de pruebas de carga y escalabilidad para una aplicación web colaborativa" [1], en el que se basa este trabajo, se pretende mejorar el rendimiento de la aplicación debido a los problemas de sobrecarga del servidor que se observaron a la finalización de dicho TFM. Por ello, como continuación de su trabajo, se propone analizar el problema, buscar cómo la industria está resolviendo problemas similares, y comprobar tras la aplicación de de estas soluciones si se ha conseguido reducir esta sobrecarga.

De esta manera, este trabajo estará compuesto por tres partes, una inicial donde se plantea profundizar en el problema buscando las causas de este, buscar cómo la

industria resuelve este tipo de problemas, aplicando estas soluciones de tal manera que la funcionalidad actual de la aplicación siga siendo la misma, y una comprobación de los test de carga para confirmar si estos cambios han sido útiles y su comparación con los que obtuvo Alda Peñafiel

1.2. Motivación

Dentro de los múltiples cambios que pueden aplicarse para incrementar la calidad de la aplicación objeto de este trabajo, se ha buscado mejorar la experiencia del usuario con respecto a los tiempos de espera. La aplicación puede tener un máximo de 100 jugadores y un maestro de juegos (o anfitrión de sala), por cada sala que haya activa, pudiendo haber varias salas activas en un momento dado. Teniendo en cuenta esto, es imprescindible encontrar un sistema que nos permita tener una gran concurrencia de usuarios al mismo tiempo, haciendo peticiones al servidor, sin que esto aumente de manera significativa la carga de este, pudiendo dar respuesta a todos en un tiempo adecuado.

En este trabajo, nos centraremos especialmente en una sala con 100 usuarios, siguiendo las mismas métricas de Alda Peñafiel en su TFM, considerándolo un éxito si hay una mejora en las métricas usadas al respecto.

1.3. Objetivos

A la finalización de este trabajo se pretenden alcanzar los siguientes objetivos:

- Revisar y analizar el problema de sobrecarga de la aplicación.
- Encontrar y aplicar una solución que permita la concurrencia de un número muy elevado de usuarios sin modificar la funcionalidad actual de la aplicación.
- Revisar las pruebas de carga y comprobar si se aprecia la mejora tras la aplicación de los cambios.

Con estos objetivos en mente se pretende allanar el camino para nuevos desarrollos y mejoras que se puedan realizar a futuro en la aplicación.

1.4. Estructura de la Memoria

Para mejorar la legibilidad de esta memoria, se presentan los capítulos que la componen junto con un breve resumen de los temas que se van a tratar en estos.

Capítulo 2: Acerca de Crossroads 2.0: Explicación de la aplicación y el estado en del que se parte, junto con los resultado obtenidos del TFM de Alda Peñafiel y los objetivos que se planean alcanzar en este trabajo.

Capítulo 3: Conceptos Teóricos: Explicación de los conceptos teóricos necesarios para comprender todos los temas que se van a tratar, así como las tecnologías usadas, tanto actualmente por la aplicación como las que se han encontrado para poder resolver los problemas que actualmente presenta y por qué se selecciona una tecnología frente a otra.

Capítulo 4: Técnicas y Herramientas: En este capítulo se pretende explicar la implementación de la tecnología usada, mostrando herramientas en las que se apoya este proyecto para la implementación de la solución

Capítulo 5: Aspectos Relevantes del Desarrollo: Se explican los conceptos relevantes de la solución, problemas que se encontraron en la implementación

Capítulo 6: Conclusiones y Líneas de Trabajo Futuro: Se muestran los resultados de las pruebas obtenidos, su interpretación y posibles líneas de trabajo futuro que se abrirían tras el trabajo.

2: Acerca de Crossroads 2.0

En este capítulo se pretende explicar el funcionamiento inicial de la aplicación, para ayudar al lector a entender la situación original de la que partió el proyecto; el problema detectado por Manuel Alda [1]; y los objetivos principales que se plantean.

2.1. Crossroads 2.0

Crossroads 2.0: Aplicación web

Como se ha comentado en otras secciones, Crossroads 2.0 es una aplicación web educativa gamificada, donde los usuarios desarrollarán de manera simulada el trabajo de políticos, formando grupos para responder una serie de preguntas (de carácter económico, social, biológico y energético) y tratar de alcanzar un consenso en las respuestas, con el fin de obtener unos resultados finales en forma de gráficas y valores, que indican la evolución del PIB y el calentamiento global (mediante medición del CO2) con respecto a las preguntas obtenidas y al conjunto de IAMs que usa la aplicación. Estos valores pueden ser después compartidos con el resto de grupos, y discutidos de forma que ayude a su comprensión y a educar sobre la importancia que las decisiones políticas tienen en estos ámbitos. De esta forma, el objetivo principal de la aplicación es ayudar a la gente a tomar consciencia de la repercusión e implicaciones de estas cuestiones, haciéndolas mas visibles y accesibles a un usuario promedio mediante una aplicación interactiva. La aplicación se puede probar actualmente aquí (http://virtual.lab.inf.uva.es:20262).

Dinámica de uso

Crossroads 2.0 está compuesto por 2 aplicaciones distintas. Por una parte, existe una aplicación web, que, si bien está fuera del alcance del presente proyecto, debe tenerse en cuenta para el desarrollo puesto que usan el mismo *backend*. Esta aplicación está diseñada para que un mismo usuario cree la sala y juegue solo. Por otra parte, la aplicación objeto de este proyecto, que, aunque también es una aplicación web, se diferencia de la anterior al estar pensada para ser utilizada en multijugador.

En la aplicación web se distinguen 2 tipos de usuarios

Usuario Moderador, Maestro o Host: Es un usuario registrado en la aplicación. Este usuario tiene la capacidad de crear salas y controlar y modificar dichas salas.

Usuario jugador, Invitado o Guest: Este es un usuario no registrado en la aplicación. Accede a las partidas mediante un código que se genera al crear la sala, formando grupos antes de empezar la partida. Una vez asignado el grupo, los jugadores que pertenecen al mismo grupo, tienen que llegar a consensos para responder a las distintas decisiones políticas a las que les expone el juego para poder mostrar unos resultados finales que indican el estado del planeta tras aplicar dichas decisiones.

Lo primero que se encuentra cualquier tipo de usuario al acceder a la aplicación es la página de bienvenida, mostrada en la Figura 2.1. Desde aquí podremos separar los usuarios **Jugadores**, de los usuarios **Moderadores** dependiendo de sus acciones. Teniendo en cuenta esto, las dinámicas de los distintos usuarios se definen de la siguiente forma.

Figura 2.1: Página principal de Crossroads 2.0

Moderador

Lo primero que tiene que hacer todo moderador es registrarse como tal en la página. Para ello pulsará el enlace de *Registrarse como moderador* que se ve en la parte inferior de la Figura 2.1. Esto mostrará una nueva página (Figura 2.2), donde el usuario deberá introducir los datos correspondientes.

Figura 2.2: Formulario de creación de cuenta para Moderador

Una vez rellenado el formulario y creada la cuenta de **Moderador**, el usuario será redirigido automáticamente al menú de selección para **Moderadores** (Figura 2.3), donde, entre otras opciones, el usuario podrá buscar entre las salas que haya creado, estén siendo usadas ahora mismo o hayan sido partidas terminadas; crear una sala nueva para una nueva partida; o cerrar la sesión

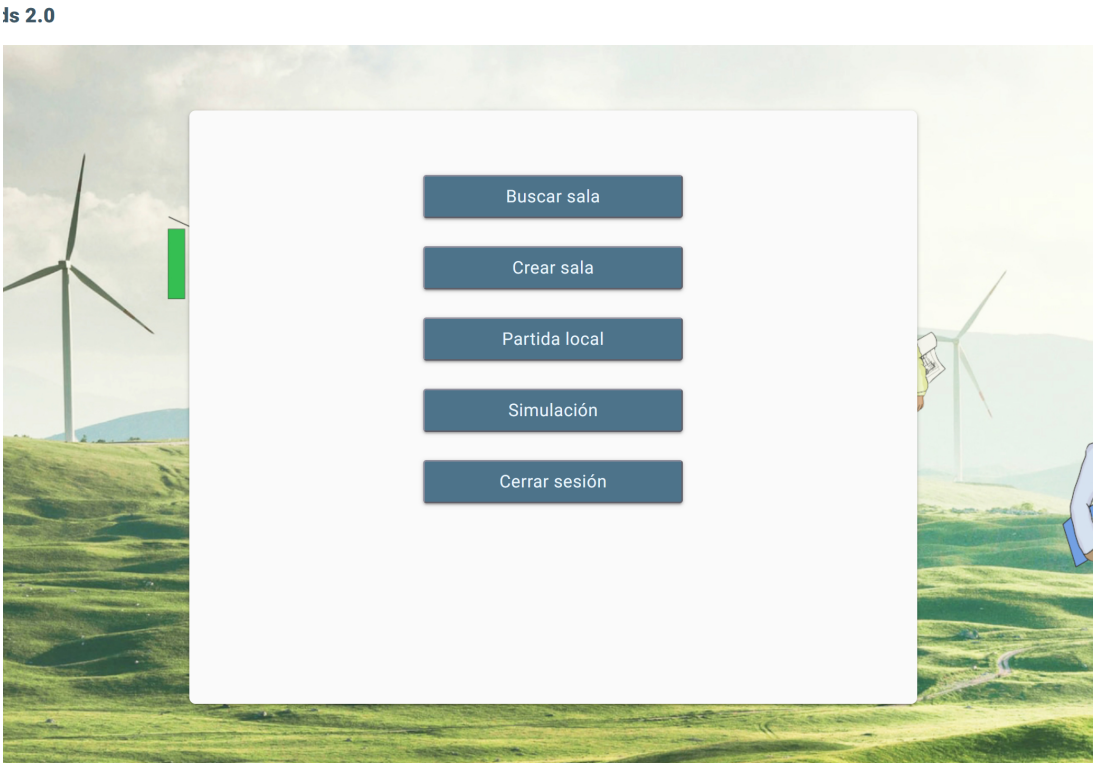

Figura 2.3: Menú de selección de opciones de moderador

Si el Moderador escoge **Crear sala**, pasará a las opciones de creación de sala como se muestra en la Figura 2.4. Aquí podrá elegir el número de rondas, cuántos grupos van a formar parte de la partida, y por cuánta gente está conformado cada grupo. Tras esto, el **Moderador** pasará a la segunda vista de la creación de sala. Como se muestra en la Figura 2.5, en esta vista la sala ya ha sido creada. Aparecen las opciones seleccionadas de la sala, como el nombre, la cantidad de personas por grupo y grupos y el número de rondas. También se genera un código único para cada sala, el cual tiene que ser pasado a los distintos jugadores que vayan a participar en la partida, y aparece la cantidad de jugadores actuales en la sala. En este punto, los **Jugadores** se unen a la partida con el código de la sala como se explicará más adelante (Figura 2.7). Una vez estén todos los jugadores, se puede empezar la partida.

Figura 2.4: Creación de la sala para la partida

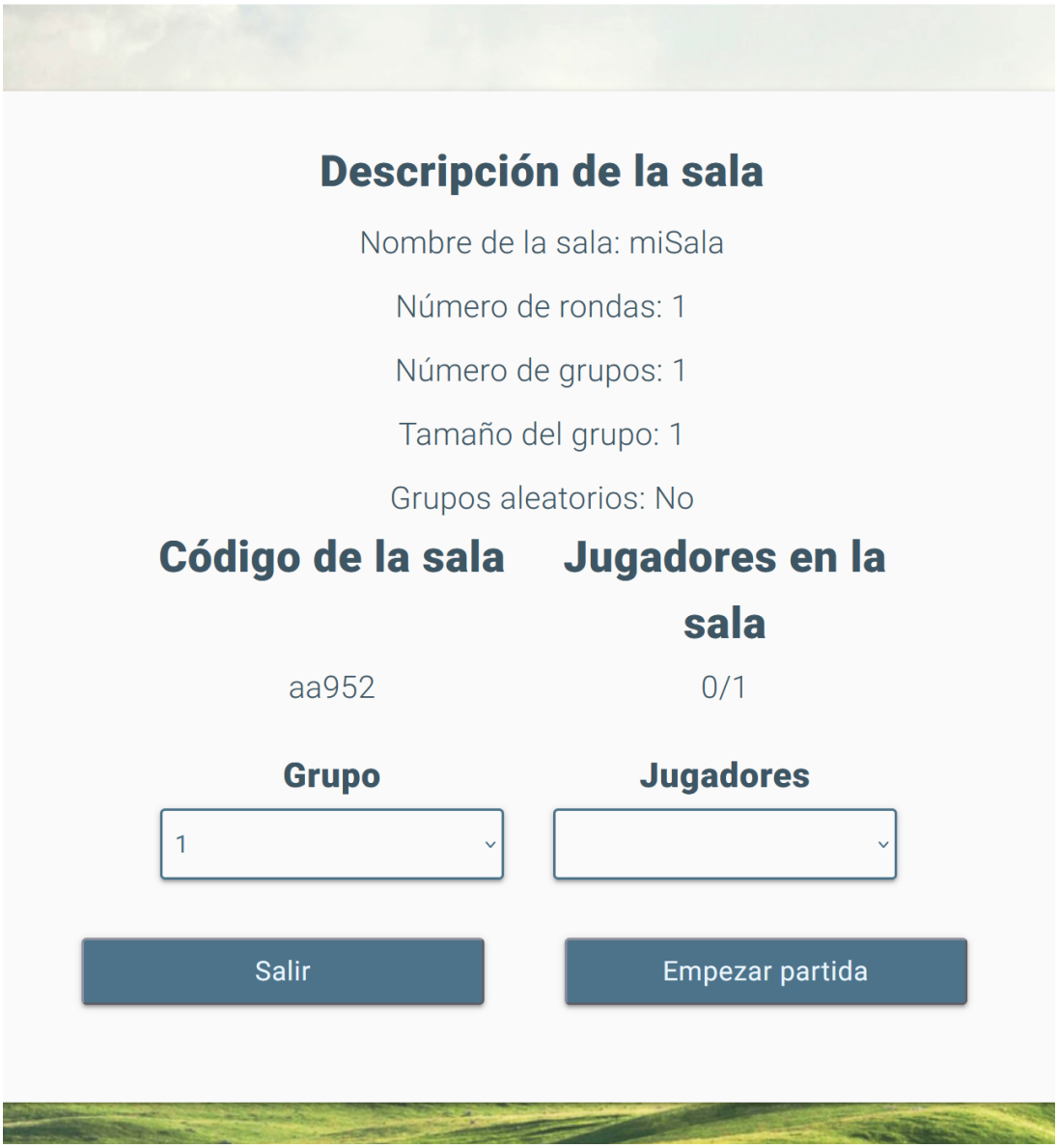

Figura 2.5: Sala creada

Empezada la partida, al **Moderador** se le muestra un panel de control interactuable (Figura 2.6). Al pulsar en los distintos grupos, se puede ver el estado de los diferentes miembros del grupo, bien respondiendo preguntas, resolviendo conflictos, o viendo los resultados finales. De manera similar, al pulsar sobre las distintas rondas, el **Moderador** puede ver el estado de los grupos de manera similar al de los jugadores por cada grupo.

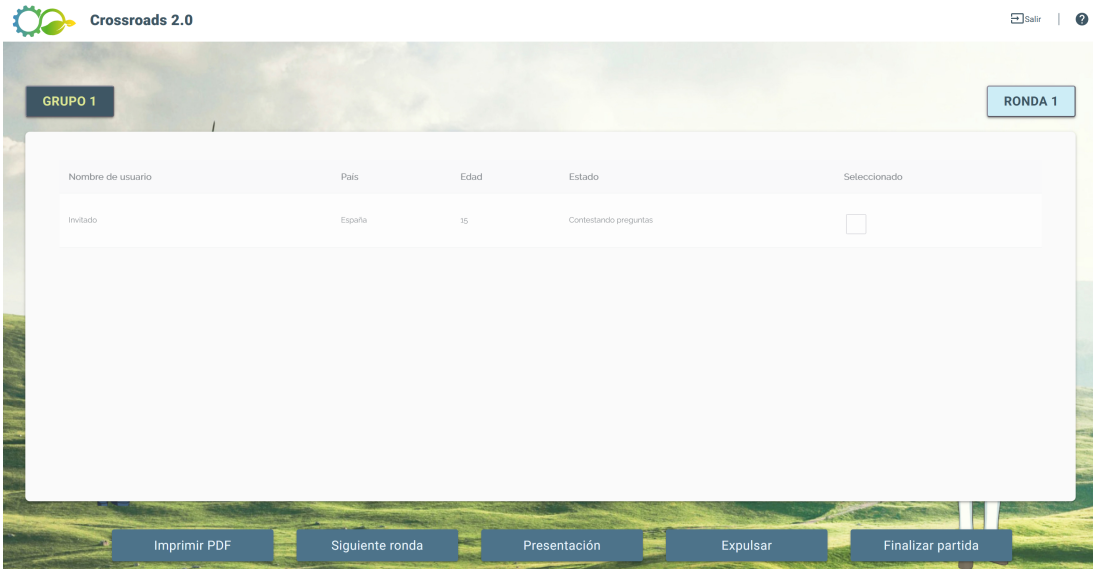

Figura 2.6: Panel de control del moderador

En el panel de control, el Moderador puede también terminar la ronda en cualquier momento, pulsando el botón de siguiente ronda. También puede expulsar jugadores de la partida, finalizar la partida o imprimir un PDF que devuelve los resultados finales de la partida.

Jugador

Como se ha explicado antes, los **Jugadores** también aterrizan en la página de bienvenida de la aplicación (Figura 2.1), donde, a diferencia del **Moderador**, seleccionarán el botón de participar, el cuál les llevará a la página de (Figura 2.7), donde, introduciendo el código de la sala, los **Jugadores** accederán a la página relativa a los datos del jugador (Figura 2.8). Aquí, los **Jugadores** tendrán que añadir un nombre único en la sala (no puede haber 2 jugadores con el mismo nombre por sala), su país y su edad.

Figura 2.7: Acceso a la sala mediante el código de sala

Figura 2.8: Página de datos del jugador

Al entrar se les mostrará la opción de ver un video explicativo de la aplicación y pasar a la elección del rol y el grupo (Figura 2.9). Los roles están relacionados con cuatro entornos de trabajo que tienen influencia sobre el cambio climático. Estos son **Economía**, **Energía**, **Biología** y **Sociología**. Cada rol tiene ciertas ayudas con consejos y recomendaciones en las distintas preguntas si estas están relacionadas con ellos. Tras seleccionar el rol, el **Jugador** puede decidir unirse a uno de los grupos disponibles, donde puede ver una lista de los jugadores que hay en cada uno.

Figura 2.9: Selección del grupo del jugador

Cuando el **Moderador** empieza la partida, a los **Jugadores** se les presenta la pantalla de preguntas del juego (Figura 2.10). A los **Jugadores** se les muestra una serie de preguntas, y entre ellos deberán escoger la respuesta que consideren más adecuada para cada caso (Figura 2.11). En este punto no es necesario que lleguen a un acuerdo entre sí, de esta manera se les da la libertad a los grupos para que usen distintas estrategias para resolver las cuestiones a las que se enfrentan.

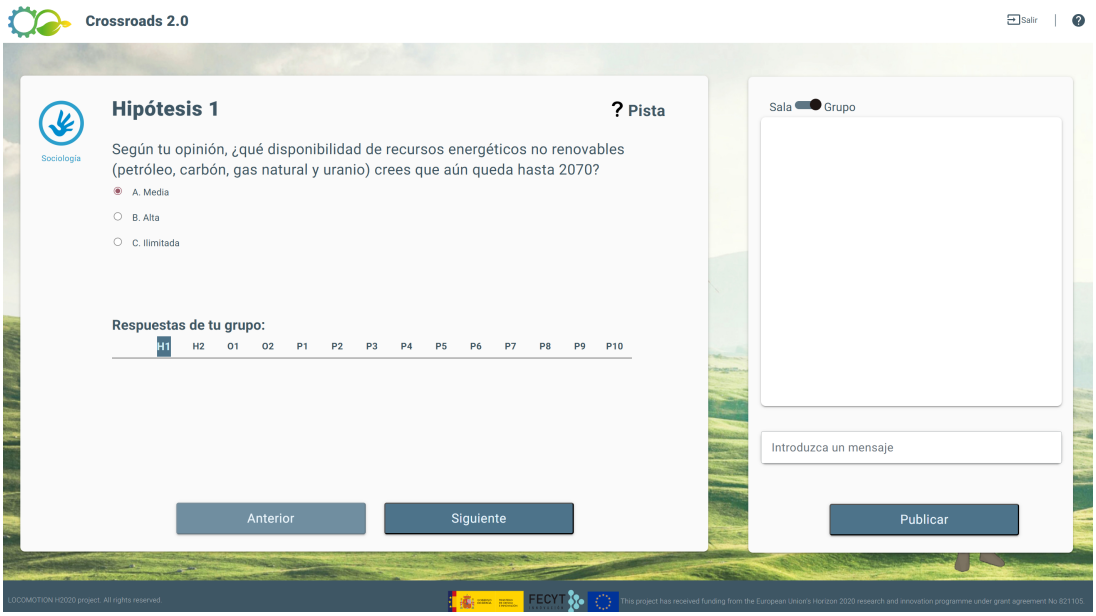

Figura 2.10: Página primera del juego

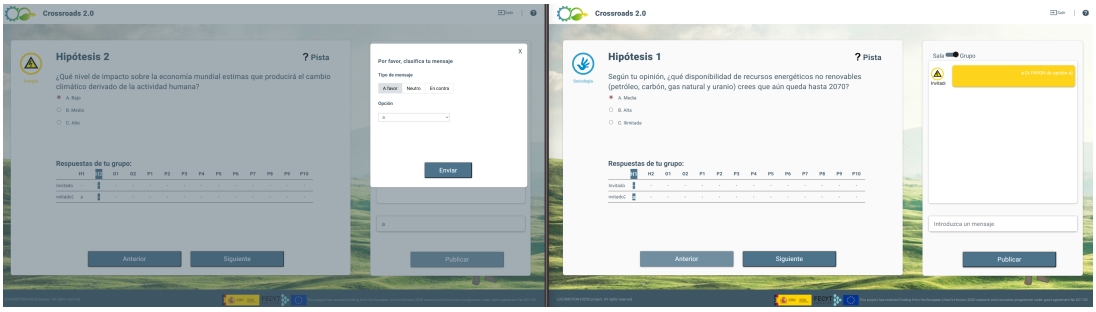

Figura 2.11: Jugadores seleccionando respuestas

Una vez el grupo ha resuelto todas las preguntas, tal y como se ha mencionado antes, en este punto no tiene por qué haber alcanzado unanimidad en todas las respuestas (Figura 2.12), pero para poder obtener los resultados, el grupo deberá de llegar a continuación a un acuerdo para cada pregunta, respondiendo todos lo mismo. Si el jugador pulsa sobre las **x** rojas, le llevará a la vista de la pregunta conflictiva (Figura 2.13), donde se le mostrará otra vez la pregunta, sus respuestas, y a mayores una tabla con los distintos valores que han elegido los miembros del grupo, indicando cuánta gente ha votado a favor, en contra, o neutral a las distintas respuestas.

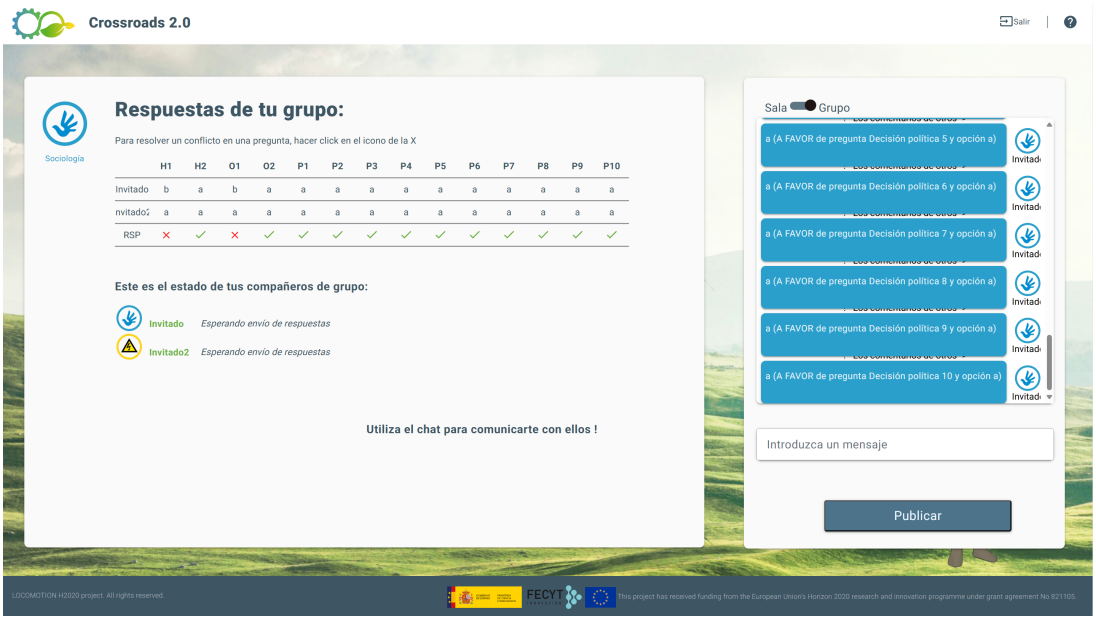

Figura 2.12: Resolución de una respuesta conflictiva

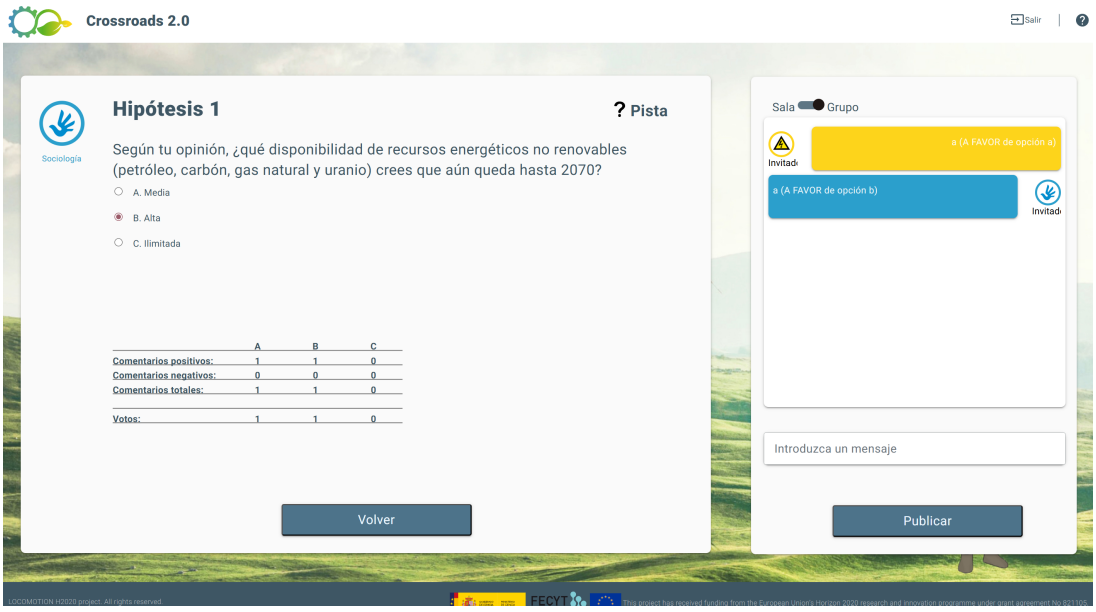

Figura 2.13: Vista para la resolución de conflictos

Una vez el grupo haya alcanzado un consenso para todas las respuestas, el Crossroads les calculará y mostrará los resultados finales de cada ronda (Figura 2.14) o partida. Aquí los **Jugadores** podrán ver una simulación de la evolución del PIB y de la temperatura del planeta teniendo en cuenta sus respuestas. También se muestran una serie de recomendaciones para mejorar esos resultados, y en la parte inferior unos gráficos de barras

calculando distintos valores para cada **Jugador** del grupo, tanto el valor de sus respuestas, como aspectos sociales del grupo.

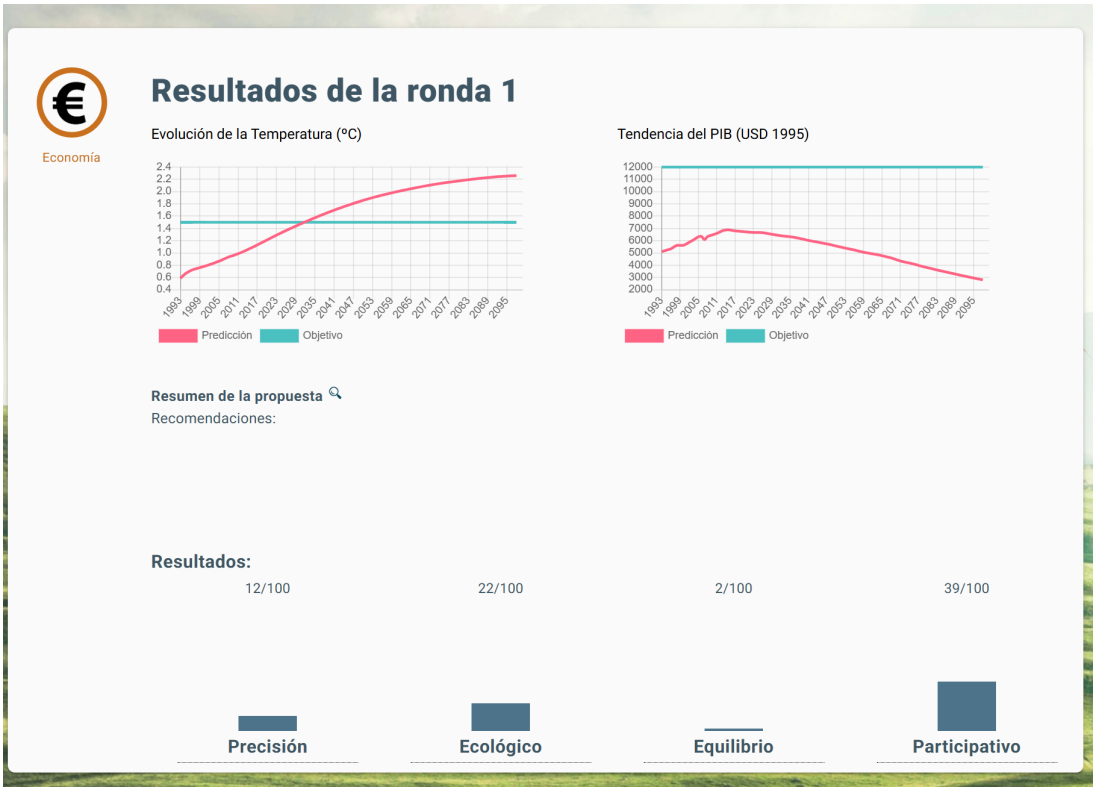

Figura 2.14: Resultados finales de la partida

2.2. Estado Original y Análisis del problema

Originalmente, la aplicación de Crossroads 2.0, se planteó utilizando una arquitectura cliente-servidor para la interacción entre la página que ven los usuarios y todos los datos que esta maneja. No obstante, se encontraron contratiempos a la hora de resolver ciertas llamadas de datos, ya que estas requerían estar constantemente actualizadas para poder funcionar correctamente, siendo algunas de estas, el chat entre los Jugadores, o el estado en el que se encuentran estos para darle la información necesaria al Moderador. Esto se resolvió mediante un temporizador para que estas llamadas se hicieran de manera recurrente cada intervalo de tiempo.

Como pudo verse en el trabajo de Alda Peñafiel, se producía una sobrecarga considerable en el servidor a medida que aumentaba el número de usuarios, haciendo que fuera inviable poder escalarlo de manera consistente, dado que los tiempos de respuesta siguen un crecimiento exponencial. Esta sobrecarga acababa produciendo errores e inconsistencias en el servidor que dificultan el uso adecuado de la aplicación.

2.3. Objetivos

Teniendo en cuenta lo mencionado en los anteriores puntos de este capitulo, podemos encontrar los siguientes objetivos principales que han de cumplirse en este trabajo:

- Localizar las principales llamadas que están afectando al rendimiento de la aplicación.
- Encontrar una solución que pueda aplicarse a la aplicación.
- Aplicar la solución.
- Comprobar si la solución mejora la situación en el servidor.

3: Marco teórico y Trabajos relacionados

3.1. Arquitectura Cliente-Servidor

La arquitectura cliente-servidor permite la comunicación y el intercambio de datos entre distintas aplicaciones de uno o varios servidores, dividiendo así el modelo en dos partes [31]

Cliente: Aplicación que realiza peticiones de servicios, como obtención de datos, realización de cálculos... Siempre es el que empieza la interacción

Servidor: Aplicación que procesa las peticiones del cliente o realiza las acciones específicas

Para su funcionamiento, el cliente y el servidor pueden residir dentro de la misma máquina o en diferentes dispositivos conectados a través de una red.

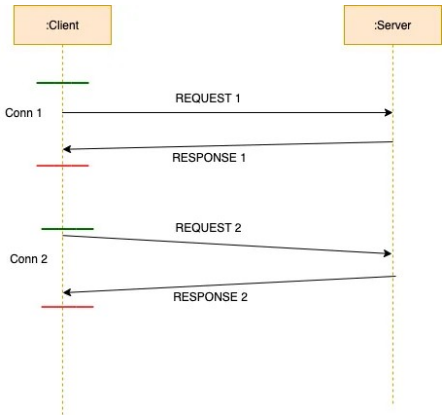

Figura 3.15: Diagrama de Secuencia del protocolo HTTP en una arquitectura cliente-servidor [21]

La figura 3.15 muestra el diagrama de secuencia del protocolo HTTP en el modelo cliente-servidor tradicional. Como se ha comentado, el cliente siempre es el que empieza la interacción, y cada vez que esta se crea, es necesario añadir al mensaje unas cabeceras con diversas opciones de la llamada. A su vez, el servidor crea una respuesta con los datos solicitados por el cliente. La respuesta puede tener distintos códigos estándar dependiendo de lo que haya sucedido en el lado del servidor. Algunas de las más comunes son: [23]

Códigos 1xx: Respuestas informativas por parte del servidor:

- **100 Continue**: todo está correcto. El cliente debe continuar con la solicitud o ignorarla si ya está terminada.
- **101 Switching protocols**: Este código se envía en respuesta a un encabezado de solicitud Upgrade por el cliente e indica que el servidor acepta el cambio de protocolo propuesto por el agente de usuario.
- **102 Processing**: El servidor aún está procesando la información que ha solicitado el cliente.

Códigos 2xx: Respuestas satisfactorias dependiendo de método usado. (GET, HEAD, PUT/PUSH, TRACE)

- **200 Ok**: La solicitud ha tenido éxito. El significado de un éxito varía dependiendo del método HTTP
- **201 Created**: La solicitud ha tenido éxito y se ha creado un nuevo recurso como resultado de ello. Ésta es típicamente la respuesta enviada después de una petición PUT.
- **202: Accepted**: La solicitud se ha recibido, pero aún no se ha procesado. No es vinculante, ya que HTTP no permite enviar posteriormente una respuesta asíncrona que indique el resultado de la solicitud. Está pensado para los casos en que otro proceso o servidor maneja la solicitud, o para el procesamiento por lotes.

Códigos 3xx: Redirecciones

- **301 Moved Permanently**: La URI que se ha solicitado ha sido permanentemente movida a otra. La nueva URI puede venir como respuesta a esta.
- **302 Found**: Este código de respuesta significa que el recurso de la URI solicitada ha sido cambiado temporalmente. Nuevos cambios en la URI serán agregados en el futuro. Por lo tanto, la misma URI debe ser usada por el cliente en futuras solicitudes.
- **308 Permanent Redirect**: Significa que el recurso ahora se encuentra permanentemente en otra URI, especificada por la respuesta de encabezado HTTP *Location*:. Tiene la misma semántica que el código de respuesta HTTP **301** Moved Permanently, con la excepción de que el agente usuario no debe cambiar el método HTTP usado: si un POST fue usado en la primera petición, otro POST debe ser usado en la segunda petición.

Códigos 4xx: Errores de Cliente

- **401 Unauthorized**: Es necesario autenticar para obtener la respuesta solicitada. Esta es similar a 403, pero en este caso, la autenticación es posible.
- **403 Forbidden**: El cliente no posee los permisos necesarios para cierto contenido, por lo que el servidor está rechazando otorgar una respuesta apropiada.
- **404 Not Found**: El servidor no pudo encontrar el contenido solicitado. Este código de respuesta es uno de los más famosos dada su alta ocurrencia en la web.

Códigos 5xx: Errores de Servidor

- **500 Internal Server Error**: El servidor ha encontrado una situación que no sabe cómo manejarla.
- **503 Service Unavailable**: El servidor no está listo para manejar la petición. Causas comunes puede ser que el servidor está caído por mantenimiento o está sobrecargado.
- **504 Gate Timeout**: Esta respuesta de error es dada cuando el servidor está actuando como una puerta de enlace y no puede obtener una respuesta a tiempo.

3.2. HTTP Polling

El *HTTP Polling* funciona de manera muy similar a HTTP tradicional, la diferencia principal radica en la necesidad de asegurarse de que un recurso está constantemente actualizado. Para conseguir esto, existen dos variantes diferentes [8]:

Short Polling: El cliente lanza las llamadas a intervalos definidos, esto asume que el recurso que se pide puede no estar disponible, por lo que el servidor puede devolver llamadas vacías. Cuando se cumple el intervalo de tiempo, se vuelve a solicitar el recurso.

Long Polling: El cliente espera a que el servidor responda con la llamada antes de volver a lanzarla otra vez, por lo que los mecanismos en los que se basa están pensados para que el servidor mande siempre una respuesta manteniendo abierta la conexión con el cliente. Cuando el servidor contesta, se vuelve a solicitar el recurso, bien de manera inmediata o tras un intervalo de tiempo establecido.

Con esto se puede ver que, teniendo en cuenta cómo funciona HTTP, ambos generarán una gran carga en la red y el servidor debido al *overhead* que se produce, aunque *Long Polling* generará significativamente menos puesto que esperará por una nueva respuesta del servidor antes de volver a pedir el recurso.

3.3. Arquitectura publish-subscriber

El patrón arquitectónico Publish-Subscribe [15] es un patrón de comunicación donde los emisores de mensajes (publishers) no envían mensajes directamente a receptores específicos (subscribers), sino que publican mensajes sin conocer qué suscriptores los recibirán. Elementos fundamentales del patrón:

- **Publishers (Publicadores)**: Crean y publican mensajes.
- **Subscribers (Suscriptores)**: Se suscriben interesados en determinada información, para recibir ciertos mensajes.
- **Message Broker**: Sistema intermediario que gestiona la distribución de mensajes.

Recientemente se introduce el término de *topics* [3] para referirse a categorías/temas de mensajes en los que el suscriptor está interesado, es decir, el tipo de información publicada por el publicador que le interesa.

Los beneficios que se pretende obtener con la aplicación del patrón publish-suscribe son:

- **Desacoplamiento**: Los publishers y subscribers no se conocen directamente entre sí. Esto permite que el sistema sea más flexible y escalable.

- **Comunicación asíncrona**: Los mensajes se envían sin esperar respuesta inmediata, mejorando el rendimiento.
- **Flexibilidad y Escalabilidad**: Puedes agregar nuevos publishers o subscribers sin afectar el resto del sistema.

También se debe vigilar en el patrón que su aplicación podría traer consigo como desventaja introducir complejidad adicional en el sistema y la posible pérdida de mensajes si no se configura correctamente.

3.4. WebSocket

La arquitectura publish-subscriber permite el establecimiento y mantenimiento de una conexión de datos abierta entre uno o varios clientes y un servidor. De manera similar a la arquitectura cliente-servidor mediante HTTP, desde el servidor se habilitan unos *endpoints* usando WebSocket (abreviado aquí como WS) a los que los clientes se suscriben para recibir y enviar mensajes. De manera similar a un patrón Observador, en el momento en que se produce un cambio, este es distribuido a todos los clientes que estén escuchando en el *endpoint*.

Para establecer una conexión, como puede verse en la Figura 3.16, WebSocket consta de 2 pasos:

Handshake inicial: El cliente pide una actualización de protocolo de transmisión al protocolo WS si se estaba usando http o a WSS en caso contrario (Figura 3.17). Esto presenta los esquemas URI de ws y wss para WebSocket y WebSocket Secure [22]

Establecimientos de la conexión bidireccional de mensajes: El cliente y el servidor establecen una conexión bidireccional de mensajes, donde los cambios producidos en los datos del *endpoint* hace que el servidor propague el mensaje a todos los clientes que estén conectados. Estos cambios pueden venir de otro cliente o de otras llamadas que se realizan en el servidor

Es importante recalcar que la comunicación es bidireccional; sin necesidad de establecer nuevas conexiones, el cliente puede escuchar y producir cambios en el flujo de datos al que esté conectado mediante mensajes.

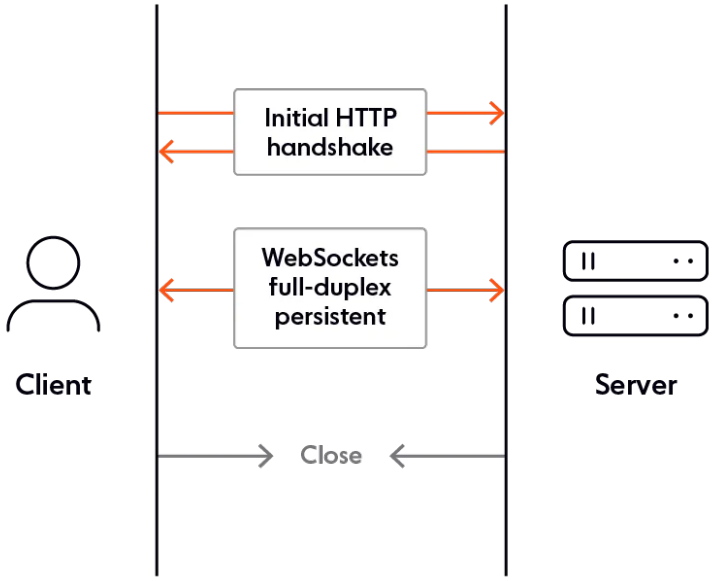

Figura 3.16: Estructura de una conexión WebSocket [8]

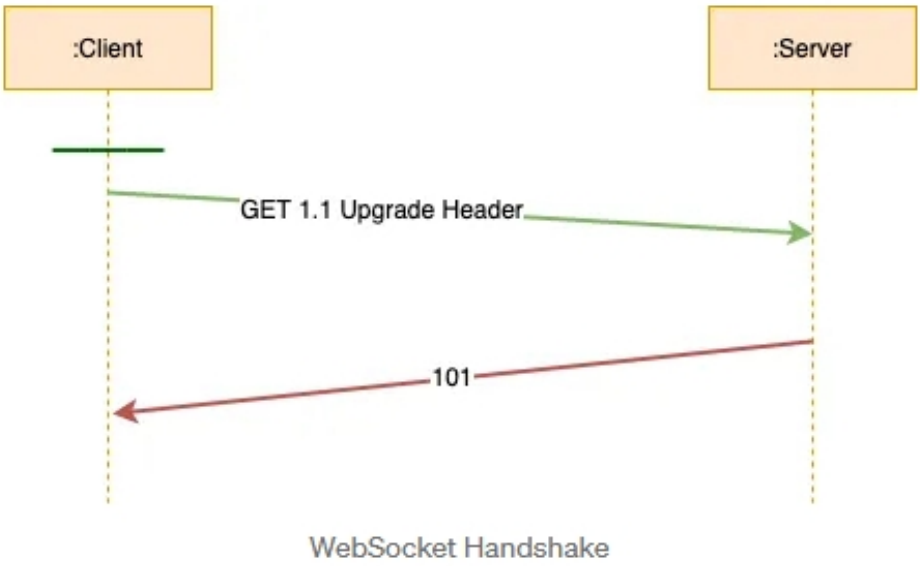

Figura 3.17: Handshake inicial [21]

El estándar de WebSocket [32], nos presenta con varias funciones que implementa. Las que más nos importan ahora mismo son

- *WebSocket(url, protocol)*: Creación del WebSocket
- *send(data)*: Envío de datos a la conexión. Los datos enviados pueden ser de tipo *String*, *Blob*, *ArrayBuffer* o *ArrayBufferView*
- *close(code, reason)*: Cierre de la conexión

Y unos estados para controlar cómo está la conexión [27], siendo estos importantes para controlar el flujo de respuesta de las funciones:

- CONNECTING (valor numérico 0)
- OPEN (valor numérico 1)
- CLOSING (valor numérico 2)
- CLOSED (valor numérico 3)

Para tener control de lo que pasa en la transmisión de los datos (send), WebSocket usa cuatro eventos distintos

- onopen
- onmessage
- onerror
- onclose

Un ejemplo que englobe el funcionamiento de los estados y los eventos sería el caso de uso para cuando se acaba de establecer la conexión y la recepción de un mensaje:

Conexión establecida:

1. El estado se cambia a *OPEN*.
2. Se actualiza el valor del atributo *extensions*.
3. Se actualiza el valor del atributo *protocol*.
4. Se lanza un evento *open* al objeto WebSocket.

Recepción de mensaje:

1. Si el estado no es *OPEN*, se lanza un return.

2. Se determina *dataForEvent* mediante el cambio producido en *type* y *binary type*. Esto indica si los datos recibidos van a ser de tipo *texto*, *blob* de datos o *arraybuffer*.
3. Se lanza un evento *message* para que sea capturado por el WebSocket

El protocolo WebSocket también pone a disposición el *frame* ***PING PONG***. En este *frame* un agente envía un mensaje **PING** y otro responde con un mensaje **PONG**. Esto puede ser usado como *keep-alive*, para analizar el estado de la red o como instrumento de latencia, entre otros. Los agentes pueden mandar **PING** o **PONG** no solicitados cuando deseen, por ejemplo, para descubrir conexiones fallidas o para mostrar métricas de latencia a los usuarios.

Códigos de estado

El protocolo también posee un conjunto de códigos de estado, que puede que sean mandados solamente al cerrar la conexión. Estos son enviados como una razón del cierre, pero las acciones que debe tomar el *endpoint* al respecto de estos códigos quedan sin definir por la especificación del RFC 6455 [2]. Los códigos puestos a disposición son los siguientes:

0-999: Estos códigos de estado no están siendo usados.

1000-2999: Los códigos de estado en este rango están reservados para el uso del protocolo:

- **1000**: Cierre normal del protocolo.
- **1001**: La conexión se está cerrando, bien por caída del servidor o porque se está navegando a otra página.
- **1002**: Se cierra la conexión debido a un error del protocolo.
- **1003**: El *endpoint* está cerrando la conexión debido a que ha recibido un tipo de dato que no puede aceptar.
- **1004**: Reservado para uso futuro.
- **1005**: Es un valor reservado y no debe ser usado como código para cerrar el *socket*. Se reserva para aplicaciones que esperan un código de estado y sirve para indicar que realmente no se ha mandado un código.
- **1006**: Valor reservado y que no debe usarse para cerrar un *socket*. Se reserva para aplicaciones que esperan un código de estado y sirve para indicar que la conexión se ha cerrado de manera anormal.
- **1007**: Lanzado por el *endpoint* para indicar que se cierra la conexión debido ha que ha recibido datos en el mensaje que no pueden interpretarse de acuerdo a este, por ejemplo texto que no esté codificado como UTF-8 en un mensaje de texto.
- **1008**: Código que indica que el *endpoint* está cerrando el *socket* debido ha que ha recibido un mensaje que viola su política. Este es un estado genérico que se manda cuando ningún otro se ajusta bien, o por si se *query* ofuscar la razón real.

1009: El *endpoint* está cerrando la conexión debido a que recibió un mensaje demasiado grande.

1010: El *endpoint* (cliente) está cerrando la conexión porque ha esperado que el servidor negociara una o más extensiones, pero no las ha recibido en el mensaje de respuesta del *handshake*. La lista de extensiones esperadas debería aparecer en la razón del *frame* de *Close*. Este código de estado no es usado por el servidor porque tiene la opción de fallar en el *handshake*.

1011: Este código es lanzado por el servidor al encontrar una condición inesperada que impide que se pueda completar la petición.

1015: Este código no debería ser usado. Se usa para las aplicaciones que esperan un código para indicar que la conexión ha sido cerrada debido a un fallo al realizar un *handshake* TLS.

3000-3999: Estos códigos están reservados para el uso de librerías, *frameworks* y aplicaciones.

4000-4999: Este intervalo está reservado para uso privado, por lo que no puede estar registrado.

3.5. Comparativa entre los protocolos revisados

Habiendo visto estos tipos de conexiones, se condensa la información más útil para poder elegir el protocolo que se va a usar, teniendo en cuenta que actualmente se está usando un **Short Polling** en la aplicación para resolver las respuestas [9] [10].

Parámetro	WebSocket	HTTP	Long Polling
Tipo Comunicación	Bidireccional	Petición-respuesta	Petición-respuesta
Tipo Conexión	Persistente	No persistente	Persistente
Overhead	Bajo, solo el handshake	Alto, por cada petición	Muy alto, las peticiones se repiten
Latencia	Baja	Alta debida a la preparación por cada petición	Alta debida a la preparación por cada petición
Caso de uso	Aplicaciones en tiempo real, juegos, chat	Navegación en internet, retorno de documentos	Actualización constante de datos de aplicaciones, chats
Facilidad de implementación	Difícil	Fácil	Moderado
Escalabilidad	Alta, se mantiene eficiente para varios clientes	Moderada, debido al consumo elevado de recursos	Moderada, debido al alto consumo de recursos
Estandarización	Soporta TLS (wss://)	Soporta TLS (https://)	Soporta TLS (https://)
Interoperabilidad	Soportado por la mayoría de navegadores modernos	Universalmente soportado	Universalmente soportado

Tabla 3.1: Comparación protocolos revisados

A la vista de la comparativa realizada en la Tabla 3.1 entre las diferentes opciones, la opción que más interesa implementar en este punto es WebSocket, aunque sea esta la más difícil. Esta decisión se justifica al ser una tecnología cuyos usos principales son los de chat, juegos, y acciones en tiempo real, con una baja latencia y que mantiene su eficiencia con varios clientes, permitiendo que pueda escalar más fácilmente que otras. Para facilitar su implementación se busca una librería que pueda asistir con los detalles de WebSoket, encontrando en este caso **STOMP**.

3.6. Refactorización y Rearquitectura

La refactorización de la arquitectura o rearquitectura suelen tener orígenes diversos, desde redefinición o modificación de requisitos, cambios en la infraestructura o tecnología en la que se basa el proyecto o como consecuencia de errores o malas decisiones que deben ser corregidos, cambios en las estructuras organizaciones o procesos de negocios, o nuevas legislaciones [20]. De esta manera, se proponen cambios que cambien la manera en la que una aplicación realiza su tarea, vengan estos cambios desde dentó o desde fuera de la empresa.

Por su parte, la refactorización de código es una técnica que busca alterar la estructura interna de este, sin que se vea afectado su comportamiento ni funciones [29] [7]. Con esto se busca que el código mejore su legibilidad, mantenibilidad, eficiencia o que se reduzca su complejidad. También puede usarse para mejorar la arquitectura del código, modificando su estructura para alinearla mejor con los principios de diseño y las mejores prácticas, como aplicando patrones de diseño o eliminando malas prácticas arquitectónicas y malos olores.

Para conseguir estos fines existen varias técnicas. Algunas de estas son:

Renombrar (Rename): cambiar el nombre de variables, funciones, clases o módulos para que sean más descriptivos. Esto aumenta la legibilidad del código.

Extracción de Método (Extract Method): extraer un bloque de código en una nueva función o método para hacer el código más legible y reducir la duplicación.

Extracción de Variable (Extract Variable): crear una variable para almacenar una expresión compleja o repetitiva, lo que hace que el código sea más claro y más fácil de entender.

Mover Método (Move Method): mover un método de una clase a otra si el método es más relevante en un contexto diferente.

Mover Campo (Move Field): similar al movimiento de método, pero se refiere al atributo de una clase en lugar del método.

Eliminar Parámetros (Remove Parameters): reducir el número de parámetros de una función, si es posible, para simplificar la invocación de la función.

Simplificación de Expresiones Condicionales (Simplify Conditional Expressions) : simplificar expresiones condicionales complejas, como anidaciones if-else, para hacerlas más legibles.

Combinar Métodos (Combine Methods): combinar dos o más métodos similares en uno solo, reduciendo así la complejidad del código.

Eliminar Código No Utilizado (Remove Dead Code): eliminar el código que nunca se utiliza o se alcanza en ninguna parte del programa.

Agrupar Variables (Group Variables): agrupar variables relacionadas en estructuras de datos, como objetos o arreglos, para mejorar la organización del código.

Extracción de Interfaz (Extract Interface): crear una interfaz para abstraer las funciones de una clase, de modo que puedas usar la interfaz en varios contextos.

Descomposición de Clase (Split Class): dividir una clase en dos o más clases más pequeñas para mejorar la cohesión y la claridad.

Reemplazo de Código Duplicado (Replace Duplicate Code): identificar y eliminar el código duplicado para evitar errores y simplificar el mantenimiento.

Reemplazo de Cadenas Mágicas (Replace Magic Strings/Numbers): sustituir literales constantes (por ejemplo, cadenas o números mágicos) por constantes con nombre para hacerlos más explícitos.

Reemplazo de Jerarquías con Campos (Replace Hierarchy with Field): reemplazar una jerarquía de clases con un campo o atributo para simplificar la estructura.

Extracción de Superclase (Extract Superclass): crear una clase base común para compartir código entre clases relacionadas. Descomposición de Métodos Largos (Long Method Refactoring): dividir métodos largos en métodos más pequeños para mejorar la legibilidad y mantenibilidad del código.

Para realizar correctamente la refactorización existen una serie de guías que sirven para asegurar el correcto funcionamiento de los cambios que se apliquen.

Pruebas automatizadas: antes de comenzar cualquier *refactoring*, asegúrate de tener un conjunto sólido de pruebas automatizadas. Las pruebas te ayudarán a verificar que los cambios no hayan introducido nuevos errores. Ejecuta las pruebas antes y después del *refactoring* para confirmar que el comportamiento del software sigue siendo el mismo.

Planificación: planifica cuidadosamente el *refactoring*. Considera qué partes del código requieren mejoras y cómo puedes dividirlas en pasos manejables. Establece objetivos claros para el *refactoring*. Pequeños pasos: realiza *refactoring* en pequeños pasos incrementales. Modificar una pequeña parte a la vez facilita la gestión de los cambios y el control del proceso.

Comprender el código: antes de comenzar a refactorizar una parte del código, asegúrate de comprender completamente su funcionamiento. Una buena comprensión del código existente es esencial para evitar errores y comportamientos no deseados.

Evitar la proliferación de comentarios: si consideras necesario comentar en exceso el código, puede ser una señal de que el código es poco claro. En lugar de comentar, trata de hacer que el código sea más descriptivo y autoexplicativo a través del *refactoring*.

Mantener el código abierto: durante el *refactoring*, mantén el código abierto y accesible para todo el equipo. La colaboración y la retroalimentación de otros miembros del equipo pueden ser muy útiles.

Supervisar las métricas de calidad del código: utiliza herramientas y métricas de calidad del código, como la complejidad ciclomática, la cobertura del código y el análisis estático, para evaluar la efectividad del *refactoring*. Control de versiones: asegúrate de utilizar un sistema de control de versiones como Git. Esto te permitirá rastrear los cambios, retroceder si es necesario y colaborar con otros durante el *refactoring*.

Mantener la documentación actualizada: actualiza la documentación del código, como los comentarios en el código fuente o la documentación externa, para reflejar los cambios realizados durante el *refactoring*.

Respetar los principios SOLID: los principios SOLID (Responsabilidad Única, Abierto/Cerrado, Sustitución de Liskov, Segregación de Interfaces, Inversión de Dependencias) son pautas importantes para el diseño de código de alta calidad. Durante el *refactoring*, intenta aplicar estos principios cuando sea posible.

Mejoras incrementales: no intentes resolver todos los problemas en una sola sesión de *refactoring*. Realiza mejoras incrementales y regulares con el tiempo.

Retroalimentación y colaboración: comparte tu trabajo de *refactoring* con tu equipo y busca retroalimentación. Otros miembros del equipo pueden tener sugerencias valiosas.

Medir el impacto: después de completar el *refactoring*, evalúa si se lograron las mejoras previstas. Supervisa el rendimiento, la legibilidad y la mantenibilidad del código para ver si han mejorado.

4: Técnicas y herramientas

Durante el desarrollo de la solución propuesta, se siguieron varios pasos para poder identificar y resolver el problema. Partiendo de lo que se sabía originalmente, el TFM de Alda Peñafiel [1], se decide analizar en profundidad los resultados encontrados. Aquí no solo podemos ver la sobrecarga y el aumento exponencial de los tiempos durante el proceso, si no también se pueden ver algunas de las llamadas al *backend* más problemáticas, por lo que se decide lanzar de nuevo las pruebas para comprobar los resultados. A la vez, también se decide revisar directamente el código para ver cómo se relacionan los datos obtenidos. Tras esto se decide analizar posibles soluciones aplicadas por la industria al respecto, con el objetivo de implementarlas y comprobar al final si se aprecia una mejora significativa.

4.1. Análisis de las sobrecarga

Como se ha mencionado antes, se decide usar JMeter para comprobar la cantidad de llamadas que más se están haciendo, y contrastarlo en el código para encontrar algún patrón que nos pueda ayudar a identificar la raíz del problema. Como se puede ver en la figura 4.18, existen 3 variables con tiempos de 1, 2.5 y 5 segundos para controlar la ejecución de las llamadas. Estas partes de código, figura 4.19, serán el objetivo principal de los cambios que vayamos a realizar. Realizando la comprobación a través de JMeter, figura 4.20, podemos ver una serie de peticiones que se lanzan significativamente más que el resto, siendo estas:

/api/room/rounds* Encargada de devolver la ronda en la que se encuentra la partida

/api/chat/get-chat* Encargada de devolver el chat para cada jugador, dependiendo de si está mirando el chat de grupo o el de la sala

/api/form/status Encargada de devolver el estado de la ronda en la que se encuentra la partida

/api/room/get-status* Encargada de devolver el estado de la sala en el que se encuentra la partida

/api/from/choice* Encargada de guardar las decisiones tomadas por los jugadores de cada pregunta

/api/player/by-room* Encargada de devolver los jugadores que están actualmente en la sala

```
timerPeriodShort: 1000, // Number of miliseconds to wait before remaking an API call
timerPeriodMedium: 2500, // Number of miliseconds to wait before remaking an API call
timerPeriodLong: 5000, // Number of miliseconds to wait before remaking an API call
```

Figura 4.18: Variables usadas para el control de lanzamiento de llamadas

```
// Lectura periódica del estado de la ronda actual desde el backend
// Si la ronda deja de estar activa, se muestran los resultados de dicha ronda
this.roundStatus$ = timer(environment.initTimerPeriod, environment.timerPeriodShort).pipe(
  switchMap(() => this.roomService.getRound(this.roomService.getRoomCode(), this.currentRound.round).pipe(
    tap((round: Round) => {
      if(round && !round.active) {
        this.cancelPolling()

        this.messageService.add({
          severity: 'info',
          life: environment.modalTime,
          summary: $localize `Round ${this.currentRound.round} finished`,
          detail: $localize `Showing results`
        })

        this.router.navigate(['/player/resultRound'])
      }
    })
  )),
  retry(),
  takeUntil(this.stopPolling)
)
```

Figura 4.19: Ejemplo uso de variable para relanzar llamada al *backend*

Requests	Executions			
Label	#Samples	FAIL	Error %	Average
Total	**47599**	**26210**	**55.06%**	**582.08**
/api/room/rounds/68046&1	16532	16532	100.00%	702.00
/api/chat/get-chat	14139	180	1.27%	234.00
/api/form/status	7373	7076	95.97%	710.80
/api/room/get-status/68046	2285	7	0.31%	573.25
/api/form/choice	1394	1394	100.00%	635.45
/api/player/by-room/68046	1098	6	0.55%	470.49
/api/form/question	497	9	1.81%	3558.92
/api/graphics/get-results	396	298	75.25%	3618.36
/api/form/feedback	297	200	67.34%	1320.60
/api/room/current-round/68046	199	199	100.00%	841.75

Figura 4.20: Número de peticiones lanzadas para para 100 Hilos en JMeter

4.2. STOMP como implementación de WebSocket

Como se indica en la documentación de Maven [30], STOMP son realmente unas siglas que significan *Simple Text Orientated Messaging Protocol* (Protocolo Simple de Mensajes Orientados a Texto). Como se ha visto en el capítulo anterior, WebSocket es un protocolo a muy bajo nivel, que permite la conexión bidireccional de *frames* de datos, bien texto o binarios, con el problema de no tener información acerca del procesamiento de rutas. Esta implementación nos facilita el acceso a los recursos que pone a disposición el protocolo de WebSocket para el establecimiento de comunicación, manejo de eventos y sus otras funciones.

STOMP en Spring Boot

Spring Boot tiene, de manera nativa [26], soporte para WebSocket mediante el módulo *spring-websocket*. Gracias a este módulo, se pone a disposición de los desarrolladores la implementación **WebSocketMessageBrokerConfigurer**, el cual nos sirve para registrar el *endpoint* y configurar el *broker* que manejará los mensajes. El uso de la anotación *@EnableWebSocketMessageBroker* es lo que nos permite activar WebSocket e integrarlo con la estructura de mensajes de STOMP. También nos ofrece el método *withSockJS()*, que sirve como *fallback* en caso de que el navegador no dé soporte a WebSocket, emulando su funcionamiento a través de llamadas HTTP. También mejora la seguridad, ya que las conexiones de WebSocket están abiertas a todos los clientes por defecto. Si se desea evitar que cualquier cliente pueda conectarse al *endpoint*, mediante la implementación de *HandshakeInterceptor* se pueden usar los *tokens* JWT para validar la autenticación antes de establecer conexiones.

Mientras tanto, STOMP se encarga de la parte del manejo de mensajes y lo relativo a ellos de la siguiente manera:

Organización del enrutamiento de mensajes: Se pone a disposición, de manera análoga a la disponible en la implementación con HTTP, anotaciones para las distintas rutas que se vayan a usar. Mediante la anotación *@MessageMapping* se pueden crear rutas para la recepción de *frames* por parte del servidor que, enlazado junto a *@SendTo*, nos permiten crear rutas para el envío y la suscripción de datos manejados por la aplicación.

Gestión de las suscripciones: Si los clientes quieren suscribirse a un *topic* concreto, STOMP pone a disposición de estos el *frame* de suscripción de la forma que se muestra en la figura 4.21. STOMP se encarga de añadirlo al registro de suscripciones para poder mandarle todos los cambios que se produzcan. De manera análoga, cuando el cliente manda un *frame* de *UNSUBSCRIBE*, STOMP lo saca del registro, evitando que se envíen mensajes a clientes inactivos.

Procesamiento de mensajes entrantes: Parsea el *frame*, ataca al *endpoint* especifico, ejecuta el *@SendTo* para enviar a todos los clientes que estén escuchando la respuesta de la invocación de *@MessageMapping*, y si es necesario se usa un *broker* para otros procesamientos. STOMP maneja de manera distinta los *frames* dependiendo del comando que se pase:

- **CONNECT**: Establecimiento de sesión.
- **SEND**: Envío de mensaje al destino.
- **SUBSCRIBE**: Suscripción a un *i*.
- **UNSUBSCRIBE**: Eliminación de la suscripción del cliente.
- **DISCONNECT**: Cierre de la sesión.

Broadcast de mensajes: Cuando se mandan mensajes a STOMP, este los reenvía a los distintos suscriptores, comprobando si está manejado por un *broker* interno o uno externo.

Manejo de mensajes a un usuario especifico: STOMP también permite el envío de mensajes privados si estos son requeridos por la aplicación.

Manejo de errores: Si un mensaje no está formado correctamente o no puede ser procesado, STOMP manda un *frame* de error al cliente (Figura 4.22). Los desarrolladores también pueden manejar estos errores mediante la anotación *@MessageExceptionHandler* dentro de un controlador.

```
SUBSCRIBE
id:sub-1
destination:/topic/messages
```

Figura 4.21: Ejemplo de un *frame* que manda un cliente para suscribirse a un *topic*, en este caso el de messages

```
ERROR
message: Invalid destination
content-type: text/plain

The destination /invalid was not found.
```

Figura 4.22: Ejemplo de un *frame* de ERROR procesado por STOMP

STOMPJS en Angular

Para Angular, existe la librería STOMPJS [6] perteneciente a **StompJs Family**. Este colectivo ha creado librerías de STOMP para varios lenguajes basados en JavaScript.

Dentro de esta librería no hay un StompJs que se pueda importar, si no que usa clases como *Client*, *Frame* o *Message* para poder usarse, siendo estos copias directas a los de STOMP de Spring Boot para *frontend*. Esto nos permite tener funciones que automáticamente formen los *frames* con los que trabaja STOMP como los de *CONNECT*, *DISCONNECT* o *SEND*, al igual que agilizar la suscripción y de-suscripción a los distintos *topics* que existan.

4.3. Implementación de STOMP

Para la implementación de WebSocket, como se ha mencionado antes, se usa la implementación de STOMP [30], y la librería de STOMPJS [6] para crear el código.

Para poder controlar en el back las llamadas a WebSocket a través de STOMP, se genera una clase nueva llamada ***WebSocketConfiguration***, la cual va a contener el registro del *endpoint* para WebSocket, y los *endpoints* que se van a habilitar para el envío y recepción de mensaje 4.23

En la función ***registerStompEndpoints*** es donde registramos nuestro *endpoint* **/ws** al cual vamos a llamar desde nuestro *frontend* para poder establecer una comunicación

con este, mientras que la función ***configureMessageBroker*** se encarga de establecer el *broker* para recibir los datos (toda llamada dentro del WebSocket que empiece por **/topic** será para recibir datos) , y para poder enviarlos (toda llamada dentro del WebSocket que empiece por **/app** será para enviar datos). Una vez hecho esto solo hay que habilitar la url en la clase ***SecurityConfig*** para que la conexión pueda ser establecida añadiendo *.antMatchers(/ws/**")* en su configuración

```java
package com.crossroads.spring.config;

// import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.simp.config.MessageBrokerRegistry;
import org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker;
import org.springframework.web.socket.config.annotation.StompEndpointRegistry;
import org.springframework.web.socket.config.annotation.WebSocketMessageBrokerConfigurer;

You, 4 months ago | 1 author (You)
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfiguration implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        // Habilitamos un broker simple para enviar mensajes a destinos con el prefijo "/topic"
        registry.enableSimpleBroker(...destinationPrefixes:"/topic");
        // Definimos un prefijo para los mensajes que serán mapeados a métodos de controladores
        registry.setApplicationDestinationPrefixes(...prefixes:"/app");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        // Registramos el endpoint que usará SockJS para la comunicación
        registry.addEndpoint(...paths:"/ws").setAllowedOrigins(...origins:"*").withSockJS();
    }
}
```

Figura 4.23: Clase WebSocketConfiguration

De manera similar, en el *frontend* se crea un servicio que nos va a ayudar a establecer una comunicación con el *endpoint* abierto por el *backend*, siendo aquí más compleja puesto que hay que controlar que no se abra varias veces el *socket*, ya que esto causaría problemas en la comunicación de los datos.

La clase ***StompServcie*** que se crea, puede separarse en varios puntos:

Creación de la clase 4.24: En la creación de la clase cabe destacar que vamos a usar las funciones de la librería **stompjs**, y nos vamos a apoyar de las variables de *connetionPromise*, que resolverá la conexión con WebSocket, y *connected*, que será un *boolean* para comprobar si la conexión ha sido establecida. Junto con estas están *client*, *Client* de stompjs, y *resolveConnect* y *rejectConnect* que nos van a servir para control de conexión, si es valida o si se produce cualquier fallo.

Constructor de la clase 4.25: En el constructor crearemos el *Client* de stompjs, asignandole el socket (apuntando a la url donde esté desplegado el *backend*, por ejemplo

http://localhost:8080/); los atributos *reconnectDelay*, que marca el tiempo en ms que tarda en volver a intentar establecer conexión; y *heartbeat*, que sirve para mantener la conexión abierta, lanzando un PING/PONG; se habilita la depuración, para poder hacer una comprobación de errores; y se le asigna las funciones que tiene que seguir cuando conecta *onConnect* y cuando falla *onStompError*, las cuales están controladas con NgZone, en este caso forzando la sincronicidad mediante su función *run*.

Función *connect* 4.26: La función *connect* se lanza para poder establecer la conexión a **/ws**, también tiene un flujo de condiciones que devolverá; si está establecida la conexión, la resolución de la promesa; si se está estableciendo la conexión, la promesa; y en cualquier otro caso intentará activar el cliente STOMP que hemos creado en el constructor. Esto se hace dado que hay varios servicios diferentes que se pueden lanzar a la vez o después de que se haya conectado al *endpoint*, siendo necesario que estén todos conectados a través de la misma conexión

Función subscribe 4.27: Está función nos permite suscribirnos a los distintos *endpoints* de recepción de datos de los que dispone el *backend*, los distintos **/topic**, y recibir los datos que estos nos proporcionan. Estos datos se parsean ya que los tipos de datos que estamos mandando son de tipo *String*, como indica el estándar de **WebSocket** [33].

Función *send* 4.28: La función *send* nos permite conectarnos al *endpoint* habilitado por el *back*, los diferentes **/app** que existen, para poder enviar cambios que se realicen, como antes, para adaptarnos al *estandar* hay que forzar que los datos se pasen como texto.

```
import { Injectable, NgZone } from "@angular/core";
import { Client, Frame, IMessage, StompSubscription } from '@stomp/stompjs';
import * as SockJS from 'sockjs-client';
import { environment } from 'src/environments/environment';

You, 4 months ago | 1 author (You)
@Injectable({ providedIn: 'root' })
export class StompService {
  private client: Client;
  private connectionPromise?: Promise<void>;
  private resolveConnect!: () => void;
  private rejectConnect!: (err: any) => void;
  private connected = false;
```

Figura 4.24: Clase StompService con las variables usadas

```
constructor(private zone: NgZone) {
  this.client = new Client({
    webSocketFactory: () => new SockJS(`${environment.urlStomp}ws`),
    reconnectDelay: 5000,
    heartbeatIncoming: 0,
    heartbeatOutgoing: 20000,
    debug: (str: any) => console.log('[STOMP]', str),
    onConnect: (frame: Frame) => {
      this.zone.run(() => {
        this.connected = true;
        this.resolveConnect();
      });
    },
    onStompError: (err: any) => {
      this.zone.run(() => {
        console.error('STOMP error:', err);
        this.connected = false;
        this.rejectConnect(err);
      });
    }
  });
}
```

Figura 4.25: Clase StompService, *constructor*

```
connect(): Promise<void> {
  if (this.connected) {
    return Promise.resolve();
  }
  if (this.connectionPromise) {
    return this.connectionPromise;
  }

  this.connectionPromise = new Promise<void>((res, rej) => {
    // guardamos para resolver/rechazar desde los callbacks
    this.resolveConnect = res;
    this.rejectConnect = rej;
    this.client.activate();
  });

  return this.connectionPromise;
}
```

Figura 4.26: Clase StompService, función *connect*

```
/** Se subscribe a cualquier topic dinámicamente */
async subscribe<T = any>(
  topic: string,
  callback: (msg: T) => void
): Promise<StompSubscription> {
  await this.connect();
  return this.client.subscribe(topic, (message: IMessage) => {
    if (message.body) {
      this.zone.run(() => callback(JSON.parse(message.body)));
    }
  });
}
```

Figura 4.27: Clase StompService, función *subscribe*

```
/** Envía un mensaje a un destino /app/** */
send(destination: string, payload: any) {
  if (!this.connected) {
    console.warn('No conectado todavía, mensaje en cola o descartar');
  }
  this.client.publish({
    destination,
    body: JSON.stringify(payload)
  });
}
```

Figura 4.28: Clase StompSercice, función *send*

Cambios producidos de llamadas Http a WebSocket

A la hora de hacer las modificaciones, la complejidad principal reside más en el uso de los *endpoints* por parte de la aplicación móvil en el *backend*, por lo que hay que mantener los *endpoints* de la arquitectura cliente-servidor, pero a su vez tienen que poder comunicar todas las modificaciones correspondientes a los distintos *endpoints* de STOMP. Para el caso del *endpoint* **/api/room/get-status** los cambios producidos serían los siguientes:

Creación del controlador WebSocket 4.29: Se crean los *endpoints* que vamos a manejar, para evitar problemas de datos cruzados, los *endpoints* de WebSocket van a tener siempre un identificador como el código de la sala que los pueda diferenciar entre partidas. En este ejemplo no se está generando un *endpoint* para poder enviar datos, ya que, todos los datos que este *endpoint* mande a sus distintos consumidores,

van a ser producidos por llamadas a otros distintos, que producirán los cambios necesarios en la base de datos. Un ejemplo donde sí se generan ambos *endpoints* es en la clase **ChatWebSocketController**4.30 con sus *endpoints* para controlar las llamadas que controlan el chat de la sala

Creación de función de apoyo 4.31: Se crea la función *SendStatus* para poder llamarla cuando otro *endpoint* genere cambios en el *status*. Obtenemos los datos necesarios mediante las llamadas que sean necesarias y finalmente se ejecuta la función *convertAndSend*, que nos permite generar un cambio desde el servidor para avisar a todos los suscriptores del *endpoint* de WebSocket que le indiquemos, que se han producido cambios, junto con los cambios que se han producido.

Llamada a función de apoyo cuando otro *endpoint* produce cambios 4.32: Desde el *endpoint* que produce los cambios, en este caso es **/api/room/start/id**, se añade la llamada a la función creada, indicando que es necesario que avise a los *endpoints* adecuados.

```java
package com.crossroads.spring.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.messaging.handler.annotation.DestinationVariable;
import org.springframework.messaging.handler.annotation.SendTo;
import org.springframework.stereotype.Controller;

import com.crossroads.spring.dto.RoundDto;
import com.crossroads.spring.dto.StatusDto;
import com.crossroads.spring.service.RoomService;

import lombok.AllArgsConstructor;
import lombok.extern.slf4j.Slf4j;

You, 3 months ago | 1 author (You)
@Controller
@AllArgsConstructor
@Slf4j
public class RoomWebSocketController {
    @Autowired
    private final RoomService roomService;

    @SendTo("/topic/room/get-status/{roomCode}")
    public StatusDto getRoomStatus(@DestinationVariable String roomCode) {
        return roomService.getRoomStatus(roomCode);
    }

    @SendTo("/topic/room/rounds/{code}")
    public RoundDto getRound(@DestinationVariable String code) {
        return roomService.getRound(code);
    }

}
```

Figura 4.29: *RoomWebSocketController*

```java
@MessageMapping("/chat/room/{roomId}")
@SendTo("/topic/chat/room/{roomId}")
public List<MessageDto> enviarMensajeRoom(@DestinationVariable String roomId, @Payload MessageDto messageDto) {
    if(messageDto.getTypeMessage() != TypeMessage.POSITIVE){
        chatService.save(messageDto);
    }
    chatService.sendAllMessage(messageDto);
    return chatService.getMessagesInChat(messageDto.getIdPlayer(), messageDto.getTypeChat());
}
```

Figura 4.30: *ChatWebSocketController*

```
public void sendStatus (Long id){
    Room room = roomRepository.findById(id)
            .orElseThrow(() -> new RoomException(localizedMessagesResolver.
            getLocalizedMessage(
                property:"exception.room.not-found-with-id",
                new String[]{String.valueOf(id)}
            )
        )
    );

    messagingTemplate.convertAndSend(
        "/topic/room/get-status/"+room.getRoomCode(),
        getRoomStatus(room.getRoomCode())
    );
}
```

Figura 4.31: Llamada para poder avisar de los cambios al *endpoint* creado

```
/**
 *
 * Recibe la petición para iniciar una partida.
 *
 * @param id El identificador de la sala que se desea iniciar.
 *
 * @return mensaje de confirmación del inicio de la partida.
 */
@PostMapping("/start/{id}")
public ResponseEntity<String> startGame(@PathVariable Long id){
    roomService.startGame(id);
    roomService.sendStatus(id);
    roomService.sendRound(id);
    return new ResponseEntity<>(localizedMessagesResolver.getLocalizedMessage(property:"room.start-game.success"), HttpStatus.OK);
}
```

Figura 4.32: *Endpoint* de http que tiene que propagar sus cambios a WebSocket

Por su lado, en el *frontend* estos cambios se ven reflejados de la siguiente forma:

Función *Init* 4.33: En los servicios que van a usar STOMP para la comunicación se crea una función *Init* que va a permitir controlar la suscripción a los distintos *topics* que maneje la clase. De esta forma, los *Subject* que se van a mandar a pasar al resto de la aplicación pueden ser *Subjects* normales, que no recuerdan datos, o *BehaviorSubject*, que recuerdan el último dato que se haya mandado, lanzando la función de su *subscribe* como si se acabaran de recibir datos. También se crea la misma cantidad de *StompSubscriptions* que *endpoints* para guardar las suscripciones que nos devuelve la conexiones. Tras esto establecemos la conexión, pasándole el *topic* al que queremos conectarnos y un *callback* donde actualizaremos el valor de nuestros *subject* correspondiente.

Función retorno llamada 4.34: Función para enviar el *subject* como observable, para que las clases que llamen a esta función puedan suscribirse al *topic* a través del *subject* que generamos.

Función envío de datos 4.35: Por su parte, para enviar los datos, solo es necesario indicarle la ruta (*/app/...*) a la que deseamos conectarnos y los datos que queremos pasarle.

```
export class RoomService {
  private static readonly ENDPOINT = 'room'
  private static readonly ROOM_CODE = 'ROOM_CODE'
  private static readonly ROOM_ID = 'ROOM_ID'
  private static readonly ROUND_NUMBER = 'ROUND'

  private statusSubscription: StompSubscription | undefined;
  private roundSubscription: StompSubscription | undefined;
  private statusResponseSubject = new Subject<RoomStatus>();
  private roundResponseSubject = new BehaviorSubject<Round>({} as Round);

  constructor(private httpClient: HttpClient, private playerService: PlayerService, private stomp: StompService) {}

    initRound(roomCode: string): void {
      this.statusSubscription?.unsubscribe();
      this.roundSubscription?.unsubscribe();

      this.getCurrentRound(roomCode).subscribe((round) => {
        this.roundResponseSubject.next(round);
        this.setRoundNumber(round.round);
      });

      this.stomp.connect().then(() => {
        this.stomp.subscribe<RoomStatus>(
          `/topic/room/get-status/${roomCode}`,
          (status:RoomStatus) => {
            this.statusResponseSubject.next(status);
          }
        ).then((subscription:StompSubscription) => {
          this.statusSubscription = subscription;
        });
        this.stomp.subscribe<Round>(
          `/topic/room/rounds/${roomCode}`,
          (round:Round) => {
            this.roundResponseSubject.next(round);
          }
        ).then((subscription:StompSubscription) => {
          this.roundSubscription = subscription;
        });
      })
    }
```

Figura 4.33: Inicialización de los *topics* pertenecientes a room

```
getRoomStatus(): Observable<RoomStatus> {/******* */
  return this.statusResponseSubject.asObservable();
}
```

Figura 4.34: Función para el control de los observables con los datos

```
sendMessage(message: Message, roomId: number, groupId: number): void {
  let url = '';
  switch(message.typeChat){
    case TypeChat.GROUPCHAT:
      url = `/app/chat/group/${groupId}`;
      break;
    case TypeChat.ROOMCHAT:
      url = `/app/chat/room/${roomId}`;
      break;
  }
  this.stomp.send(url, message);
}
```

Figura 4.35: *chatService* función para enviar datos al *endpoint* correspondiente

4.4. Herramientas de medición

Para comprobar si se han cumplido los objetivos de mejora de rendimiento de la aplicación, es necesario revisar las herramientas de medición que se están usando actualmente en el mercado. Para ello se empezará por revisar la tecnología que se ha usado en el TFM de Alda Peñafiel [1], y se explorará alguna otra por si hubiera problemas con esta.

JMeter

Esta es la herramienta que se escogió para el TFM de Alda Peñafiel, en el punto **5.1 Selección de herramienta**, abarcando las páginas 64,65 y 66 de la memoria del proyecto. En este punto empieza explicando los principales criterios de evaluación que se van a usar:

C1. Con este criterio se valora que la aplicación disponga de un plan gratuito con la duración y características suficientes para poder implementar los escenarios de prueba y poder ejecutarlos.

C2. Este requisito valora la posibilidad de realizar una propuesta para extender la funcionalidad del software empleado en la realización de las pruebas de carga.

C3. Se valora que el lenguaje de programación empleado para codificar los escenarios de prueba sea conocido por el desarrollador. También se valora el caso de aquellas herramientas en las que no sea necesario hacer uso de ningún lenguaje de programación.

C4. Se valora la existencia de gran cantidad de tutoriales y ejemplos accesibles a través de Internet. También se valora que la herramienta posea un manual o página web con su documentación.

C5. Se valora la existencia de reseñas en Internet que reflejen que el uso de la herramienta es habitual en el sector.

C6. Se valora que el software sea de tipo SaaS o que la generación de la carga de trabajo (o usuarios virtuales) se lleve a cabo a través de máquinas virtuales desplegadas en un proveedor de *cloud*. Con cualquiera de estas dos características se elimina la limitación de recursos de *hardware* empleados en la ejecución de las pruebas, por lo que se consigue acelerar dicho proceso.

C7. Se valora que se haya encontrado alguna referencia en toda la documentación consultada a alguna característica del software que facilite la realización de las pruebas de carga de las aplicaciones web colaborativas. Como este criterio está relacionado con la característica principal de la aplicación que se quiere probar, se le ha asignado el mayor peso. (p. 64 y 65)

Les asigna un peso, como se muestra en la tabla 4.2 extraída de la Tabla 5.5 del documento de Alda Peñafiel [1] (p. 65).

ID	Criterio/requisito	Peso
C1	Versión gratuita (o plan gratuito)	3
C2	Es open source	4
C3	Lenguaje de programación conocido	3
C4	Existencia de amplia documentación y ejemplos	2
C5	Herramienta de uso extendido en el ámbito de las pruebas de carga	2
C6	Posibilidad de ejecución en el cloud	2
C7	Presenta característica que facilita las pruebas de aplicaciones colaborativas	5

Tabla 4.2: Listado de requisitos y criterios para la selección de la herramienta con la que realizar las pruebas de Crossroads 2.0

Y tras esto evalúa las distintas herramientas de manera similar a como se muestra en la tabla 4.3, siendo en este caso la Tabla 5.6 en el documento de Alda Peñafiel [1] (p. 66)

Nombre herramienta	Puntuación final
Artillery.io	14
Bees with Machine Guns	12
BlazeMeter	12
Boomq.io	5
CloudTest from Akamai	10
Eggplant	2
Flood Element	14
Fortio	10
Gatling	11
HeadSpin	5
JMeter	14
K6	14
LoadFocus	7
LoadNinja	9
LoadRunner	9
Loadster	7
LoadView	12
Locust	14
Neotys Neoload	10
nGrinder	7
Parasoft Load Test	7
puppeteer-webperf	10
Rational Performance Tester	7
Siege	12
Taurus	14
The Grinder	9
Tsung	14
WAPT	8
WebLOAD	7

Tabla 4.3: Puntuaciones obtenidas por cada herramienta tras evaluar los criterios de selección sobre cada una de ellas

Por último explica que lo siguiente teniendo en cuenta que la referencia a la tabla que indica aquí (Tabla 5.6) es la Tabla 4.3 para este documento

> Tras evaluar cada una de las herramientas frente a la lista de criterios anterior, se han obtenido las puntuaciones que aparecen en la Tabla 5.6. Como se puede observar, hay un empate entre las herramientas Artillery.io, Flood Element, JMeter, K6, Locust y Taurus. Como para ninguna de estas herramientas se

> ha certificado el cumplimiento del criterio C7, se ha decidido seleccionar la herramienta JMeter, pues esta herramienta es conocida por el desarrollador y es la herramienta mencionada en la única respuesta a la encuesta a profesionales del sector. Esta herramienta se ha seleccionado a pesar de lo indicado en [40] pues, aunque Locust tiene mejor rendimiento que JMeter, ha pesado más la experiencia previa en el manejo de JMeter por parte del desarrollador. (p.65)

Debido a problemas a la hora de realizar la conexión con WebSocket, que se explicarán en el capítulo 5, se decide descartar JMeter y se decide usar Grafana K6

Grafana

Grafana [11] es una *dashboard* para visualizar, hacer consultas y poner alertas en los datos, sin importar dónde estén guardados. Entre sus características más importantes destacan:

Unificación de los datos: Grafana puede tomar datos de distintas fuentes y mostrarlos en un solo lugar.

Accesible por cualquier: Democratizando los datos, Grafana pretende facilitar una cultura donde estos puedan ser fácilmente accesibles y usados por todos

Un *dashboard* que todos puedan usar: Los *dashboards* de Grafana no solamente intentan dar un significado a los datos colectados de diferentes fuentes, si no que además pueden ser compartidos con otros miembros mejorando la colaboración y la transparencia.

Flexible y versátil: Grafana permite adaptar el panel del *dashboard* para crear una visualización que sea eficaz para los usuarios

Gracias a sus distintos módulos, **Grafana** puede mostrar los datos de distintos servidores de manera comprensible, y generar alertas si se detecta que suceden ciertos eventos importantes, como por ejemplo que el servidor esté caído, o se hayan producido una serie de errores consecutivos o proporcionales en las llamadas de la página.

Prometheus

Como se indica correctamente en la página [14], **Prometheus** es una BBDD Open Source que usa agentes colectores de telemetría para recoger y almacenar métricas usadas para la monitorización y alertas. **Grafana** proporciona **soporte nativo** para este *plugin*, por lo que es fácil integrarlos conjuntamente.

Prometheus es una herramienta fácilmente configurable, permitiendo recoger datos de varias fuentes y mostrarles todos en un único sitio pudiendo ser mostrados con un tipo único de consultas. Los datos que recoge los obtiene con una clave temporal, para facilitar su representación y compresión en distintos escenarios de uso.

Grafana K6

La herramienta pone a disposición varias aplicaciones para facilitar su uso, entre ellas la propia para lanzar los test, y otra que permite una interfaz gráfica para generar los test mediante una grabación, llamada K6 Studio (Figura 4.36). Al realizar una grabación (Figura 4.37), lo primero que pide es una url a la que acceder. Tras esto abre un navegador Chrome (esta herramienta funciona con Chrome o Chromium) en la página elegida, y automáticamente empieza a grabar las llamadas realizadas en la página, tanto las enviadas, como las respuestas esperadas. Al cerrar el navegador que ha abierto, la grabación se detiene y pregunta acerca de las llamadas que se van a usar agrupadas por la *host* origen (Figura 4.38), y genera un *script* de pruebas con todos los casos del recorrido. El *script* generado puede ser usado a posteriori (Figura 4.39) para ejecutar una prueba de estrés de carga, al asignarle un número de **Usuarios Virutales** (*vu* por sus siglas en ingles) o el **target** en el *script*, donde se le puede asignar ademas un *rump up* para los usuarios. Este *script* está escrito en JavaScript, por lo que permite todas sus definiciones a parte de las añadidas por las librerías de K6.

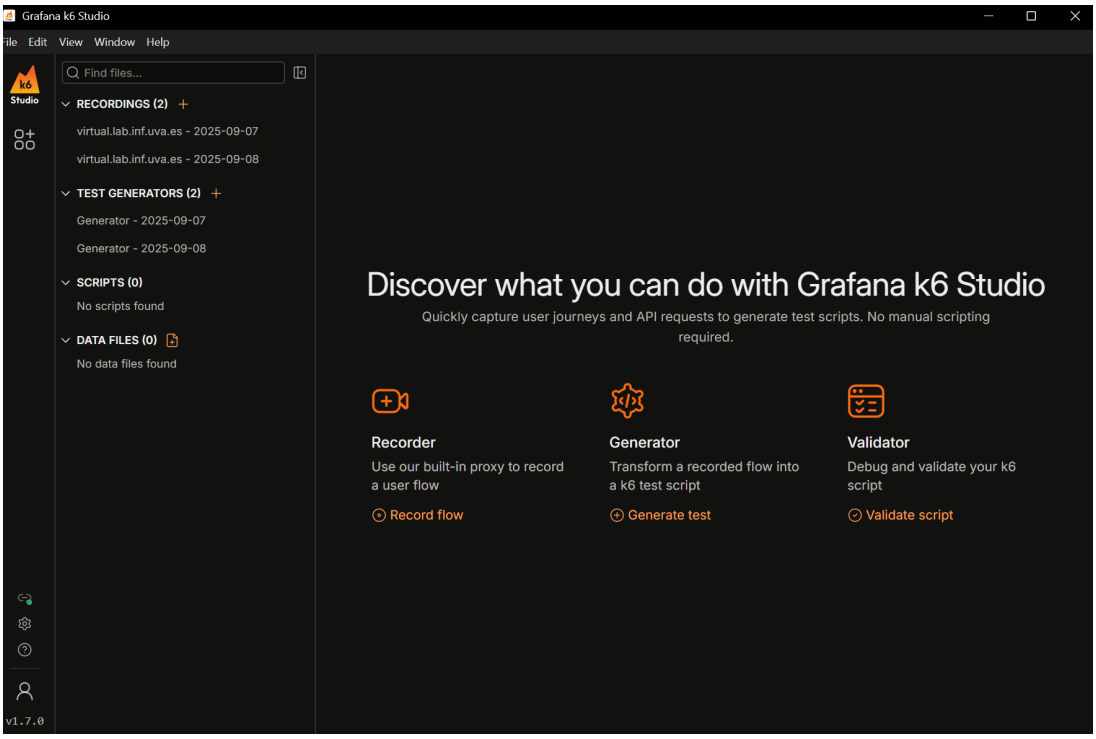

Figura 4.36: Herramienta K6 Studio

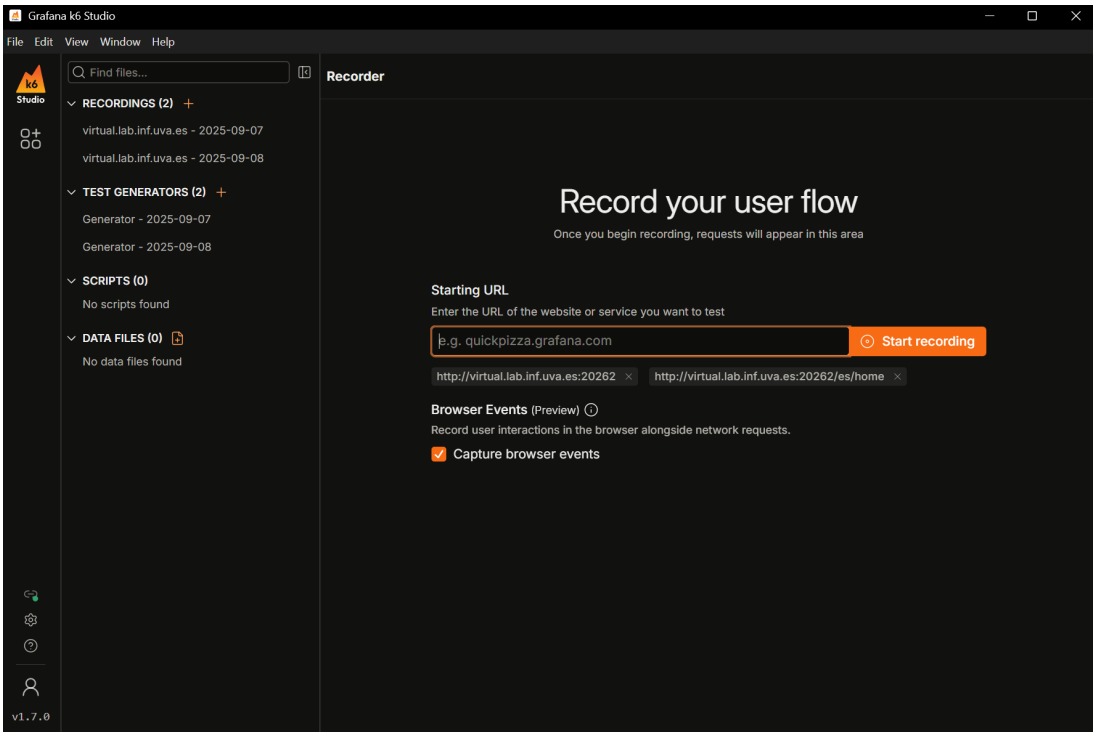

Figura 4.37: Vista de grabación de K6 Studio

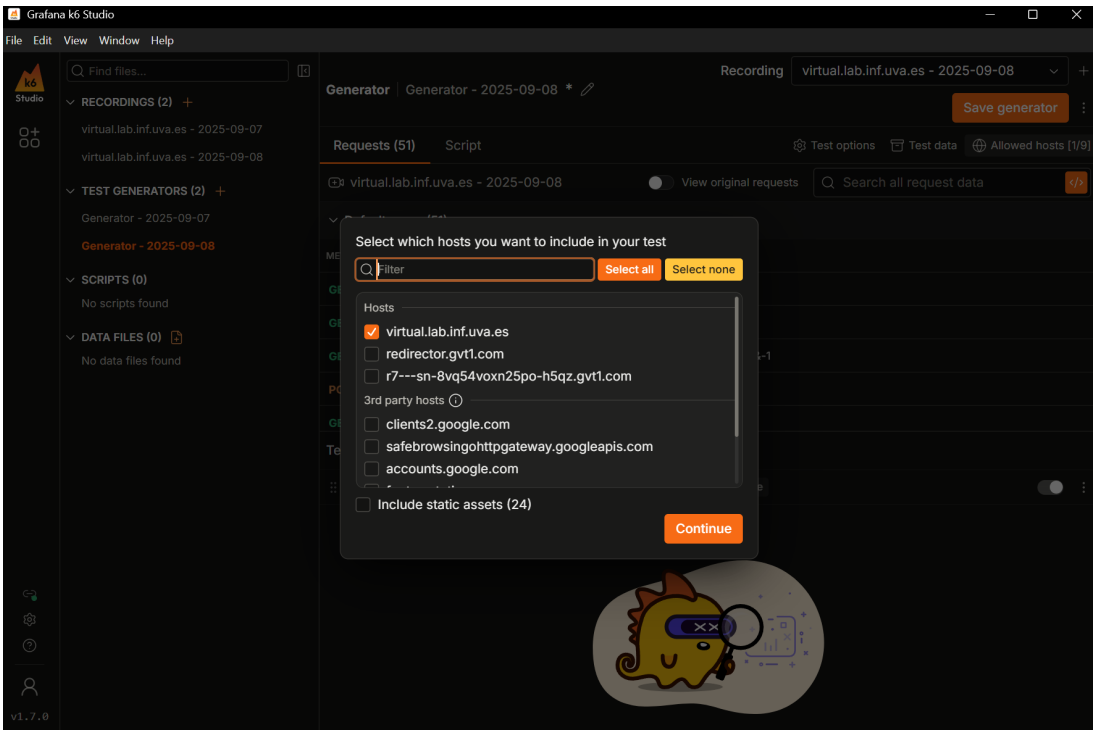

Figura 4.38: Selección de los host de origen

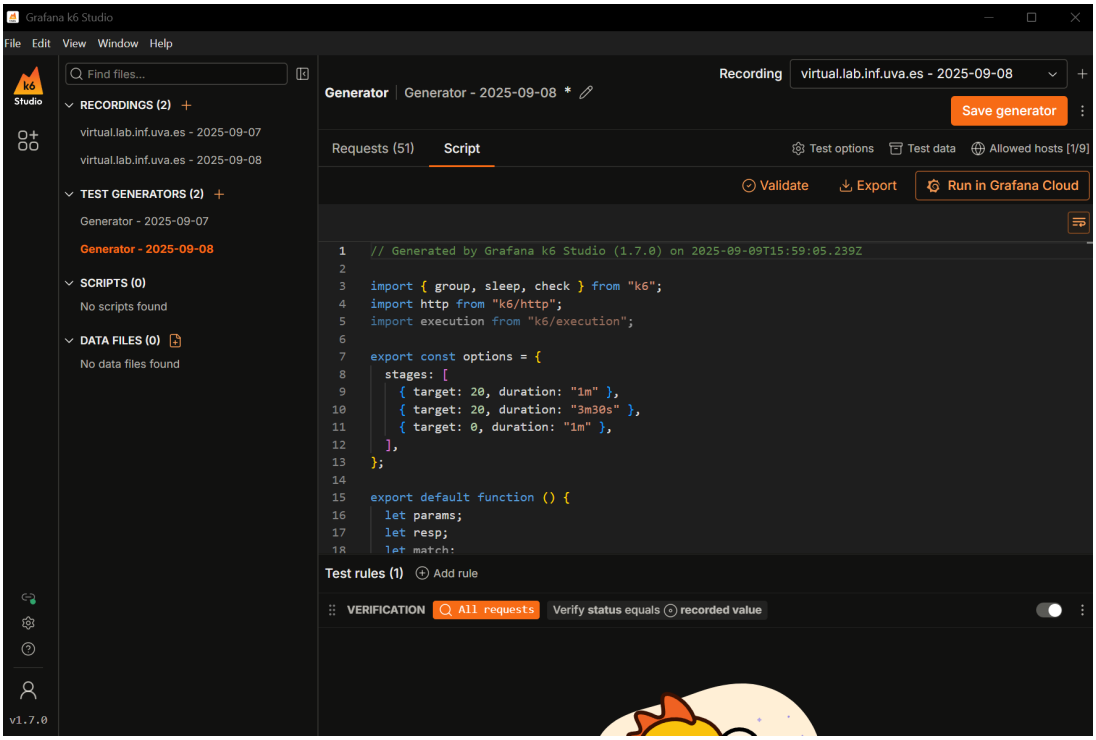

Figura 4.39: Script JS generado por K6 visto desde K6 Studio

5: Aspectos relevantes del desarrollo del proyecto

Durante el desarrollo hubo varios puntos de inflexión que cambiaron provocaron varias veces el cambio de código para poder adaptarse a las necesidades que se iban afrontando. De esta manera se destacan los siguientes:

5.1. Creación y primeros pasos

Para comprobar que era viable realizar las modificaciones se crearon 2 aplicaciones muy básicas para simular un chat, ya que esta era la primera llamada que se iba a modificar, puesto que estas solo se usan en la aplicación web. Esta aplicación sencilla no usaba STOMP, ya que se basó en la implementación que trae por defecto de WebSocket de Maven y Angular.

Al ser una aplicación sencilla permitió demostrar el concepto y la base para posteriores desarrollos, pero no permitió prever los problemas a que habría en una aplicación tan compleja. El principal fue que usaba el sistema de identificación del usuario **Moderador** para identificar en el WebSocket. El *proxy* puede usar un par usuario contraseña para identificarse, pero en esta primera implementación forzaba a que solo el cliente donde el usuarios **Moderador** estuviera identificado se pudiera conectar, mientras que el resto de clientes fallaran. La mayor parte del tiempo de desarrollo de esta primera iteración fue usado en intentar conectar los demás clientes al *socket*, pero sin éxito

5.2. Primera implementación de STOMP

Tras probar varias cosas se encuentra la librería STOMP. Para empezar a usarla solo hubo que crear la clase, **WebSocketConfiguration**, que, como ya se ha explicado antes, permite controlar el *socket* en el *backend*. Por su parte en el *frontend* se modificó el archivo **ChatService** para adaptarse a los nuevos cambios. Como se puede ver en la Figura 5.40, se está usando SockJS en vez de STOMP, también se está generando el *socket* en el

constructo, esto se hizo así porque se pensaba usar un *socket* para cada conexión que fuera necesaria, similar a como funciona en HTTP, en vez de un único *socket* y las suscripciones a los distintos *topics* como se acabó usando. De esta manera, el *endpoint* al que había que conectarse era ws-chat. El cambio de tipo de datos a texto y su parseo se estaba haciendo también dentro de la misma clase, y se estaba devolviendo un *Subject* para que los distintos componentes se suscribieran y recibieran los cambios.

```typescript
import { Injectable } from '@angular/core';
import * as SockJS from 'sockjs-client';
import * as Stomp from 'stompjs';
import { Observable, Subject } from 'rxjs';
import { Chat } from './../models/chat/chat';
import { Message } from './../models/chat/message';
import { environment } from './../../environments/environment';

You, 5 months ago | 3 authors (You and others)
@Injectable({
  providedIn: 'root'
})
export class ChatService {
  private stompClient: Stomp.Client;
  private messageSubject = new Subject<Message>();

  constructor() {
    const socket = new SockJS(`${environment.urlStomp}ws-chat`);
    this.stompClient = Stomp.over(socket);
  }

  connect(): void {

    this.stompClient.connect({}, () => {
      // Nos suscribimos al topic donde el servidor difunde los mensajes
      this.stompClient.subscribe('/topic/public', (message: Stomp.Message) => {
        if (message.body) {
          this.messageSubject.next(JSON.parse(message.body));
        }
      });
    });
  }

  sendMessage(message: Message): void {
    this.stompClient.send('/app/chat.sendMessage', {}, JSON.stringify(message));
  }

  getMessages(): Observable<any> {
    return this.messageSubject.asObservable();
  }
}
```

Figura 5.40: Primera iteración de ChatServices

Esta primera iteración funcionaba perfectamente en la aplicación, pero hubo complicaciones cuando se intentó implementar el segundo *endpoint*, ya que solo se conectaba a uno de los dos *endpoints* disponibles.

5.3. Segunda implementación de STOMP

Debido a los problemas anteriormente mencionados, se optó por un nuevo cambio en el *frontend* para conectarse correctamente al *backend*. En esta iteración ya se empieza a usar la librería **STOMPJS**, y se crea junto con esta el servicio **StompService** (Figura 5.41) para controlar que los clientes solo puedan abrir un *endpoint* cada uno, pero permitiendo que puedan conectarse a este si está abierto y puedan enviar datos, suscribirse y recibir datos de los distintos *topics* que haya habilitados.

```typescript
import { Injectable } from '@angular/core';
import * as Stomp from 'stompjs';
import { Observable, Subject } from 'rxjs';
import { Message } from './../models/chat/message';
import { TypeChat } from '../enums/type-chat';
import { StompService } from './stomp.service';
import { StompSubscription } from '@stomp/stompjs';

You, 4 months ago | 3 authors (You and others)
@Injectable({
  providedIn: 'root'
})
export class ChatService {
  private roomSubscription: Stomp.Subscription | undefined;
  private groupSubscription: Stomp.Subscription | undefined;
  private roomMessageSubject = new Subject<Message>();
  private groupMessageSubject = new Subject<Message>();

  constructor(private stomp: StompService) {}

  initChat(roomId: number, groupId: number): void {
    this.roomSubscription?.unsubscribe();
    this.groupSubscription?.unsubscribe();

    this.stomp.connect().then(() => {
      this.stomp.subscribe<Message>(`/topic/chat/room/${roomId}`, (message: Message) => {
        if (message) {
          this.roomMessageSubject.next(message);
        }
      }).then((subscription: StompSubscription) => {
        this.roomSubscription = subscription;
      });
      this.stomp.subscribe<Message>(`/topic/chat/group/${groupId}`, (message: Message) => {
        if (message) {
          this.groupMessageSubject.next(message);
        }
      }).then((subscription: StompSubscription) => {
        this.groupSubscription = subscription;
      });
    })

  }
```

Figura 5.41: Segunda iteración *ChatService*, donde se empieza a usar ya STOMPJS y StompService para controlar la conexión

Estos cambios permitieron conectar varios servicios que se ejecutaran en distintos componentes y en distintos momentos pudieran usar el mismo *socket*. También empezó a haber complicaciones con las distintas funcionalidades, teniendo que revisar varios componentes y seguir adaptándolos al nuevo uso de *sockets*, donde estos no iban a ser llamados constantemente cada poco tiempo, si no cada vez que hubiera cambios, lo cual produjo algunas inconsistencias durante el desarrollo y las pruebas que se hicieron. Esto junto con llamadas anidadas que había, complicó bastante el cambio de las siguientes llamadas en el *frontend*, ya que hasta ahora en el *backend* tan solo era necesario crear una

nueva clase (Figura 5.42), donde se le indicara los *maps* de *topic* y *send* que iba a tener para los datos, y que usara la misma función que estaba usando hasta ahora, tan solo añadiendo diferenciadores como el id de la sala o grupo, identificadores únicos, con la idea de evitar que distintas salas de juego pudieran interferir entre sí.

```
@MessageMapping("/chat/room/{roomId}")
@SendTo("/topic/chat/room/{roomId}")
public List<MessageDto> enviarMensajeRoom(@DestinationVariable String roomId, @Payload MessageDto messageDto) {
    if(messageDto.getTypeMessage() != TypeMessage.POSITIVE){
        chatService.save(messageDto);
    }
    chatService.sendAllMessage(messageDto);
    return chatService.getMessagesInChat(messageDto.getIdPlayer(), messageDto.getTypeChat());
}
```

Figura 5.42: Muestra de mapeado de una función de WebSocket en la clase ChatWebSocketController para el control de los mensajes de la sala

5.4. HTTP propaga cambios a WebSocket

El último punto que quedó de mejora para los cambios venía de llamadas que producían cambios en los datos manejados por alguno de los *topics*. Como se puede ver en la Figura 5.43, la clase **RoomWebSocketController** no tiene ninguna manera de recibir datos, pero sí tiene un *topic* para transmitir sus cambios a todos los clientes conectados. Esto se debe a que los cambios que se producen en estas funciones vienen dados por otras llamadas que pueden transmitir los cambios a lo largo de la ejecución del código.

```java
package com.crossroads.spring.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.messaging.handler.annotation.DestinationVariable;
import org.springframework.messaging.handler.annotation.SendTo;
import org.springframework.stereotype.Controller;

import com.crossroads.spring.dto.RoundDto;
import com.crossroads.spring.dto.StatusDto;
import com.crossroads.spring.service.RoomService;

import lombok.AllArgsConstructor;
import lombok.extern.slf4j.Slf4j;

You, 3 months ago | 1 author (You)
@Controller
@AllArgsConstructor
@Slf4j
public class RoomWebSocketController {
    @Autowired
    private final RoomService roomService;

    @SendTo("/topic/room/get-status/{roomCode}")
    public StatusDto getRoomStatus(@DestinationVariable String roomCode) {
        return roomService.getRoomStatus(roomCode);
    }

    @SendTo("/topic/room/rounds/{code}")
    public RoundDto getRound(@DestinationVariable String code) {
        return roomService.getRound(code);
    }

}
```

Figura 5.43: La clase *RoomWebSocketController* no tiene forma de recibir mensajes directamente, puesto que todos los cambios son producidos por otras llamadas

Estos cambios habitualmente se producen por llamadas hechas a través de HTTP, por ejemplo, para el caso de la función *getRoomStatus*, los cambios que se producen en los clientes venir originados de de la función *startGame* de la clase **RoomController-2** (Figura 4.32), donde se llama a la función *sendStatus* de **RoomService** para obtener los cambios y enviarlos por el *topic* correspondiente (Figura 5.45), usando la función *converAndSend* que pone a disposición **SimpMessagingTemplate**, que serializa el objeto dado, lo envuelve como un *menssaje*, que **STOMP** puede entender, y lo envía al destino dado, en este caso /topic/room/get-status/roomCode. De esta manera, se consigue que los cambios se puedan transmitir correctamente a los *topics* en cuestión, pero esto aumentó la

complejidad de muchas llamadas HTTP, ya que ahora tienen que ser estas las encargadas de avisar que se han hecho cambios mediante la llamada *send* correspondiente.

```java
/**
 *
 * Recibe la petición para iniciar una partida.
 *
 * @param id El identificador de la sala que se desea iniciar.
 *
 * @return mensaje de confirmación del inicio de la partida.
 */
@PostMapping("/start/{id}")
public ResponseEntity<String> startGame(@PathVariable Long id){
    roomService.startGame(id);
    roomService.sendStatus(id);
    roomService.sendRound(id);
    return new ResponseEntity<>(localizedMessagesResolver.getLocalizedMessage(property:"room.start-game.success"), HttpStatus.OK);
}
```

Figura 5.44: Endpoint /room/start/id que controla el envío de datos a los *topics* de status y round

```java
public void sendStatus (Long id){      You, 4 months ago • creacion de llamadas room/get-status y room/rou…
    Room room = roomRepository.findById(id)
            .orElseThrow(() -> new RoomException(localizedMessagesResolver.getLocalizedMessage(property:"exception.room.not-found-with-id", new String[]{String.valueOf(id)})));

    messagingTemplate.convertAndSend("/topic/room/get-status/"+room.getRoomCode(), getRoomStatus(room.getRoomCode()));
}
```

Figura 5.45: Función SendStatus

5.5. Revisión Comparativa del Proyecto

En esta sección se planea analizar, mediante el uso de diagramas, las principales diferencias entre la versión original y la modificada que se ha generado con este trabajo. Para esto se usarán los diagramas de despliegue y los esquemas de componentes del proyecto *backend*.

Lo primero que destaca, como se puede ver en las Figuras 5.46 y 5.47, es el control que tiene el entorno de Apache Server, que mediante un proxy inverso, controla a quien dirigir las llamadas, de tal manera que el cliente solo puede realmente atacar a dicho proxy. Con los cambios realizados, el cliente puede atacarlo usando el protocolo HTTP o WS siendo estas igualmente capturadas por el proxy.

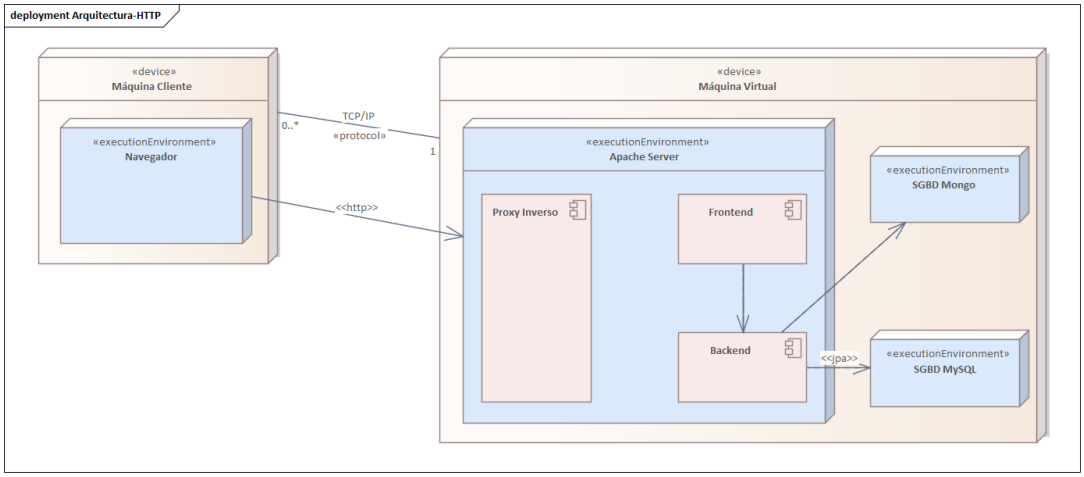

Figura 5.46: Diagrama de despliegue de la versión original

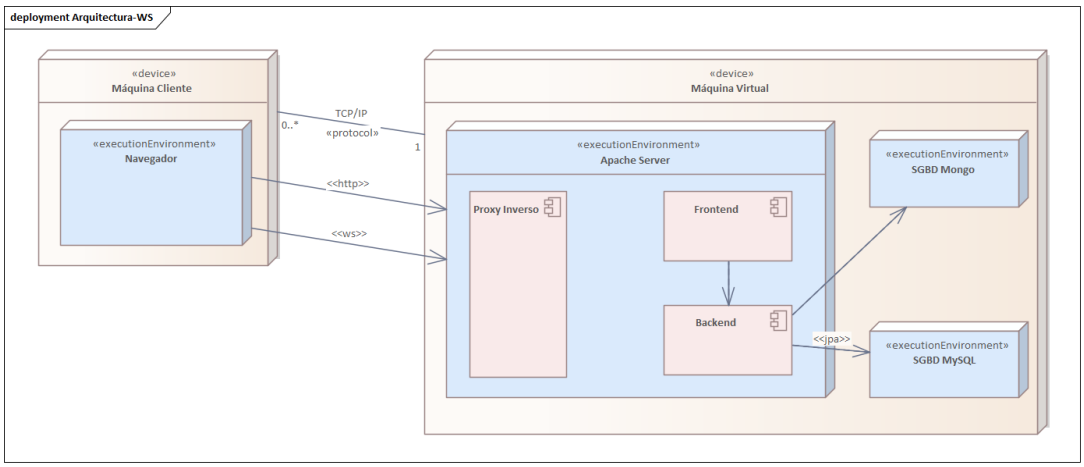

Figura 5.47: Diagrama de despliegue de la versión modificada

La extensión de los cambios realizadas por la rearquitectura puede verse sobre todo en las clases *Server* y *Controller*, ya que estas manejan finalmente la interacción entre el *frontend* y el *backend*, y la interacción de este último con las BBDD. De esta manera, se puede observar los cambios entre las Figuras 5.48 y 5.49 la aparición de las clases **ChatWebSocketController**, **FormWebSocketController**, **RoomWebSocketController** y **PlayerWebSocketController** que poseen los *endpoint* para poder acceder mediante WebSocket. Estas reciben como parámetro simplemente un *String*, ya que, como se ha visto anteriormente, WebSocket solo admite ***Texto***, ***Blobs*** y ***ArrayBuffer***. Siendo estos datos generados como JSON, son fácilmente transformados a texto plano. Algunas funciones pueden recibir parámetros extra que son obtenidos del propio *endpoint*, como puede ser el código de la sala o del grupo al que pertenece quien envía esta información.

Figura 5.48: Esquema de clases de control de la versión original

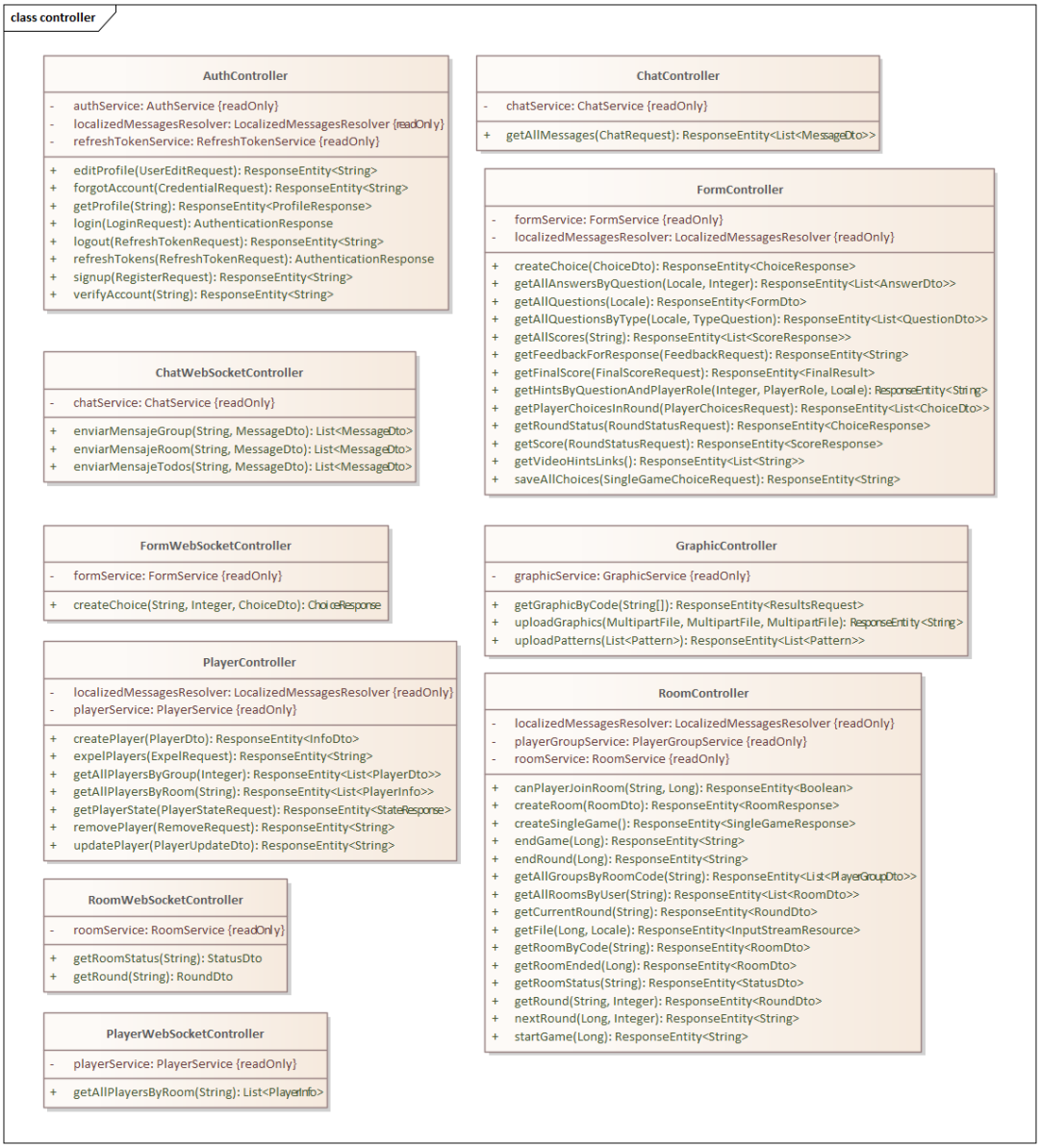

Figura 5.49: Esquema de clases de control de la versión modificada

Por su parte, como puede verse en las Figuras 5.50 y 5.51, se han generado nuevas funciones *send* en las clases correspondiente (**ChatService**, **FormService**, **RoomService** y **PlayerService**) para poder mantener las suscripciones de WebSocket correctamente actualizadas. Esto es debido a la actualización de los datos en la base de datos, ya que llamadas HTTP pueden modificar el estado de datos que necesitan ser enviados a los suscriptores. En la versión original esto no era necesario puesto que las llamadas correspondientes se actualizaban a intervalos fijos, por lo que sus datos estarían actualizados en el siguiente, pero con WebSocket es necesario avisar cuando se produzcan estos cambios. De esta manera, las funciones *SendStatus* o *SendRound* de la clase **RoomService** serán ejecutadas cuando el estado de la ronda cambie, por ejemplo, al iniciar una partida, o al acabar la ronda.

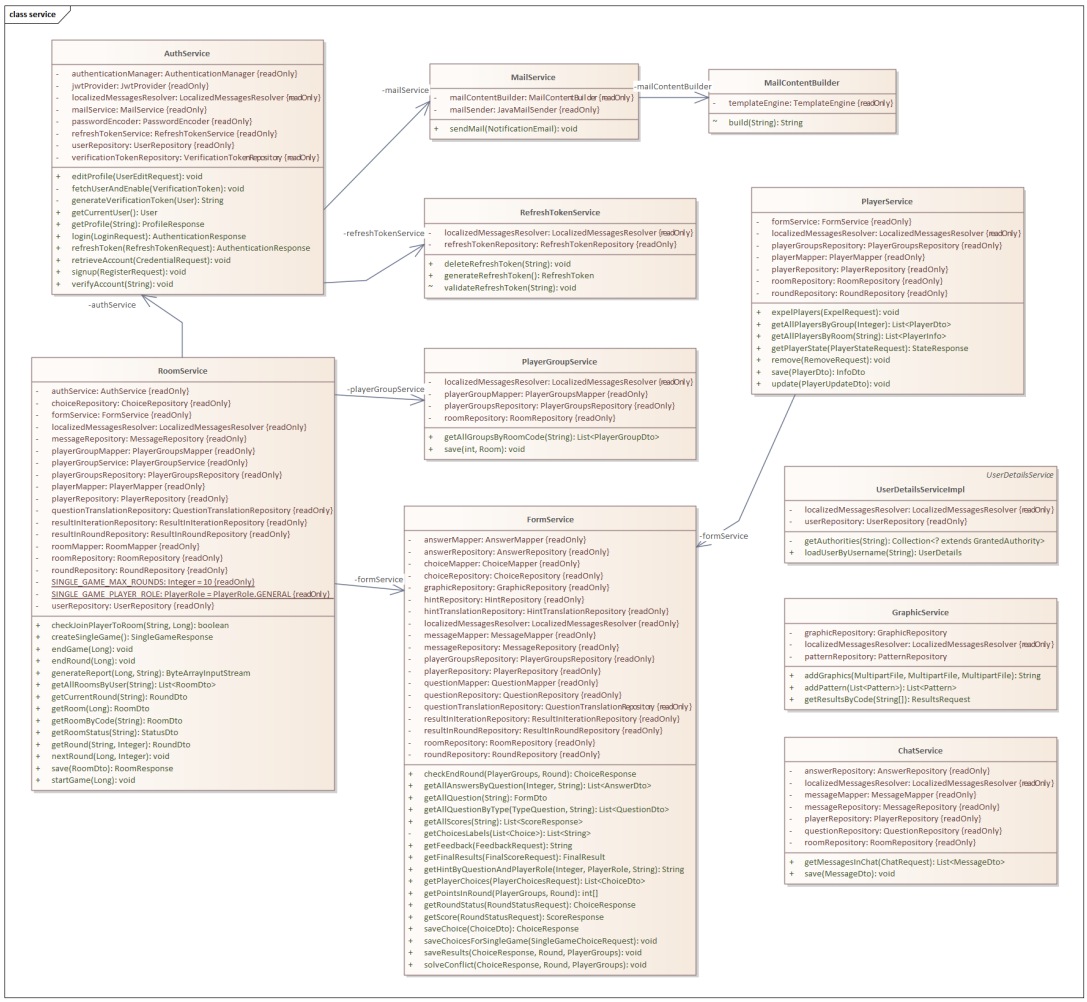

Figura 5.50: Esquema de clases de servicio de la versión original

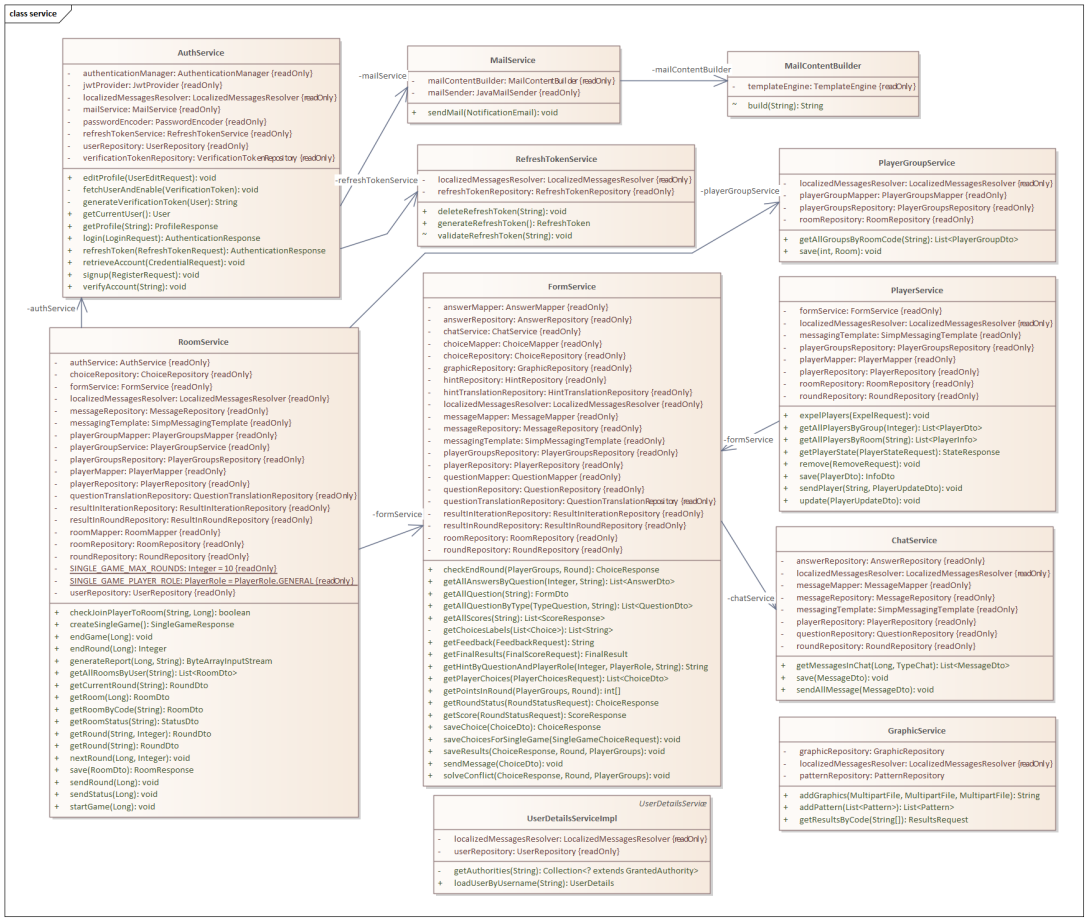

Figura 5.51: Esquema de clases de servicio de la versión modificada

5.6. Graphana y JMeter

Como se explicó con anterioridad, para las pruebas de carga se pensó en usar JMeter, ya que es la herramienta que se había usado para comprobar la carga del servidor en el TFM de Alda Peñafiel. De esta forma, una vez se desplegó en la máquina virtual la aplicación y se hizo funcionar, se modificaron los test de carga para que apuntaran a la nueva máquina. Tras esto se pudieron lanzar los test de carga y se obtuvieron unos datos similares a los obtenidos por Alda Peñafiel.

Una vez alcanzado este punto, se actualizó el código en la máquina virtual para poder habilitar los cambios con WebSocket y se confirmó su funcionamiento mediante unas pruebas manuales de uso de la propia aplicación, y se dispuso a modificar los test para eliminar las llamadas http modificadas por este trabajo y pasarlas a WebSocket (Figura 5.52, y aunque sí se logró que se conectara al *socket*, solo se consiguió que se suscribiera al primer *topic*, (SUBSCRIBE /topic/room/get-status/68046), y que leyera la suscripción confirmando que efectivamente estaba conectado, pero el resto de conexiones fallaba.

Debido a esto se decidió buscar alternativas, siendo Grafana la herramienta que más utilidad se le veía para este caso.

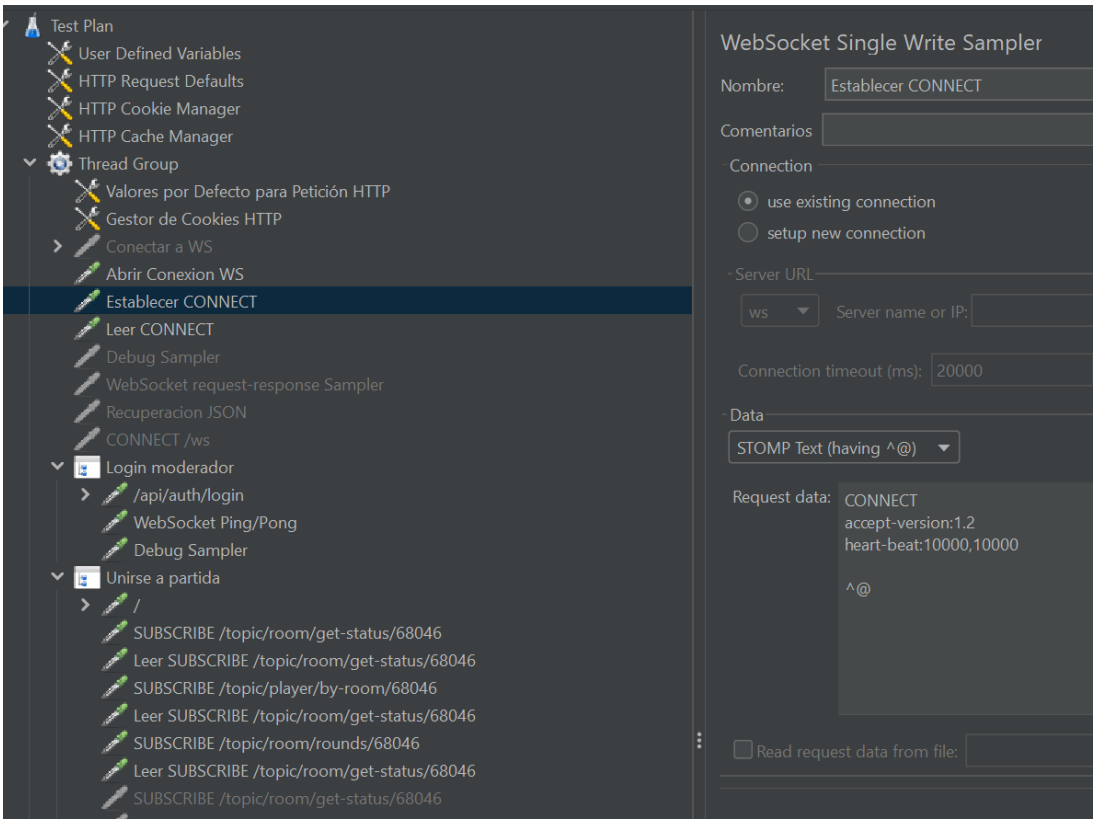

Figura 5.52: Llamadas WebSocket creadas con los *frames* de STOMP

Grafana

Para la instalación de Grafana se decide usar el OSS ya que permite la instalación directa sobre la máquina virtual. Para esto se instala Grafana como indica en la guía [12], y se modifica en su archivo de configuración la ruta que va a ser usada, puesto que se va a acceder a través de Apache. En el archivo de configuración se modifica la siguiente variable $root\_url = \%(protocol)s : // \%(domain)s : \%(http\_port)s/grafana/$

Mientras tanto en la configuración de Apache se añaden las siguientes líneas:

ProxyPreserveHost on
ProxyPass /grafana http://27.0.0.1:3000
ProxyPassReverse /grafana http://127.0.0.01:3000

Con estos cambios podemos entrar en Grafana con la siguiente url:

http://virtual.lab.inf.uva.es:20262/grafana

Con esto se tiene desplegado Grafana, pero aún se necesitan los datos. De esta forma se decide trabajar con **Proemetheus** [14] para recopilar todos los datos que se generen (como agregador de datos), y las fuentes de **Apache_exporter** [19] y **Telegraf** [16], para obtener los datos que queremos mostrar.

Para los *dashboards* se deciden usar 2 distintos con distintas fuentes de datos, ambos usando el agregador de **Prometheus**:

Apache [28]: Un *dashboard* que muestra los datos de **Apache_exporter**, pudiendo ver diversos datos, como el *uptime* o la cantidad de KB enviados por el servidor de Apache (Figura 5.53).

Apache2 [17]: Un *dashboard* que muestra los datos de **Telegraf**, en este caso pudiendo comprobar principalmente los datos de las llamadas entrantes con los códigos de respuesta de esta, pudiendo analizar los problemas al momento de contestar (Figura 5.54), o los tiempos medios de respuesta, carga de CPU. Este *dashboard* fue modificado del original para adaptarse a las necesidades del proyecto.

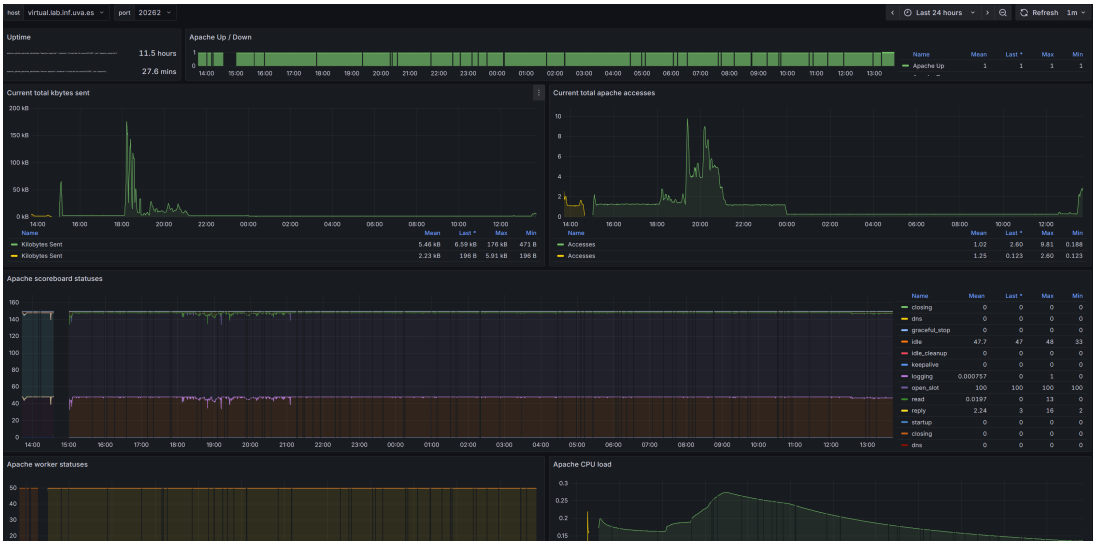

Figura 5.53: *Dashboard* con muestra del *uptime* y gráficas de datos obtenidos de Apache_exporter

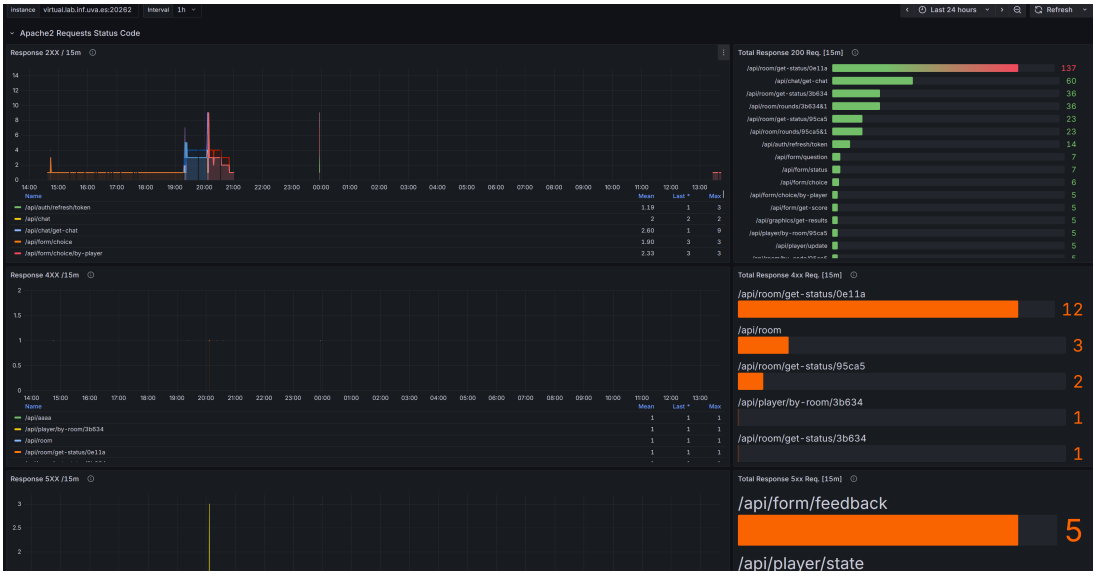

Figura 5.54: *Dashboard* que muestra los datos de las llamadas hechas a /api o /ws disgregadas en los códigos 2xx, 4xx y 5xx, junto con la cantidad de llamadas realizadas ordenadas de mayor a menor

Para poder realizar correctamente la instalación de **Apache_exporter** fue necesario crear otro Host Virtual de Apache para que recoja los datos, haciendo que escuche en el puerto 8081, creando un servicio que se pueda arrancar desde *systemctl*, y siendo expuesto a través del puerto 80, ya que es el único puerto que tenemos disponible de acceso desde el exterior. De manera similar se realiza la instalación con **Telegraf** [16]. Se decide exponer tanto **Apache_exporter** como **Telegraf** hacia fuera, no porque sea imprescindible, si no para que puedan ser revisados y vistos desde el navegador (Figuras 5.55, 5.56), comprobando que se tienen los datos que se necesitan y que se está haciendo la recogida correcta de datos por parte de **Prometheus**.

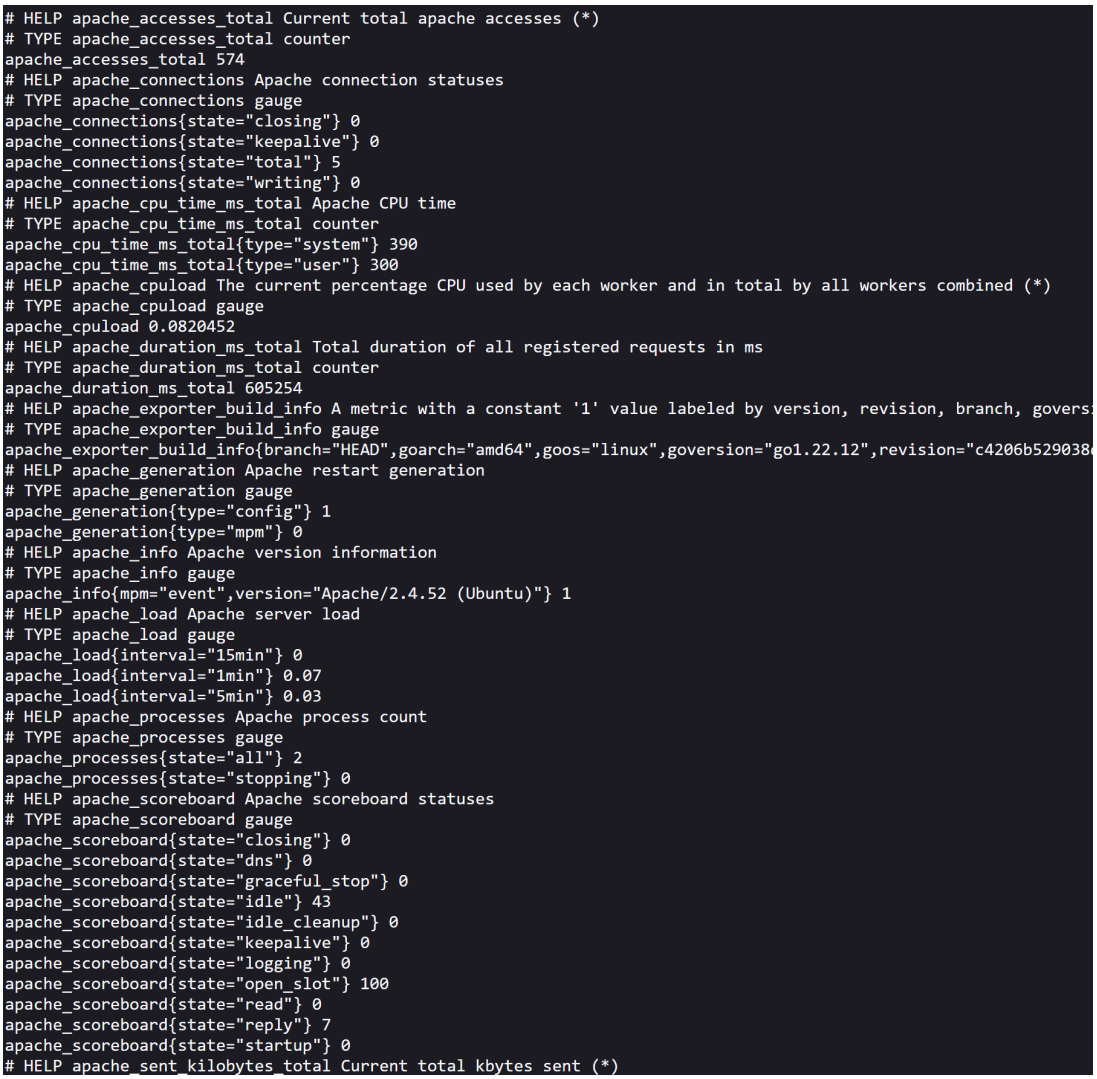

Figura 5.55: Página de visualización de datos de apache_exporter (http://virtual.lab.inf.uva.es:20262/apache-exporter/metrics)

```
# HELP apache2log_http_version Telegraf collected metric
# TYPE apache2log_http_version untyped
apache2log_http_version{agent="Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:142.0
api/live/ws",resp_code="400",verb="GET"} 1.1
apache2log_http_version{agent="Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:142.0
virtual.lab.inf.uva.es:20262/es/chair/dashboard",request="/api/room/get-status/0e
apache2log_http_version{agent="Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:142.0
virtual.lab.inf.uva.es:20262/prometheus/query?
g0.expr=%28apache2log_resp_bytes%7Binstance%3D%7E%22virtual.lab.inf.uva.es%3A2026
prometheus/api/v1/notifications/live",resp_code="503",verb="GET"} 1.1
apache2log_http_version{agent="Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:142.0
virtual.lab.inf.uva.es:20262/prometheus/targets",request="/prometheus/api/v1/noti
apache2log_http_version{agent="Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:142.0
virtual.lab.inf.uva.es:20262/prometheus/targets",request="/prometheus/api/v1/scra
# HELP apache2log_resp_bytes Telegraf collected metric
# TYPE apache2log_resp_bytes untyped
apache2log_resp_bytes{agent="Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:142.0)
api/live/ws",resp_code="400",verb="GET"} 330
apache2log_resp_bytes{agent="Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:142.0)
virtual.lab.inf.uva.es:20262/es/chair/dashboard",request="/api/room/get-status/0e
apache2log_resp_bytes{agent="Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:142.0)
virtual.lab.inf.uva.es:20262/prometheus/query?
g0.expr=%28apache2log_resp_bytes%7Binstance%3D%7E%22virtual.lab.inf.uva.es%3A2026
prometheus/api/v1/notifications/live",resp_code="503",verb="GET"} 580
apache2log_resp_bytes{agent="Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:142.0)
virtual.lab.inf.uva.es:20262/prometheus/targets",request="/prometheus/api/v1/noti
apache2log_resp_bytes{agent="Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:142.0)
virtual.lab.inf.uva.es:20262/prometheus/targets",request="/prometheus/api/v1/scra
# HELP apache_BusyWorkers Telegraf collected metric
# TYPE apache_BusyWorkers untyped
apache_BusyWorkers{host="virtual",port="8081",server="localhost"} 1
# HELP apache_BytesPerReq Telegraf collected metric
# TYPE apache_BytesPerReq untyped
apache_BytesPerReq{host="virtual",port="8081",server="localhost"} 1592.65
# HELP apache_BytesPerSec Telegraf collected metric
# TYPE apache_BytesPerSec untyped
apache_BytesPerSec{host="virtual",port="8081",server="localhost"} 2073.04
# HELP apache_CPUChildrenSystem Telegraf collected metric
# TYPE apache_CPUChildrenSystem untyped
apache_CPUChildrenSystem{host="virtual",port="8081",server="localhost"} 0
# HELP apache_CPUChildrenUser Telegraf collected metric
# TYPE apache_CPUChildrenUser untyped
apache_CPUChildrenUser{host="virtual",port="8081",server="localhost"} 0
# HELP apache_CPULoad Telegraf collected metric
# TYPE apache_CPULoad untyped
apache_CPULoad{host="virtual",port="8081",server="localhost"} 0.157609
# HELP apache_CPUSystem Telegraf collected metric
# TYPE apache_CPUSystem untyped
apache_CPUSystem{host="virtual",port="8081",server="localhost"} 0.28
# HELP apache_CPUUser Telegraf collected metric
# TYPE apache_CPUUser untyped
apache_CPUUser{host="virtual",port="8081",server="localhost"} 0.3
# HELP apache_ConnsAsyncClosing Telegraf collected metric
# TYPE apache_ConnsAsyncClosing untyped
```

Figura 5.56: Página de visualización de datos de Telegraf (http://virtual.lab.inf.uva.es:20262/telegraf/metrics)

Una vez hecho esto, se monta **Prometheus**, siguiendo la guía que se encontró [24], y la guía para añadir **Apache_exporter** [25], añadiendo **Telegraf** de la misma manera, modificando el archivo prometheus.yml y añadiendo los trabajos para ambas fuentes y habilitando el servicio. Después, y de manera similar a como se hizo con las fuentes, se expone **Prometheus** al puerto 80 de Apache para que pueda ser revisado y se puedan realizar pruebas de peticiones de datos con las *queries* que ofrece (Figura 5.57).

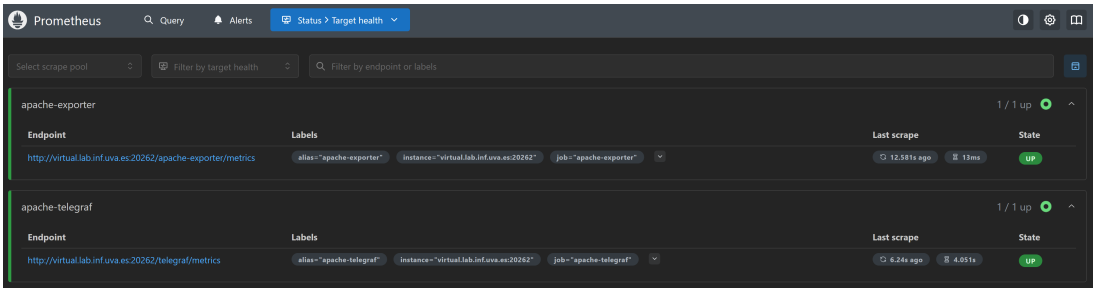

Figura 5.57: Lista de fuentes disponibles en Prometheus, junto con su estado (http://virtual.lab.inf.uva.es:20262/prometheus)

5.7. Pruebas de Carga

Como se ha mencionado antes, las pruebas de carga no se pueden realizar con JMeter por problemas con WebSocket, por lo que se recurrirá a una aplicación que también pone a disposición Grafana, **K6** [13], una herramienta para realizar pruebas a aplicaciones web. Esta herramienta permite realizar varios tipos de prueba, aunque en el ámbito de este trabajo, solo se explicará las pruebas de carga.

Ejecución de las pruebas

Grabada la prueba y generado el *script*, se modifica para poder añadir el código de la sala a la que poder acceder mediante de una variable de entorno a la que se le da el nombre de **ROOM** y se puede pasar a través del comando de la consola. Por otra parte se decide ir ajustando el número de vu a mano, modificando el *script*, de manera paralela a como hizo Alda Peñafiel en su TFM, se pretende realizar las pruebas con 30, 50 y 100 vu. Con el ejemplo de la siguiente configuración se pude hacer un *rump up* a 30 vu en 1 minuto, la ejecución con 30 vu durante 25 minutos y una desescalada a 0 vu durante 1 minuto:

```
export const options = {
  stages: [
    { target: 30, duration: "1m" },
    { target: 30, duration: "25m" },
    { target: 0, duration: "1m" },
```

```
  ],
};
```

También se planea usar la opción **full** del modo de resultado con –summary-mode full, la cual generará un perfil detallado de los resultados de la prueba. El comando también acepta una opción para extraer los datos en formato csv con –out csv=[nombre-archivo], pero genera archivos demasiado grandes y con demasiada información para ser utilizados eficientemente en este trabajo.

De esta manera, para lanzar las pruebas se usa la aplicación de K6, en este caso, y al ser usado desde Windows, lanzado desde Powershell sería el siguiente comando:

```
k6 run -e ROOM=[CODIGO_SALA] --summary-mode full k6-master-test.js
```

Intentando imitar lo más cercano posible a las pruebas realizadas por Alda Peñafiel, se virtualiza una máquina con las mismas prestaciones que poseía la máquina que usó para las pruebas, estas serían:

Máquina virtual con Ubuntu como sistema operativo, CPU con 4 cores, una memoria RAM de 16 GB y 50GB de espacio total de disco.

Como es una máquina virtualizada y tampoco se dispone de mucho tiempo, se decide reducir el perfil de pruebas a 5 pruebas con cada cantidad de Usuarios Virtualizados para ambos casos, tanto la aplicación original, donde se usa HTTP *Polling*; como con las modificaciones realizadas, donde se usa WebSocket. El objetivo principal de estas pruebas no es saber cómo de bien rinde la aplicación, si no comparar si los cambios han sido realmente efectivos, por lo que el reducido número de pruebas no se considerará como un problema a la fiabilidad de los resultados.

5.8. Resultados Obtenidos

Tras realizar las grabaciones del proyecto Crossroads 2.0 en la versión modificada, se descubre un problema con la herramienta, esta no es capaz de recoger correctamente los eventos de WebSocket, por lo que sólo guardan las interacciones realizadas por HTTP. Para representar WebSocket, se modifica el *script* de tal manera que lance 20 veces la cantidad de usuarios virtualizados para el flujo de HTTP, en una petición ficticia a uno de los *sockets* disponibles para generar trabajo. Por otra parte, el flujo de las pruebas del *script*, no pudo realizar correctamente todas las comprobaciones que fueron grabadas debido a la imposibilidad de inicializar correctamente la sala, por lo que se disparó el número de llamadas HTTP fallidas. Aunque esto afecta a la exactitud de las pruebas, se estima que no influye en la comparativa que se pretende realizar entre ambas versiones, ya que ambas van a ser equivalentes.

Los datos que se van a tener en cuenta en la comparativa son los siguiente:

- La cantidad de datos enviados, recibidos y sus ratios.
- La cantidad de iteraciones realizadas y su ratio.
- El uso de CPU en las pruebas.

Explicado esto, se obtienen los siguientes resultados para cada versión de la aplicación.

Resultados Aplicación Original

Las pruebas con la aplicación en su estado original se caracterizaron por un uso excesivo de CPU, como se muestra en las Figuras 5.58, 5.59, 5.60 y 5.61 y en la Tabla 5.4. El uso tan elevado de este recurso, teniendo una máxima para todos los casos por encima del 95 % de uso y una media de entre el 80-90 %, son un signo de un sistema que ha llegado al máximo de sus capacidades, explicando de esta forma la cantidad tan baja de iteraciones por segundo que podía resolver, entre 3 y 4 como se muestra en la Tabla 5.5

Como se comprobó en el trabajo de Alda Peñafiel, estos resultados eran esperables. La aplicación sufre una rápida degradación del servicio, apreciable en las Figuras 5.64 y 5.65 y la Tabla 5.5, donde las pruebas realizadas con 100 VUs fueron las que menor cantidad de iteraciones realizaron. Contrasta por otro lado la cantidad de datos que se están mandando y recibiendo, Figuras 5.62 y 5.63, que son significativamente elevadas, llegando casi a 1GB de datos enviado por prueba.

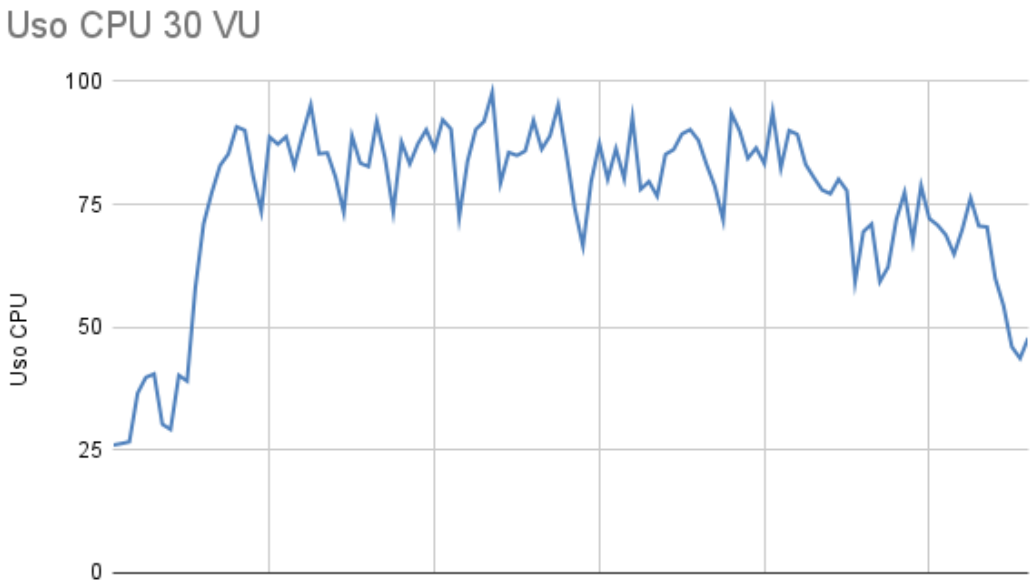

Figura 5.58: Uso de CPU para 30 VU por la aplicación original

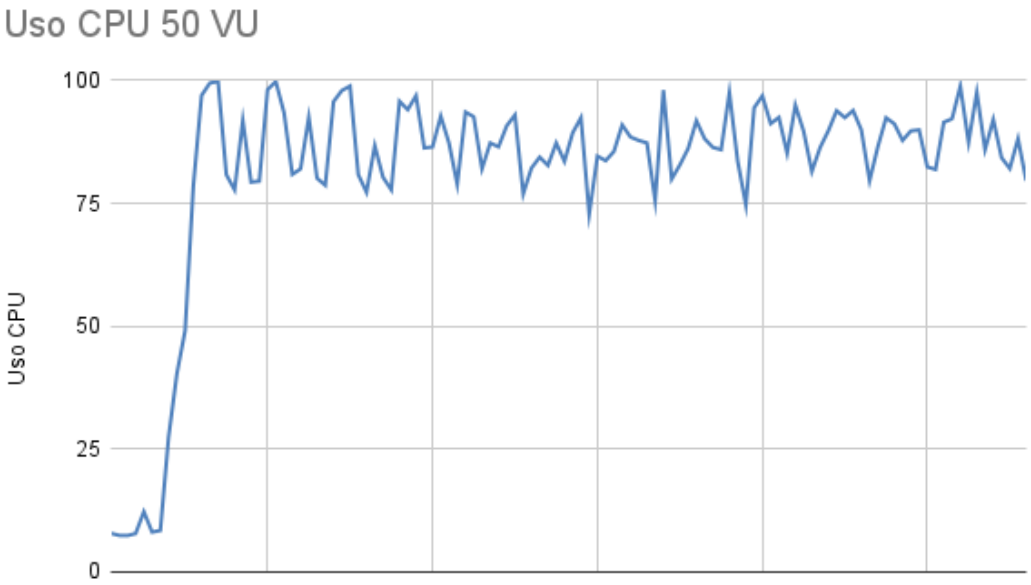

Figura 5.59: Uso de CPU para 50 VU por la aplicación original

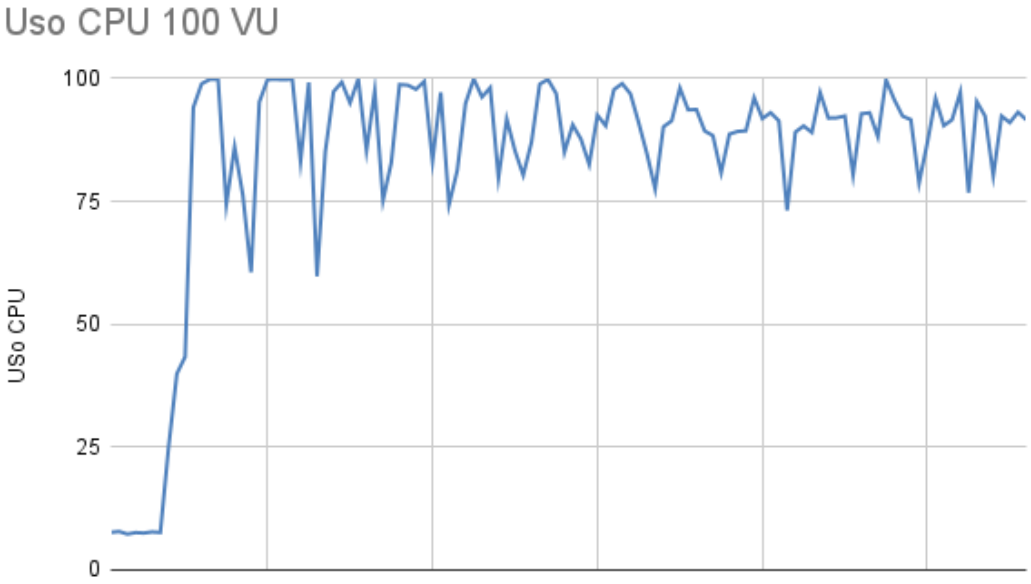

Figura 5.60: Uso de CPU para 100 VU por la aplicación original

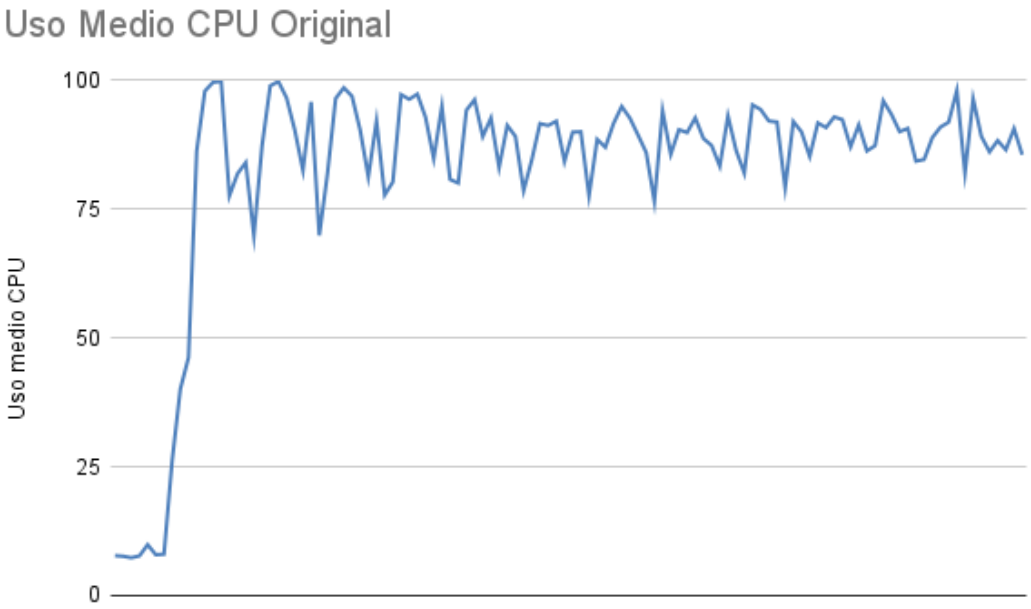

Figura 5.61: Uso medio por la aplicación original

VU	Media	Mediana	Mínimo	Máximo	p90	p95
30	75,87305357	80,66	26,006	97,76	90,312	92,396
50	81,51403571	86,62	7,396	99,8	96,898	98,085
100	83,820375	91,49	7,298	99,92	99,236	99,809
Media	82,66720536	89,28	7,347	99,83	96,5	97,976

Tabla 5.4: Uso CPU para la aplicación original

		Iteraciones						Productividad
Nº VU	Peticiones	Media	Mediana	Mínimo	Máximo	p90	p95	Iteraciones/s
30 VU	746442	5230,666667	5640	4018	6034	5955,2	5994,6	3,221314
50 VU	968792,8	7020,2	7117	6511	7364	7322,8	7343,4	4,3731455
100 VU	835535,048	6064	7563	50	7628	7618,8	7623,4	3,743209877

Tabla 5.5: Resultado pruebas de carga de la aplicación original

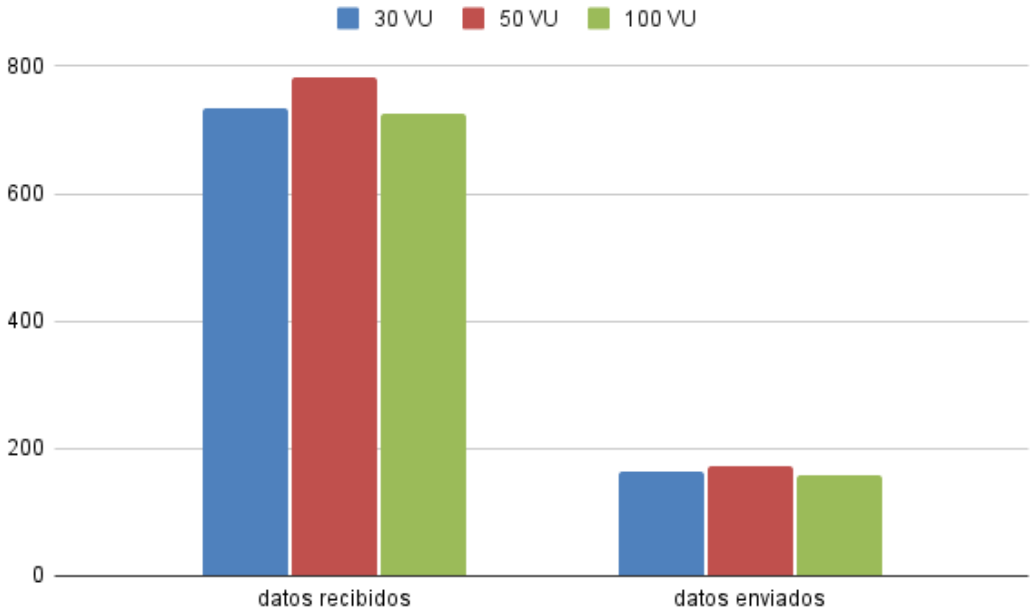

Figura 5.62: Datos enviado y recibidos por la aplicación original (MB)

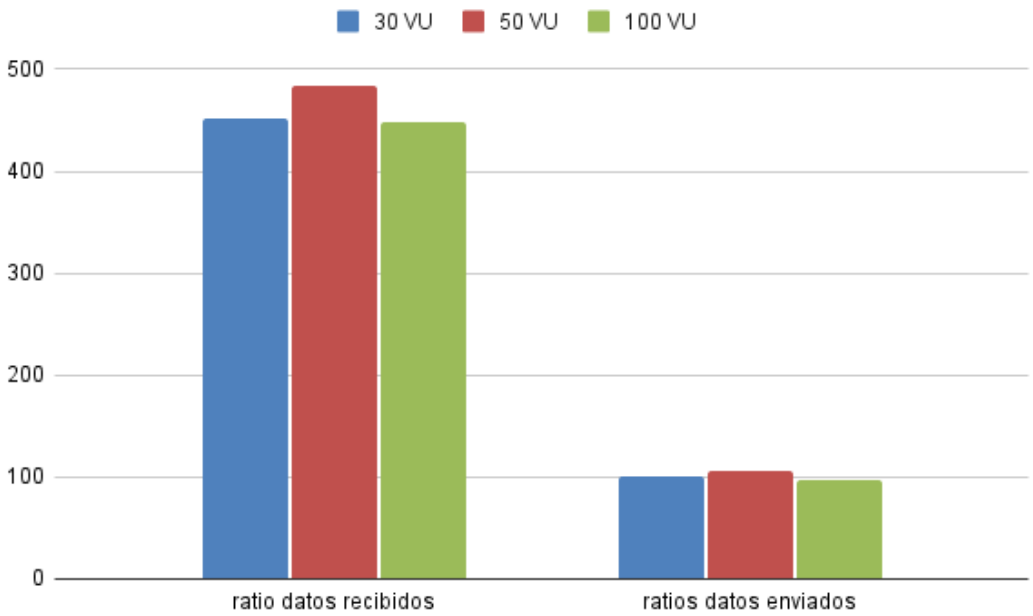

Figura 5.63: Ratio de datos enviados y recibidos por la aplicación original (KB/s)

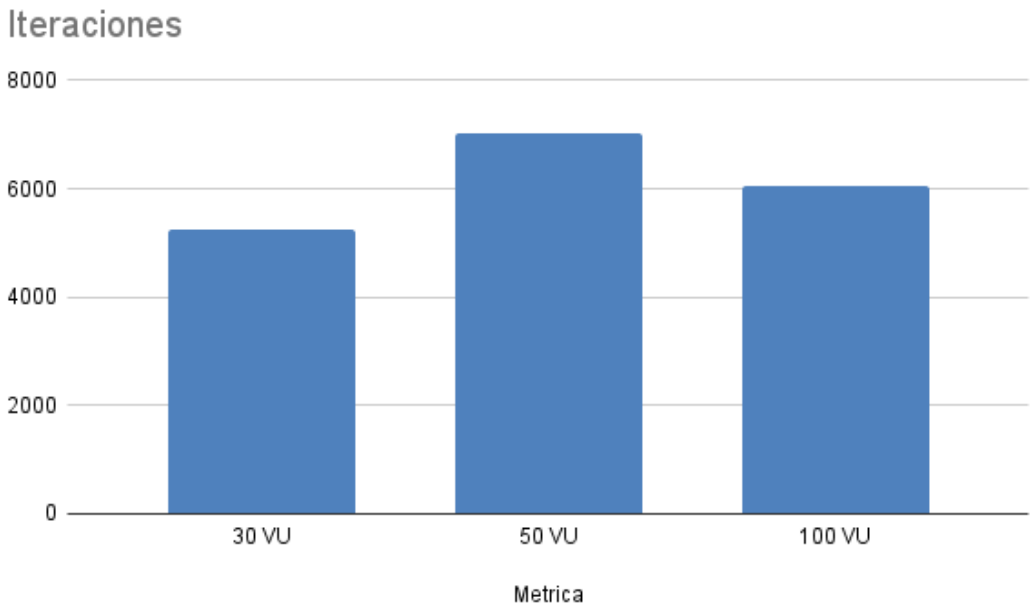

Figura 5.64: Iteraciones realizadas por la aplicación original

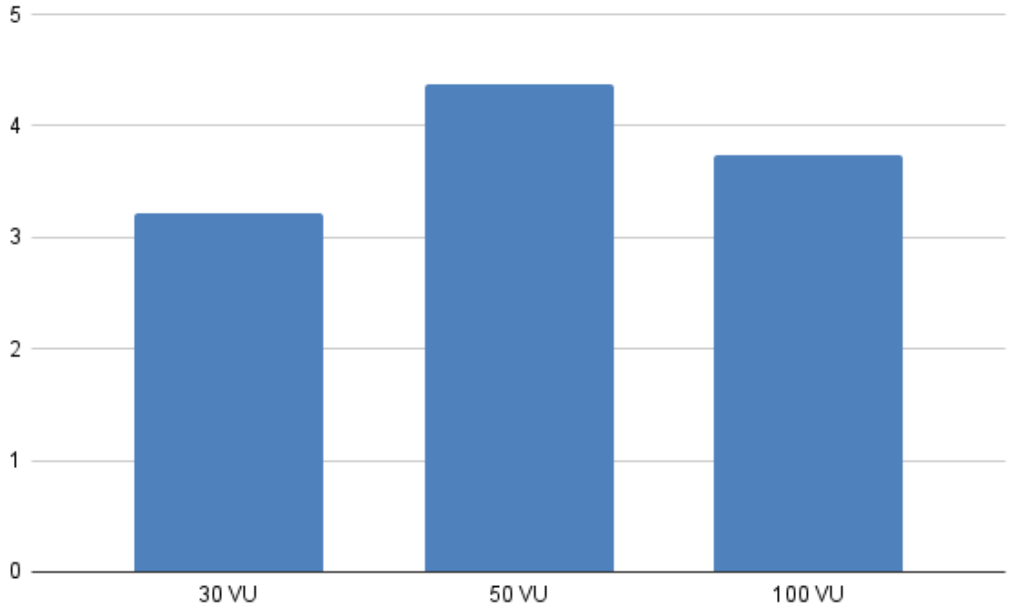

Figura 5.65: Ratio de iteraciones realizadas por la aplicación original

Resultados Modificación

En contraposición con los resultados obtenidos por parte de la aplicación en su versión original, las pruebas realizadas a la versión con las modificaciones realizadas realizadas, apenas supera el 31 % de uso de CPU de máximo, siendo su media inferior al 16 % (Figuras 5.66, 5.67, 5.68, 5.69 y Tabla 5.6, y como se muestra en la comparativa de la Figura **??**). Estos valores son muy fluctuantes, seguramente debidos a la simulación de WebSocket, o las aplicaciones de recogida de datos, junto con **Grafana** funcionando todo en el mismo sitio, aunque se aprecia una atenuación en el tiempo.

Por otro lado, se aprecia una reducción en la cantidad de datos mandados (Figuras 5.70 y 5.71) en comparación con la versión original. También existe una mejoría significativa en la cantidad de iteraciones por segundo que se resuelven, Tabla 5.7, mostrando que tiene todavía capacidad de crecimiento (Figuras 5.72 y 5.73). El número tan bajo de peticiones mostrado en la Tabla 5.7 es debido a que solo está mostrando peticiones HTTP, de las cuales se hacen menos al haber eliminado las llamadas *HTTP Polling* que se estaban haciendo con la modificación.

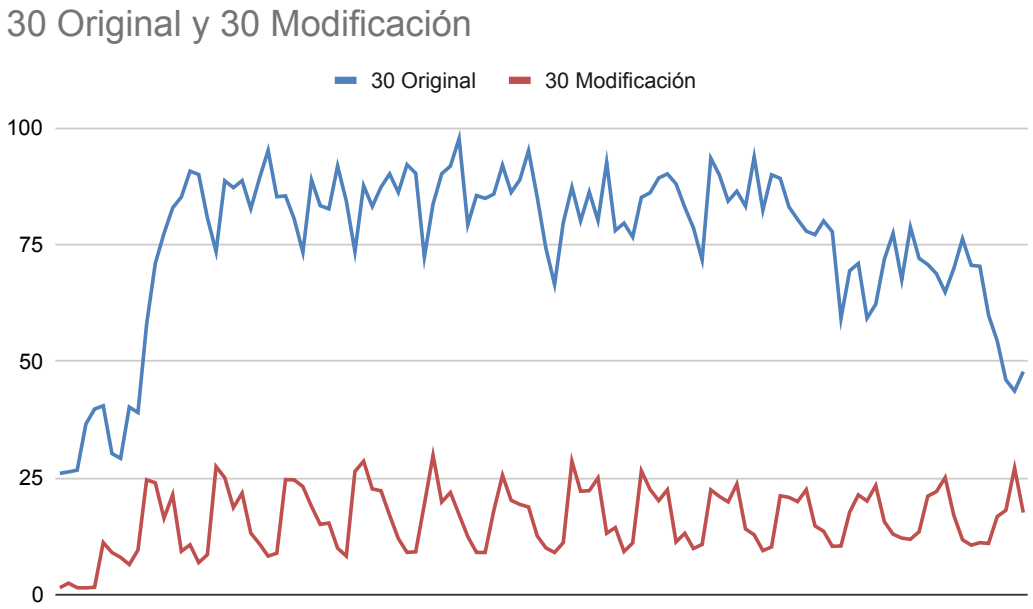

Figura 5.66: Uso de CPU para 30 VU por la aplicación modificada

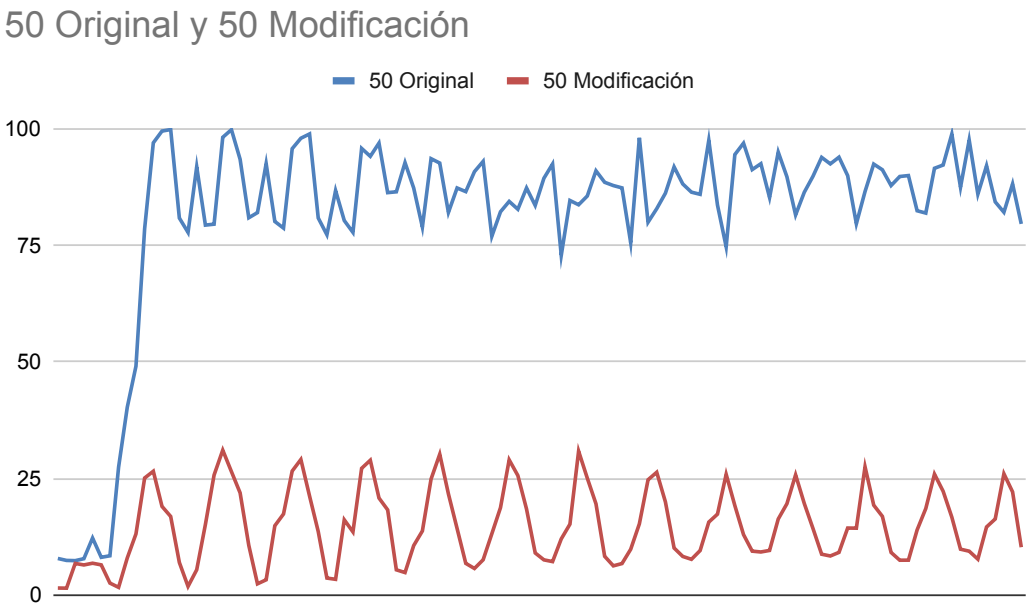

Figura 5.67: USo de CPU para 50 VU por la aplicación modificada

Figura 5.68: Uso de CPU para 100 VU por la aplicación modificada

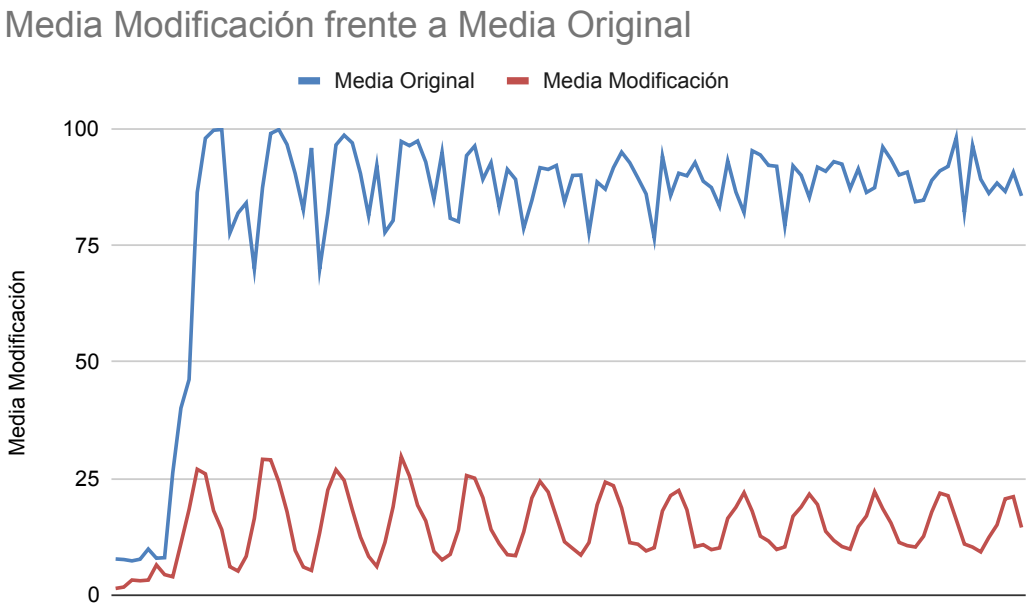

Figura 5.69: Uso medio de CPU por la aplicación modificada

VU	Media	Mediana	Mínimo	Máximo	p90	p95
30	15,96053571	15,482	1,45	29,86	24,6578	26,4628
50	14,73928571	14,289	1,496	31,08	26,308	28,15
100	13,77105357	12,408	1,238	34,16	24,968	28,4678
Media	14,823625	14,004	1,444	29,69333333	24,2642	25,79863333

Tabla 5.6: Uso CPU para la aplicación modificada

		Iteraciones						Productividad
Nº VU	Peticiones	Media	Mediana	Mínimo	Máximo	p90	p95	Iteraciones/s
30 VU	50401,75	925779,75	928668	906842	938941	938488,9	938714,95	567,3296375
50 VU	59460,8	1035559,4	1033172	1027348	1044332	1042944,4	1043638,2	633,5504004
100 VU	75947,8	1299627	1301077	1287429	1309929	1307589,8	1308759,4	790,252629

Tabla 5.7: Resultados de las pruebas de carga de la modificación

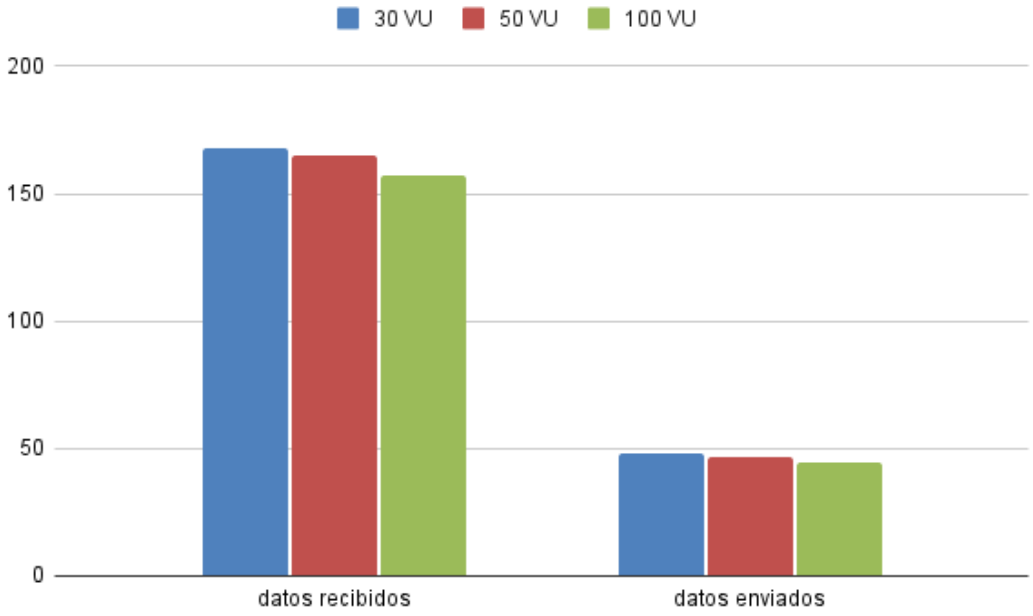

Figura 5.70: Datos enviados y recibidos por la aplicación modificada (MB)

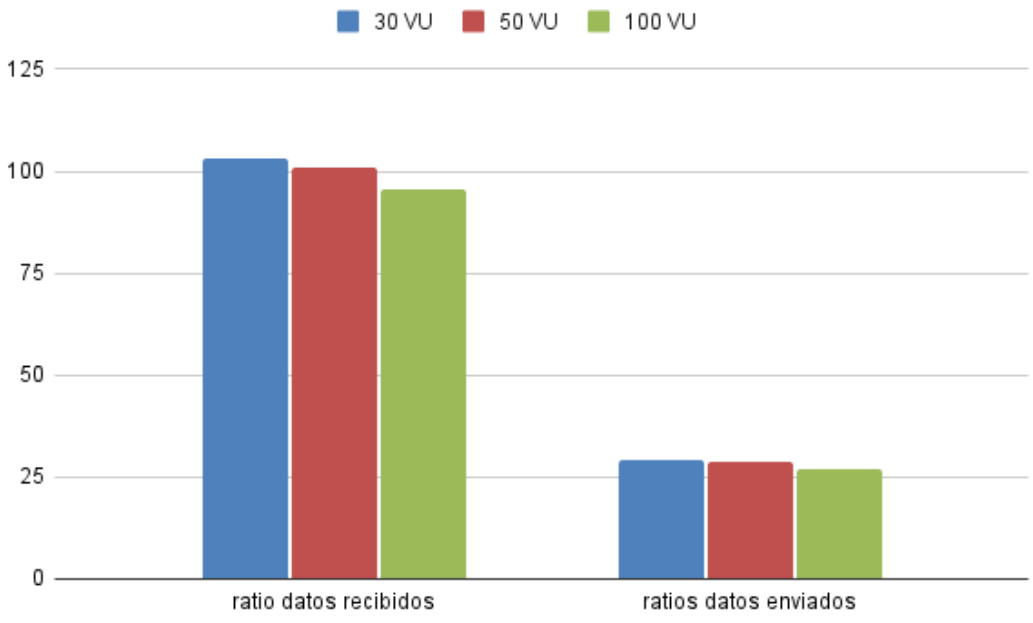

Figura 5.71: Ratio de datos enviados y recibidos por la aplicación modificada (KB/s)

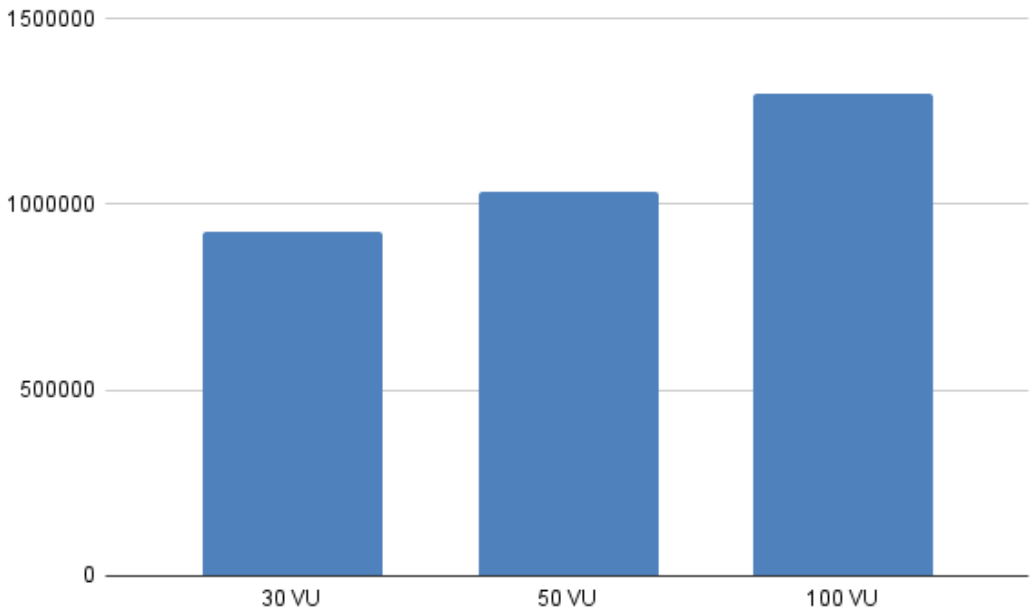

Figura 5.72: Iteraciones realizadas por la aplicación modificada

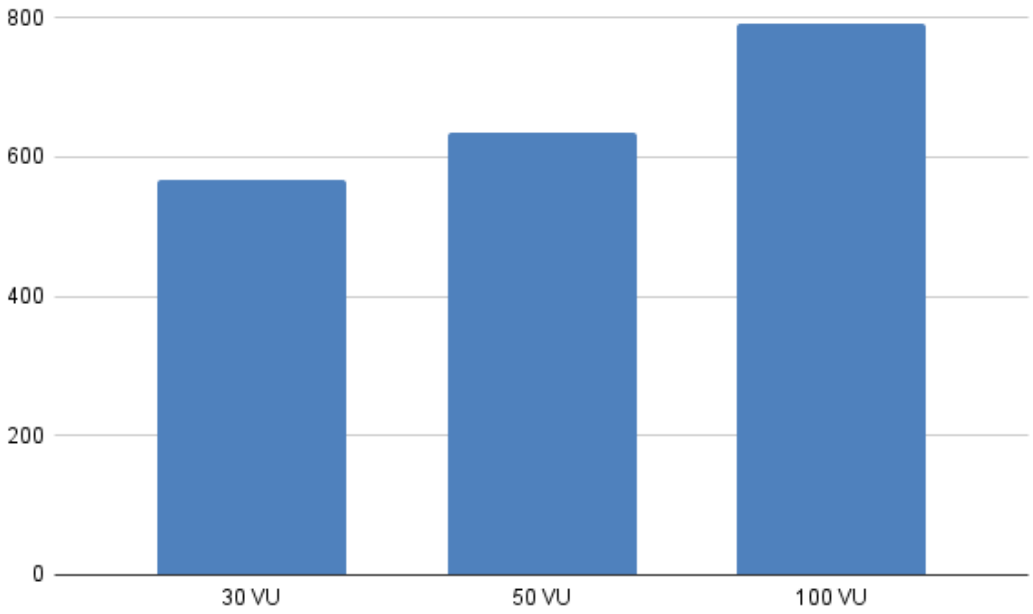

Figura 5.73: Ratio de iteraciones realizadas por la aplicación modificada

Discusión de los resultados

Tras haber analizado estos resultados, se puede decir:

- Tras la aplicación de los cambios, se observa una mejora en la eficiencia del uso de los recursos. Se resuelve el cuello de botella que existía con el uso de CPU.
- Se observa una mejora en la capacidad de respuesta del servidor.
- Se consideraría recomendable rehacer parte de los test para atenuar los fallos producidos durante las pruebas, mejorando así la exactitud de los datos obtenidos.
- Se recomendaría aumentar el numero de VU progresivamente para comprobar la capacidad de carga real con las modificaciones, y si esta es suficiente para un uso nominal de la aplicación en un entorno de producción real.

Se confirma que las modificaciones han sido útiles para mejorar la aplicación. Con estas se consigue hacer un uso mas eficiente de los recursos disponibles para dar servicio a los usuarios. De esta manera se puede seguir escalando el uso de aplicación con la arquitectura actual, pudiendo dar cabida a más usuarios concurrentes. También se recomienda mejorar las pruebas con WebSocket para encontrar el uso máximo teórico que pudiera tener, o al menos si cumple las especificaciones para el flujo nominal esperado de usuarios.

6: Conclusiones y Líneas de trabajo futuras

Una vez expuesto todo lo relativo a este trabajo, es momento de hacer una evaluación completa de los objetivos cumplidos por este. Para ello se recuperan los objetivos anteriormente expuestos y se procede a explicar su grado de realización. Tras esto se presentará unas líneas de trabajo futuro derivadas de los trabajos realizados.

Revisión del cumplimiento de objetivos

Como se ha mencionado antes, se recuperan los objetivos anteriormente expuestos en este trabajo, junto con una explicación de su grado de completitud:

- **Localizar las principales llamadas que están afectando al rendimiento de la aplicación**: De manera exitosa se ha conseguido detectar las llamadas principales causantes de la sobrecarga que producía la aplicación en el servidor.
- **Encontrar una solución que pueda aplicarse a la aplicación**: De manera exitosa se ha conseguido encontrar una solución que es usada en casos reales, WebSocket.
- **Aplicar la solución**: De manera exitosa se ha conseguido implementar la solución mediante el uso de STOMP. Estos cambios han causado problemas de la estabilidad de la aplicación, los cuales serán tratados en las líneas de trabajo futuro.
- **Comprobar si la solución mejora la situación en el servidor** De manera mayoritariamente exitosa se ha conseguido comprobar que la solución mejora la situación en el servidor. No se ha conseguido comprobar una carga real de WebSocket debido a las restricciones de tiempo con las que se encontró el proyecto. De manera similar, no se ha podido simular por completo una partida de manera correcta en las pruebas de carga, haciendo que estos datos sean una simulación. Aunque estas pruebas no hayan podido ser todo lo reales, si han servido para comprobar la mejora de rendimiento.

De esta manera se consideran alcanzados satisfactoriamente alcanzados los objetivos propuestos por este proyecto.

Líneas de trabajo futuro

Durante el desarrollo de la aplicación se observó que el alcance de las modificaciones realizadas en el proyecto fueron mayores de lo esperado. Esto provocó que surgieran varias inestabilidades en su funcionamiento, entre las que destacan:

- Si un usuario **Jugador** perdía la conexión, o se forzaba una actualización por parte del navegador, producía una inestabilidad con la transmisión de datos y el flujo nominal de la partida, impidiendo que se pudiera terminar de responder correctamente todas las respuestas.
- Se vuelve necesario recargar la página del navegador tras terminar cada partida, tanto para usuarios **Jugador**, como para usuarios **Moderador**, para evitar problemas con información almacenada en este entre partidas.
- Se detectó también un problema producido cuando, al terminar la partida, un grupo no ha terminado exitosamente ninguna ronda de juego (ha alcanzado la pantalla que muestra los resultados), pasando en este punto a un estado inconsistente, quedándose atascado en la pantalla del cuestionario, que impide al grupo llegar a la vista final de resultados de la partida.

Por otro lado, y como se comentó en las conclusiones derivadas de las pruebas,

- Se consideraría recomendable rehacer parte de los test para atenuar los fallos producidos durante las pruebas, mejorando así la exactitud de los datos obtenidos.
- Se recomendaría aumentar el numero de VU progresivamente para comprobar la capacidad de carga real con las modificaciones, y si esta es suficiente para un uso nominal de la aplicación en un entorno de producción real.

Considerando los problemas y sugerencias previamente mencionados, surgen dos nuevas líneas de trabajo:

- Encontrar soluciones a los problemas en los flujos de la aplicación y revisar y reforzar los flujos alternativos, haciendo la aplicación más resistente ante estados inconsistentes e interacciones no esperadas por parte de los usuarios.
- Mejorar los test de carga usados para confirmar, de manera más precisa, las conclusiones alcanzadas por este trabajo, mejorando también el uso que se hace de **Grafana**, y usando los csv generados por **K6** para alimentar un dashboard nuevo, creado para mejorar la revisión de estos datos por parte de interventores humanos.

Apéndices

Apéndice A

Plan de Proyecto

Este apéndice presentará el plan de proyecto elaborado para la realización del trabajo. Dicho plan también contempla la planificación de la fase de desarrollo y la explicación de los problemas que surgieron durante esta.

A.1. Planificación temporal del desarrollo del proyecto

Durante la estancia en la asignatura de *I+D+i* se planificaron las tareas de investigación necesarias para poder llevar a cabo el proyecto, de esa planificación se creó un plan de desarrollo para el proyecto (Tabla A.1).

Semana	Fecha Inicio	Fecha Fin	Descripción
1	01/05/23	07/05/23	Desarrollo e implementación de la llamada chat/get-chat a WebSocket
2	08/05/23	14/05/23	Desarrollo e implementación de la llamada form/choice a WebSocket
3	15/05/23	21/05/23	Desarrollo e implementación de la llamada room/rounds a WebSocket
4	22/05/23	28/05/23	Desarrollo e implementación de la llamada room/get-status a WebSocekt
5	29/05/23	04/06/23	Desarrollo e implementación de la llamada player/by-room a WebSocekt
6	05/06/23	11/06/23	Arreglo y resolución de problemas generados por los cambios

Tabla A.1: Plan de desarrollo

Siendo las horas reservadas para la asignatura de *I+D+i* de 190, se repartieron entre 6 semanas, permitiendo tiempo y margen de maniobra para problemas no previstos en el momento de desarrollo. Esto se debe a la necesidad de modificar en profundidad una aplicación compleja (rearquitectura), usando tecnologías en las que no se estaba especializado (Maven y WebSocket), por lo que se esperaban complicaciones.

Debido a estas complicaciones hubo que intentar readaptar el calendario puesto que no se estaban pudiendo cumplir ninguno de los objetivos originalmente propuestos. La complejidad de la implementación de WebSocket en esta aplicación por parte de Maven no fue considerada correctamente. Por otra parte, y habiéndose terminado el tiempo de planificación sin muchos avances, surgió un problema personal por el que se tuvo que abandonar el proyecto durante 2 años, retrasando todavía más su desarrollo.

A.2. Planificación real

Cuando se pudo volver a trabajar en el proyecto, se encontró rápido una solución y se volvió a reestructurar el plan original, dando esto resultado a una calendarización real del mismo (Tabla A.2). Esta se vio extensamente modificada con el plan anterior debido a las mismas complicaciones de complejidad. También se tardó más tiempo del pensado ya que no se podía emplear de este en el desarrollo debido a las misa causas externas anteriormente mencionadas. Tampoco se le pudo dedicar las semanas enteras de trabajo que se muestran, por lo que se estima que se ha podido dedicar un tiempo efectivo de entre el 50-70 % de una jornada laboral completa por cada semana de trabajo representada en la tabla A.2, haciendo esto un total de entre 360-504 horas de trabajo efectivo, sin tener en cuenta la ayuda recibida por parte de consultas externas que se realizaron.

Semana	Fecha Inicio	Fecha Fin	Descripción
1	01/04/25	06/04/25	Desarrollo e implementación de la llamada chat/get-chat a WebSocket
2	07/04/25	13/04/25	Actualización de Stomp en el *frontend* implementación de la llamada form/choice a WebSocket
3	14/04/25	20/04/25	Desarrollo e implementación del resto de llamadas pendientes a WebSocket
4	21/04/25	27/04/25	Implementación de las funciones *Send...()* para poder enviar datos al socket
5	28/04/25	04/05/25	Arreglo de errores producidos por las implementaciones
6	05/05/25	11/05/25	Arreglo de errores producidos por la implementación
7	12/05/25	18/05/25	Arreglo de errores y nueva funciones *Send*
8	19/05/25	25/05/25	Arreglos y fin del desarrollo
9	26/05/25	31/05/25	Despliegue en la máquina virtual
10	01/06/25	07/06/25	Inicio de las pruebas de carga con JMeter
11	08/06/25	14/06/25	Continuación de pruebas de carga con JMeter
12	15/06/25	21/06/25	Análisis de WebSocket en JMeter
13	07/07/25	13/07/25	Búsqueda de alternativas para implementar WebSocket JMeter
14	16/07/25	20/08/02	Búsqueda de alternativas para JMeter
15	25/08/25	31/08/25	Implementación de Grafana
16	25/08/25	31/08/25	Implementación de Grafana
17	01/09/25	07/09/25	Implementación de módulos auxiliares, Prometheus, Apache_exporter y Telegraf
18	08/09/25	14/09/25	Realización de las pruebas de carga con k6

Tabla A.2: Planificación real

Coste del proyecto

Teniendo en cuenta la cantidad de horas estimadas, se puede hacer un calculo estimado del coste de desarrollo del proyecto. Esto se hará usando las referencias del BOE. A fecha de escritura, se ha podido localizar «BOE» núm. 218, de 11 de septiembre de 2021, páginas 110953 a 110957 (5 págs.), donde se estipulan los siguientes precios, mostrados en la Tabla A.3

Personal	Euros/Hora
Analista/Estadístico superior	57
Programador/Estadístico Técnico/Documentalista	42
Operador/Auxiliar	34

Tabla A.3: Planificación real

Teniendo estas consideraciones en cuenta, se calcula el precio mostrado como personal **Programador/Estadístico Técnico/ Documentalista** por lo que el precio estimado por horas de trabajo sería entre 15120 a 21168 euros. También hay que tener en cuenta el equipo usado para su desarrollo, en este caso **ASUS Vivobook 15** con una capacidad de 16GB de RAM valorado en 499 Euros. Este coste prorrateado es de 61.75 por el tiempo medio que se ha usado de trabajo. Esto hace un total de entre 15181.75 a 21229.75 Euros de coste total por su desarrollo.

Apéndice B

Documentación del Programador

En este apéndice se incluye la documentación necesaria para que el programador de aplicaciones pueda comprender la estructura de la solución software aportada y para poder modificarla.

B.1. Introducción

Para reproducir este trabajo tiene como pre-requisito una máquina Ubuntu con las siguientes prestaciones:

- CPU con 4 cores
- Memoria RAM de 16 GB
- Disco duro de 50GB

También se necesitará instalar **Git**, **Apache2**, **Maven Grafana**, **Prometheus**, **Apache_Exporter** y **Telegraf**. A parte de esto se recomienda una máquina extra desde la que lanzar las pruebas de carga con **K6**

B.2. Estructura de directorios

La estructura de directorios es la que maneja Ubuntu con la instalación de todos estos programas. Se recomienda crear una carpeta donde se encuentre el proyecto de *crossroads-backend* para facilitar las rutas de ejecución de la aplicación. Se añadirá a la ruta **/var/www/html**, la carpeta con la compilación de los artefactos del proyecto *crossroads-frontend*. Todos los demás archivos que se usen tendrán la ruta colocada, junto a su nombre previo al código que tengan.

B.3. Manual del programador

Instalación de **Git**

```
sudo dnf install git-all
```

Los repositorios actuales son https://gitlab.inf.uva.es/tfm-roberto/crossroads-frontend apuntando a la rama **subscriberSocket** y https://gitlab.inf.uva.es/tfm-roberto/crossroads-backend apuntando a la rama **websocket-stomp**.

Se clonan los repositorios y se mueven a la raíz.

```
mkdir TFM
cd TFM
git clone https://gitlab.inf.uva.es/tfm-roberto/crossroads-frontend
git clone https://gitlab.inf.uva.es/tfm-roberto/crossroads-backend
sudo mv TFM /
```

Instalación de **Maven**

```
sudo apt install maven
```

Se crea un servicio para que arranque automáticamente el back tras cualquier recuperación del servidor
/etc/systemd/system/arranque.service

```
[Unit]
Description=Arranca Back.

[Service]
ExecStart=/usr/bin/bash -c
    'mvn --log-file /TFM/mvn.log \
    -f /TFM/crossroads-backend spring-boot:run'

[Install]
WantedBy=multi-user.target
```

Se inicia el servicio

```
sudo systemctl enable arranque.service
```

Se instala **Apache**

```
sudo apt install apache2
sudo a2enmod proxy
sudo a2enmod proxy_http
sudo a2enmod rewrite
sudo a2enmod status
```

Se modifica la configuración por defecto que trae para que se adapte a todos los cambios que vamos a realizar

/etc/apache2/sites-available/000-default.conf

```
<VirtualHost *:80>
    ServerAdmin webmaster@localhost
    DocumentRoot /var/www/html

    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined
    LogFormat "%v %h %l %u %t \"%r\" %>s %O \"%{Referer}i\"\
        "%{User-Agent}i\" %D" telegraf
    CustomLog /var/log/apache2/telegraf_access.log telegraf

    ServerName localhost
    DocumentRoot /var/www/html/crossroads-frontend
    <Directory "/var/www/html/crossroads-frontend">
        Options Indexes FollowSymLinks
        AllowOverride all
        Require all granted

        RewriteEngine on
        RewriteBase /
        RewriteRule ^../index\.html$ - [L]

        RewriteCond %{REQUEST_FILENAME} !-f
        RewriteCond %{REQUEST_FILENAME} !-d
        RewriteRule (..) $1/index.html [L]

        RewriteCond %{HTTP:Accept-Language} ^de [NC]
        RewriteRule ^$ /de/ [R]

        RewriteCond %{HTTP:Accept-Language} ^en [NC]
        RewriteRule ^$ /en/ [R]

        RewriteCond %{HTTP:Accept-Language} !^en [NC]
        RewriteCond %{HTTP:Accept-Language} !^de [NC]
        RewriteRule ^$ /es/ [R]
```

```
    </Directory>
# Proxy para conexion backend
    ProxyPreserveHost On
    ProxyPass        /api  http://127.0.0.1:8080/api
    ProxyPassReverse /api  http://127.0.0.1:8080/api

# Proxy para WebSocket
    ProxyPass        /ws   ws://127.0.0.1:8080/ws
    ProxyPassReverse /ws   ws://127.0.0.1:8080/ws

# Proxy para Grafana
    ProxyPreserveHost On
    ProxyPass        /grafana   http://127.0.0.1:3000
    ProxyPassReverse /grafana   http://127.0.0.1:3000

# Proxy para Pyroscope
    ProxyPreserveHost On
    ProxyPass        /pyroscope   http://127.0.0.1:4040
    ProxyPassReverse /pyroscope   http://127.0.0.1:4040

# Proxy para Prometheus
    ProxyPreserveHost On
    ProxyPass        /prometheus   http://127.0.0.1:9000
    ProxyPassReverse /prometheus   http://127.0.0.1:9000

# Proxy para Apache-Exporter
    ProxyPreserveHost On
    ProxyPass        /apache-exporter   http://127.0.0.1:9117
    ProxyPassReverse /apache-exporter   http://127.0.0.1:9117

# Proxy para Telefraf
    ProxyPreserveHost On
    ProxyPass        /telegraf   http://127.0.0.1:9273
    ProxyPassReverse /telegraf   http://127.0.0.1:9273

</VirtualHost>
```

Dentro del proyecto de *crossroads-frontend* se lanza el siguiente comando

```
npm install
npm run build
```

Tras esto se ha generado el artefacto dentro de la carpeta /dist llamado **crossroads-frontend**

```
cd dist/
sudo mv -rf corssroads-frontend /var/www/html
```

Darle permisos a la carpeta para que pueda ser accedida correctamente por Apache y se inicia el servicio

```
chown -R www-data:www-data /var/www/html
sudo systemctl enable apache2
```

Instalamos Grafana

```
sudo apt-get install -y adduser libfontconfig1 musl
wget https://dl.grafana.com/grafana-enterprise/release/12.1.1/
    grafana-enterprise_12.1.1_16903967602_linux_amd64.deb
sudo dpkg -i grafana-enterprise_12.1.1_16903967602_linux_amd64.deb
```

Modificamos el archivo de configuración para que apunte correctamente a la ruta que le hemos indicado en **Apache /etc/grafana/grafana.ini** (comprobar que las siguientes lineas están correctamente formadas

```
# The http port to use
http_port = 3000

# The public facing domain name used to access grafana from a browser
domain = 127.0.0.1

# Redirect to correct domain if host header does not match domain
# Prevents DNS rebinding attacks
;enforce_domain = false

# The full public facing url you use in browser, used for redirects and emails
# If you use reverse proxy and sub path specify full url (with sub path)
root_url = %(protocol)s://%(domain)s:%(http_port)s/grafana

# Serve Grafana from subpath specified in `root_url`
# setting. By default it is set to `false` for compatibility reasons.
serve_from_sub_path = false
```

Se habilita el servicio

```
sudo systemctl enable grafana
```

Se instala **Prometheus** y se habilita como usuario

```
    sudo groupadd --system prometheus
    sudo useradd -s /sbin/nologin --system -g prometheus prometheus
    sudo mkdir /var/lib/prometheus
    for i in rules rules.d files_sd; do sudo mkdir -p /etc/prometheus/${i};
        done
    tar xvfz prometheus-*.tar.gz
    cd prometheus-*
    sudo mv prometheus promtool /usr/local/bin/
    sudo mv prometheus.yml  /etc/prometheus/prometheus.yml
    rm -rf /tmp/prometheus
```

Se modifica la configuración
/etc/prometheus/prometheus.yml

```
# my global config
global:
  # Set the scrape interval to every 15 seconds. Default is every 1 minute.
  scrape_interval:     15s
  # Evaluate rules every 15 seconds. The default is every 1 minute.
  evaluation_interval: 15s
  # scrape_timeout is set to the global default (10s).

# Alertmanager configuration
alerting:
  alertmanagers:
  - static_configs:
    - targets:
      # - alertmanager:9093

# Load rules once and periodically evaluate them
#according to the global 'evaluation_interval'.
rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
# Apache Servers
  - job_name: apache-telegraf
    metrics_path: '/telegraf/metrics'
    static_configs:
      - targets: ['virtual.lab.inf.uva.es:20262']
        labels:
```

```
        alias: apache-telegraf

- job_name: apache-exporter
  metrics_path: '/apache-exporter/metrics'
  static_configs:
    - targets: ['virtual.lab.inf.uva.es:20262']
      labels:
        alias: apache-exporter
```

Y se crea un servicio que maneje **Prometheus**
/etc/systemd/system/prometheus.service

```
[Unit]
Description=Prometheus
Documentation=https://prometheus.io/docs/introduction/overview/
Wants=network-online.target
After=network-online.target

[Service]
Type=simple
Environment="GOMAXPROCS=4"
User=prometheus
Group=prometheus
ExecReload=/bin/kill -HUP $MAINPID
ExecStart=/usr/local/bin/prometheus \
  --config.file=/etc/prometheus/prometheus.yml \
  --storage.tsdb.path=/var/lib/prometheus \
  --web.console.templates=/etc/prometheus/consoles \
  --web.console.libraries=/etc/prometheus/console_libraries \
  --web.listen-address=0.0.0.0:9000 \
  --web.external-url=http://virtual.lab.inf.uva.es:20262/prometheus \
  --web.route-prefix=/

SyslogIdentifier=prometheus
Restart=always

[Install]
WantedBy=multi-user.target
```

La variable $Environment = "GOMAXPROCS = 4"$ indica el nº de **vcpus** de la máquina, en este caso son 4

Se habilita

```
sudo systemctl daemon-reload
sudo systemctl enable prometheus
```

Se instala **Apache_Exporter**

```
tar xvf apache_exporter-*.linux-amd64.tar.gz
sudo cp apache_exporter-*.linux-amd64/apache_exporter /usr/local/bin
sudo chmod +x /usr/local/bin/apache_exporter
```

Se añade como servicio
/etc/systemd/system/apache_exporter.service

```
[Unit]
Description=Prometheus
Documentation=https://github.com/Lusitaniae/apache_exporter
Wants=network-online.target
After=network-online.target

[Service]
Type=simple
User=prometheus
Group=prometheus
ExecReload=/bin/kill -HUP $MAINPID
ExecStart=/usr/local/bin/apache_exporter \
  --insecure \
  --scrape_uri=http://localhost:8081/server-status/?auto \
  --telemetry.endpoint=/metrics

SyslogIdentifier=apache_exporter
Restart=always

[Install]
WantedBy=multi-user.target
```

Se habilita

```
sudo systemctl daemon-reload
sudo systemctl enable apache_exporter.service
```

Se abre un puerto local
/etc/apache2/ports.conf

```
Listen 127.0.0.1:8081
```

Se crea un nuevo VirtualHost
/etc/apache2/sites-available/status-local.conf

```
<VirtualHost 127.0.0.1:8081>
    ServerName localhost

    <Location /server-status>
        SetHandler server-status
        Require local
    </Location>

    # (Opcional) evita logs ruidosos
    CustomLog /var/log/apache2/status_access.log combined
    ErrorLog  /var/log/apache2/status_error.log
</VirtualHost>
```

Se activa y recarga **Apache2**

```
sudo a2ensite status-local.conf
sudo apache2ctl configtest
sudo systemctl reload apache2
```

Se instala **Telegraf**

```
sudo apt install telegraf
```

Se modifica la configuración necesaria para el dashboard
/etc/telegraf/telegraf.conf

```
...
[[outputs.prometheus_client]]
    listen = ":9273"
...
[inputs.cpu]]
    percpu = true
[[inputs.disk]]
[[inputs.io]]
[[inputs.mem]]
[[inputs.net]]
[[inputs.system]]
[[inputs.swap]]
[[inputs.netstat]]
[[inputs.processes]]
```

```
[[inputs.kernel]]
[[inputs.diskio]]
[[inputs.procstat]]
    user = "root,telegraf"
#Apache2 Metrics
[[inputs.apache]]
  urls = ["http://localhost:8081/server-status?auto"]
  response_timeout = "5s"
[[inputs.tail]]
  name_override = "apache2log"
  files = ["/var/log/apache2/access.log"]
  from_beginning = true
  pipe = false
  data_format = "grok"
```

Se habilita el servicio

```
sudo systemctl enable telegraf
```

Se reinicia el equipo

```
sudo systemctl reboot
```

Dashboard de Grafana se añadirá como archivo extra.

Pruebas del sistema

Como se ha explicado antes, se recomienda instalar la herramienta **K6** en un equipo a parte para lanzar las pruebas. La documentación de esto se encuentra en la página correspondiente https://grafana.com/docs/k6/latest/set-up/install-k6/?pg=get&plcmt=selfmanaged-box10-cta1

Para **K6 Studio** https://grafana.com/docs/k6/latest/set-up/install-k6/?pg=get&plcmt=selfmanaged-box10-cta1#k6-studio

Dada las dimensiones de los archivos generados para las pruebas, estos serán puestos a disposición en un git.

Apéndice C

Glosario

Glosario de Terminos

IAMs: *Integrated Assessment Models*, Modelos de Evaluación Integrados.

HTTP: *HyperText Transfer Protocol* Protocolo de Transferencia de HiperTexto. Este es un protocolo de comunicación usado para el paso de mensajes a través de la red

URI: *Uniform Resource Identifier*, Identificador de Recursos Único. Es una cadena de caracteres que identifica de manera univoca un recurso abstracto o físico.

Patrón Observador: Patrón de diseño software, en el cual un objeto llamado ***subject*** mantiene una lista de dependientes llamados ***observers*** qa los cuales les avisa de cualquier cambio en el estado.

Frame: Parte, si son varios, o total del mensaje enviado a través de la conexión WebSocket que contiene los datos. Dependiendo de la estructura en el mensaje puede tener unos *flags* activos:

- Si es un mesnaje no fragmentado, el *frame* tendrá los *flags* de ***FIN*** a 1 y el de ***opcode*** distinto de 0.
- Si el mensaje está fragmentado, consistirá de un primer *frame* con los *flags* ***FIN*** a 0 y ***opcode*** distinto de 0, seguido de cero o más *frames* con ***FIN*** a 0 y ***opcode*** a 0, y un *frame* final con ***FIN*** con valor 1 y ***opcode*** con valor 0.

Keep-alive: Señal enviada en las conexiones TCP para mantenerla activa. Si al enviar la señal no se obtiene respuesta, se asume que la conexión se ha perdido.

Broker: Módulo que sirve para traducir mensajes del protocolo emisor al protocolo receptor.

Fallback: Método usado de manera alternativa en caso de que el principal falle.

Token: Factor de autenticación usado para habilitar la comunicación, suele ser una cadena alfanumérica.

Bibliografía

[1] Alda Peñafiel, M. Definición y automatización de pruebas de carga y escalabilidad para una aplicación web colaborativa, 2022. [Internet; revisado 23-junio-2025].

[2] Alexey Melnikov, I. F. Rfc 6455, 2011. [Internet; revisado 30-agosto-2025].

[3] Clements, P. F. B. L. B. D. G. J. I. R. L. P. M. R. N. J. S. *Documenting Software Architectures: Views and Beyond, Second Edition.* International series of monographs on physics. Boston: Addison-Wesley, 2010.

[4] Europea, U. Climate change, 2023. [Internet; revisado 23-junio-2025].

[5] Europea, U. Eu challenges and priorities: young europeans' views, 2025. [Internet; revisado 23-junio-2025].

[6] Famili, S. Using stompjs, 2025. [Internet; revisado 24-agosto-2025].

[7] Fiorenza, I. Refactorización: ¿qué es y por qué es importante?, 2023. [Internet; revisado 11-septiembre-2025].

[8] geeksforgeeks. Http long polling vs websockets, 2024. [Internet; revisado 26-mayo-2025].

[9] geeksforgeeks. Websockets in microservices architecture, 2024. [Internet; revisado 26-mayo-2025].

[10] geeksforgeeks. What is long polling and short polling ?, 2024. [Internet; revisado 30-agosto-2025].

[11] Grafana. Dashboard anything. observe everything., 2025. [Internet; revisado 31-agosto-2025].

[12] Grafana. Dashboard anything. observe everything., 2025. [Internet; revisado 4-septiembre-2025].

[13] Grafana. Grafana k6, 2025. [Internet; revisado 9-septiembre-2025].

[14] Grafana. Prometheus data source, 2025. [Internet; revisado 7-septiembre-2025].

[15] Gregor Hohpe, B. W. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley Longman Publishing, 2003.

[16] influxdata. Telegraf documentation, 2025. [Internet; revisado 7-septiembre-2025].

[17] kirti.nehra@opstree.com. Apache2, 2025. [Internet; revisado 7-septiembre-2025].

[18] LOCOMOTION. Locomotion, 2020. [Internet; revisado 23-junio-2025].

[19] Lusitaniae. Lusitaniae / apache_exporter, 2025. [Internet; revisado 7-septiembre-2025].

[20] María Luciana Roldán, Silvio Gonnet, H. L. Representación de la evolución y refactoring de arquitecturas de software mediante la aplicación y captura de operaciones arquitectónicas. *Revista Tecnología y Ciencia Nº 27* (2013), 197–213.

[21] Medium. System design basics: Websockets, 2021. [Internet; revisado 26-mayo-2025].

[22] Melnikov, F. . Websocket uris, 2021. [Internet; revisado 11-septiembre-2025].

[23] mozilla. Códigos de estado de respuesta http, 2025. [Internet; revisado 28-mayo-2025].

[24] Mutai, J. Install prometheus server on centos 7 / rhel 7, 2024. [Internet; revisado 7-septiembre-2025].

[25] Mutai, J. Monitor apache with prometheus and grafana in 5 minutes, 2024. [Internet; revisado 7-septiembre-2025].

[26] Obregon, A. The mechanics behind how spring boot handles websockets with stomp messaging, 2025. [Internet; revisado 25-agosto-2025].

[27] Org, M. Websocket: readystate property, 2025. [Internet; revisado 11-septiembre-2025].

[28] rfmoz. Apache, 2025. [Internet; revisado 7-septiembre-2025].

[29] Rina Diane Caballar, C. S. ¿qué es la refactorización del código?, 2025. [Internet; revisado 11-septiembre-2025].

[30] Spring-Boot. Stomp support, 2025. [Internet; revisado 24-agosto-2025].

[31] theknowledgeacademy. What is client-server architecture? explained in detail, 2025. [Internet; revisado 13-mayo-2025].

[32] whatwg. Websockets, 2024. [Internet; revisado 26-mayo-2025].

[33] xpertlab. Websocket architecture, 2024. [Internet; revisado 26-mayo-2025].