Máster en Cloud Apps
Desarrollo y despliegue de aplicaciones en la nube

Curso académico 2019/2020

Trabajo de Fin de Máster

Elastic & Fault Tolerant GroupChat Application

Autor: Miguel Soriano Carceller
Tutor: Micael Gallego Carrillo

Para Sara, a la tercera va la vencida.

Resumen

En este proyecto se aborda el desarrollo de una aplicación de mensajería multiusuario, basada en Websockets, elástica y tolerante a fallos que se pueda desplegar en un clúster de Kubernetes.
Para ello nos vamos a basar en Vert.x, un toolkit para el desarrollo de aplicaciones orientadas a eventos.

El desarrollo de la solución está estructurado en los siguientes bloques:

- Aplicación servidor de chat: el Backend de la solución, basado en el toolkit Vert.x y desplegado sobre Kubernetes
- Librería de mensajes: desarrollada en Node, para ser utilizada por aplicaciones cliente o para realizar test sobre el Backend desde consola.
- Aplicación cliente de ejemplo: desarrollada sobre Angular y en la que se puede ver la aplicación práctica de la librería de mensajes.
- Conjunto de scripts de Test que hacen uso de la librería de mensajes, para validar la escalabilidad de la solución y su tolerancia a fallos.

Realizaremos despliegues de esta tanto en un clúster de Kubernetes local como en algún proveedor cloud.

Contenido

Índice de Imágenes

1. Introducción

Entendemos la escalabilidad como la capacidad de adaptación y respuesta de un sistema a las variaciones de rendimiento a medida que el número de usuarios o de transacciones que debe realizar aumentan. Partiendo de esta premisa existen dos tipos de escalabilidad:

- El primer tipo, el escalado vertical, se basa en añadir más recursos a un nodo de la red, por ejemplo: añadir más memoria, aumentar la capacidad del procesador, etc. cuando es necesario.
- El segundo, el escalado horizontal, se basa en añadir más nodos de procesamiento al sistema.

Hasta hace relativamente poco tiempo el procedimiento más habitual para escalar sistemas era el escalado vertical, es decir, aumentar progresivamente los recursos de que se disponía a medida que aumentaba la necesidad, lo que suponía fuertes inversiones en hardware y aumentaba la complejidad de los sistemas. Este tipo de escalado no supone problemas sobre el software de las aplicaciones, ya que todo el cambio se realiza sobre el hardware. Por lo tanto, es más fácil de implementar que el escalado horizontal y puede ser una solución rápida. Y en ocasiones, más económica si el software ya está implementado. El gran inconveniente de este tipo de escalado es que está limitado por el hardware, cuyo crecimiento es limitado y presenta un punto de fallo crítico: si se produce un fallo del servidor se producirá una pérdida de servicio (descartando aquí sistemas de respaldo, etc.).

A diferencia del anterior, el escalado horizontal proporciona un crecimiento tan grande como se desee, agregando tantos servidores como sean necesarios. Soporta alta disponibilidad, ya que si un nodo falla el tráfico se puede repartir entre los nodos restantes mediante un balanceador de carga.
Pero este tipo de escalado presenta una serie de inconvenientes:

- La infraestructura es más compleja de mantener al componerse de muchos más elementos (más nodos, balanceadores de carga…).
- Si las aplicaciones ya estaban diseñadas, y no lo fueron para trabajar en clúster requiere de grandes cambios en las mismas.

Llegados a este punto cobra vital importancia el desarrollo de sistemas distribuidos, sistemas de software cuyos componentes están separados físicamente y conectados entre sí para lograr una meta común. Para comunicar los nodos de un sistema distribuido es necesario algún tipo de software que proporcione un enlace entre instancias independientes de la aplicación. Hay distintos tipos, pero en

este caso nos enfocaremos en aquellos que están orientados a mensajes. Los dos modelos más comunes son:

- **Publish-Subscribe (pub/sub)** – Los emisores (publishers) no envían los mensajes directamente a clientes (suscriptores) específicos, sino que etiquetan su mensaje con una o más categorías (en nuestro caso será el nombre de la sala) y lo publican. De manera similar, los suscriptores se suscriben a categorías (en nuestro caso salas de chat) ignorando si hay emisores publicando mensajes en ellas.

- **Cola de Mensajes**: El publicador envía mensajes a una cola donde son guardados hasta que el receptor los recibe. A diferencia de *publish–subscribe*, en la cola de mensajes el receptor tiene que ir activamente a la misma y buscar nuevos mensajes.

Nuestro sistema de chat usará el patrón *publish–subscribe*, apoyado en el Bus de Eventos proporcionado por el toolkit Vert.x. Al tener bajo acoplamiento, este modelo es más fácil de escalar horizontalmente.

2. Objetivos

El principal objetivo del proyecto es la mejor comprensión de aspectos tratados durante la realización del Máster Cloud Apps, tales como arquitectura de software, tecnologías y servicios de internet, persistencia con bases de datos, API Rest, contenedores y orquestadores (Docker y Kubernetes) despliegue de aplicaciones, sistemas escalables y distribuidos, la tolerancia a fallos, sistemas y metodologías de gestión de código fuente, etc.

Con este fin, se abordará el desarrollo de una aplicación de chat multiusuario, escalable y tolerante a fallos, que pueda ser desplegada tanto en un clúster de Kubernetes local como en proveedores cloud.

Nos apoyaremos en un proyecto realizado por Michel Maes Bermejo, en el que realizó una comparativa entre diferentes tecnologías (Akka, Vert.x y SpringBoot) de sistemas distribuidos para implementación de chats multiusuario basadas en Websockets.

3. Tecnologías y Herramientas

En este capítulo se describen las principales tecnologías y herramientas utilizadas para el desarrollo del proyecto, entendiendo tecnologías como los diferentes protocolos, lenguajes de programación, frameworks, etc., y herramientas como instrumentos para gestión del código fuente, infraestructuras de despliegue, entornos de desarrollo, etc.

3.1. Tecnologías

3.1.1. Websockets

Figura 1. Logo Websockets

Los *WebSockets* nos permiten trabajar de forma bidireccional entre el navegador del cliente y el servidor, permitiendo enviar y recibir mensajes de forma simultánea (Full Dúplex) y manteniendo siempre una conexión activa con el servidor mediante Sockets TCP. Aunque originalmente fueron diseñados para ser implementados en navegadores y servidores web, puede utilizarse por cualquier aplicación cliente/servidor. El consorcio W3C es el encargado de la normalización del API de WebSocket.

Para abrir una conexión *WebSocket*, sólo solo es necesario instanciar un objeto WebSocket pasándole como parámetro la url en la que el servidor está esperando conexiones:

```
var websocket = new WebSocket('wss://url_donde_escucha_el_servidor');
```

El API proporciona dos eventos que se disparan cuando el socket se abre y está listo, y cuando se va a cerrar:

```
websocket.onopen  = function(e){ console.log("conexión preparada"; };
websocket.onclose = function(e){ console.log("conexión cerrada"};
```

Una vez que se dispone de una conexión activa con el servidor (cuando se dispara el evento ***onopen***), se puede empezar a enviar datos al servidor con el método ***send*** a través del socket creado del siguiente modo:

```
websocket.send("Este es un mensaje enviado a través de websocket");
```

Para recibir los mensajes que el servidor envía se utiliza el evento ***onmessage***, que se dispara al recibir un mensaje, del siguiente modo:

```
websocket.onmessage = function(event) {
        console.log(JSON.parse(event.data);
});
```

3.1.2. Java

Figura 2. Logo Java

Java es un lenguaje orientado a objetos que utiliza una sintaxis derivada y similar a la de C y C++ pero reducida. Fue creado en 1995 por la empresa Sun Microsystem. El objetivo de este lenguaje era que los programadores sólo tuvieran que escribir el código de un programa una vez, y que éste, pudiese ejecutarse en cualquier dispositivo. Esto es posible gracias a la **Máquina Virtual de Java** (JVM).

3.1.3. Vert.x

Figura 3. Logo Vert.x

Vert.x es un toolkit para desarrollar aplicaciones reactivas en la JVM. Es poliglota, con soporte para múltiples lenguajes de programación, como Java, Groovy, Python, JavaScript y otros. Se trata simplemente de componentes que pueden añadirse a una aplicación sin la necesidad de modificar la estructura de esta.

Vert.x está dirigido por eventos y es no bloqueante. Los eventos se gestionan mediante el Event Loop el cual se encarga de que cada uno de ellos llegue a su handler correspondiente.
Los elementos básicos sobre los que se apoya Vert.x son:

- **Verticles**: son clases cuyo comportamiento está orientado a enviar/recibir mensajes. Para facilitar el desarrollo, Vert.x asegura que el código de un verticle nunca va a ser ejecutado por más de un thread a la vez.
- **EventBus**: se trata de un bus transversal a la aplicación que permite la comunicación entre los verticles. Esta puede establecerse mediante un mecanismo de publish-suscribe o punto a punto.
- **Event-loop**: gestiona la ejecución de los handlers de forma que cada uno es ejecutado únicamente en un thread y este no debe ser bloqueante de ninguna manera.
- **WebSockets**: sobre los que se basa la comunicación entre cliente/usario y servidor en este proyecto.

3.1.4. Hazelcast

Figura 4. Logo Hazelcast

Hazelcast es un Data Grid en Java, es decir, una plataforma escalable para la distribución de datos. Vert.x hace uso de Hazelcast como gestor del clúster y provee la capacidad de distribuir los datos entre todos los nodos de la aplicación desplegados en nuestro clúster.

Entre sus características más interesantes, y de las que haremos uso en el proyecto se encuentran:

- Implementaciones distribuidas de Set,List,Map,Lock,MultiMap
- Persistencia síncrona o asíncrona
- Discovery dinámico

3.1.5. MongoDB

Figura 5. Logo MongoDB

MongoDB es un sistema de base de datos NoSQL Opensource, orientada a documentos.

En lugar de guardar los datos en tablas, tal y como se hace en las bases de datos relacionales, MongoDB guarda estructuras de datos BSON (una especificación similar a JSON) con un esquema dinámico, haciendo que la integración de los datos en ciertas aplicaciones sea más fácil y rápida. Los documentos de una misma colección (concepto similar a una tabla de una base de datos relacional) pueden tener esquemas diferentes.

En nuestro proyecto será fundamental, ya que se usará para persistir los mensajes enviados, y permitirá la recuperación de estos en caso de pérdida de conexión de alguno de los usuarios.

3.1.6. Javascript

Figura 6. Logo Javascript

JavaScript (JS) es un lenguaje de programación ligero, interpretado, o compilado just-in-time. El estándar para JavaScript es ECMAScript. A partir del 2012 todos los navegadores modernos soportan completamente ECMAScript 5.1.

3.1.7. Node

Figura 7. Logo Node

Node.js es un entorno en tiempo de ejecución multiplataforma, de código abierto, basado en el lenguaje de programación JavaScript, asíncrono, con E/S de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google.

Node.js permite la ejecución de programas desarrollados en JavaScript en un ámbito independiente del navegador, de hecho, en algunas ocasiones se habla de NodeJS como JavaScript en servidor.

3.1.8. Angular

Figura 8. Logo Angular

Angular es un framework opensource desarrollado por Google para facilitar la creación y programación de aplicaciones web de una sola página, las webs SPA (Single Page Application). Entre las ventajas de Angular cabe destacar:

- Mejora la experiencia de uso.
- Mejora la percepción del rendimiento de las aplicaciones.
- Permite una mayor reutilización y homogeneidad para futuras aplicaciones.

En nuestro proyecto hemos desarrollado una aplicación front de ejemplo en la que se utiliza la librería de mensajes cliente desarrollada en Javascript para demostrar el funcionamiento del Backend.

3.2. Herramientas

3.2.1. Control de versiones con Git y GitHub

Figura 9. Logos de Git y de GitHub

Git es un software de control de versiones de código. Git presenta una arquitectura distribuida. En lugar de tener un único espacio para todo el historial de versiones del software como sucede de manera habitual en los sistemas de control de versiones como CVS o Subversion, en Git la copia de trabajo del código de cada desarrollador es también un repositorio que puede albergar el historial completo de todos los cambios.

GitHub es una plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git antes mencionado.

El repositorio de nuestro proyecto está alojado en GitHub, en la siguiente url:

https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant-GroupChat

3.2.2. Docker

Figura 10. Logo de Docker

Docker es una tecnología que permite la creación y el uso de contenedores de Linux. Su filosofía es crear contenedores ligeros y portables para las aplicaciones software que puedan ejecutarse en cualquier máquina con Docker instalado, independientemente del sistema operativo que tenga por debajo, facilitando así también los despliegues.

Docker ofrece un modelo de implementación basado en imágenes. Esto permite compartir una aplicación o un conjunto de servicios con todas sus dependencias en varios entornos.

3.2.3. Repositorio de imágenes DockerHub

Figura 11. Logo de DockerHub

DockerHub es un repositorio público en la nube, para distribuir imágenes Docker de aplicaciones y otros contenidos y compartirlos con otros usuarios. Está mantenido por la propia Docker y hay multitud de imágenes de carácter gratuito que se pueden descargar.

En nuestro proyecto empaquetaremos y subiremos las imágenes de las aplicaciones Backend y también la imagen del ejemplo frontend Angular.

- Imagen de la aplicación Backend: https://hub.docker.com/repository/docker/mscarceller/eftgca-backvertx

- Imagen de la aplicación Frontend: https://hub.docker.com/repository/docker/mscarceller/eftgca-front

3.2.4. Registro de Paquetes NPM

Figura 12. Logo de NPM

NPM es una herramienta de gestión de paquetes que permite la instalación de todas las herramientas de las cuales la aplicación tiene dependencia.

NPM Registry permite almacenar, de forma similar a GitHub, imágenes de paquetes y librerías que pueden después añadirse como dependencia dentro de otros proyectos y que con un simple comando *npm install* se descargarán y añadirán a nuestros proyectos.

La url del registry de nuestra librería de mensajes es: https://www.npmjs.com/package/eftgca-messages

3.2.5. Kubernetes

Figura 12. Logo de Kubernetes

Kubernetes (referido en inglés comúnmente como “K8s”) es una tecnología creada por Google y sirve para la gestión y orquestación de contenedores Docker.

Kubernetes permite asilar al desarrollador de los problemas de infraestructura y centrarse en el desarrollo de las aplicaciones y en cómo empaquetarlas. Basado en este principio todos los conceptos que usa están relacionados con este propósito.

Alguno elementos clave de Kubernetes son:

- Pods: Conjunto de contenedores y volúmenes.
- Replication Controllers: Gestor de Pods que asegura que están levantadas las réplicas y permite escalar de forma fácil. Una réplica es una copia exacta de un Pod. Levanta Pods en caso de fallos o reinicios.
- Service: Define como acceder a un grupo de Pods.

Aparte de esto elementos Kubernetes tiene otros elementos: balanceadores de carga, servicios Ingress, etc.

3.2.6. VSCode

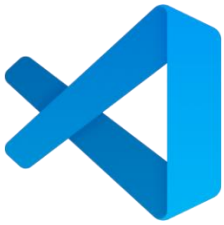

Figura 13. Logo de VSCode

Visual Studio Code es un editor de texto gratuito de Microsoft, con soporte para numerosos lenguajes de programación, tanto frontend como Backend. Además, incorpora un sistema de gestión de plugins o extensiones que facilitan la instalación de hacks para hacer más fácil el desarrollo. Es ampliamente utilizado en la comunidad Javascript, especialmente entre los desarrolladores Angular y en la comunidad Spring.

Posee un terminal integrado desde donde es fácil hacer operaciones en el sistema operativo. Además, tiene versiones compatibles con 32 y 64 bits, tanto para Linux, Mac o Windows.

3.2.7. Okteto Cloud

Figura 14. Logo de Okteto

Okteto es un proyecto de código abierto que ofrece una experiencia de desarrollo local para las aplicaciones que se ejecutan en Kubernetes. Provee de forma gratuita un clúster de Kubernetes en la nube para poder desplegar aplicaciones basadas en contenedores.

Los despliegues pueden realizarse mediante un cliente cli, o desde la web de Okteto si el código estuviese alojado en un repositorio de código.

3.2.8. Gremlin

Figura 15. Logo de Gremlin

Gremlin es una herramienta que permite realizar una serie de pruebas sobre sistemas, tales como carga de CPU sobre los sistemas, pruebas simulando fallos de red, chaos testing, etc.

Las pruebas se realizan desde la misma web, desplegando un cliente en el PC local que se comunica con la plataforma web y ejecuta los escenarios de pruebas configuradas en la misma.

Para más información visita la web de Gremlin: https://www.gremlin.com

4. Descripción de la solución

La solución que se propone está basada en dos elementos:

- Por un lado, el servidor de chat que recibirá los mensajes de cada usuario y será el encargado de almacenarlos y distribuirlos al resto de usuarios de la sala. Se trata de la aplicación de chat en sí misma.
- Para facilitar y estandarizar el uso y las comunicaciones entre las aplicaciones cliente expuestas a los usuarios y dicho servidor de chat se implementa, además, una librería de mensajes. Esta se encarga de la comunicación con el servidor y provee los métodos necesarios para enviar mensajes, tanto de texto como imágenes y ficheros al servidor. Además, emite una serie de eventos para que las aplicaciones cliente los capturen y puedan manejarlos como deseen.

4.1. Requisitos

A continuación, se enumeran los requisitos de la aplicación, tanto funcionales como no funcionales.

4.1.1. Requisitos funcionales

Los requisitos funcionales son aquellos que definen funciones del sistema:

- Grupos de chat con varios usuarios: se gestionan los grupos de chat como “salas” de conversación. Un mismo usuario puede acceder a varias salas de chat.
- Envío de mensajes de texto.
- Envío de mensajes con imágenes.
- Envío de mensajes con ficheros.

4.1.2. Requisitos no funcionales

Los requisitos no funcionales son aquellos que se enfocan en el diseño o la implementación:

- Elástica, la aplicación tiene que ser elástica, tanto a nivel de infraestructura como software, aumentar recursos cunado la carga de usuarios aumenta y liberarlo cuando disminuye.
- Tolerante a fallos.
- Actualización sin pérdida de servicio.
- Sin punto único de fallo.

4.1.3. Casos de uso

A continuación, se describen los casos de uso identificados:

HU1	Si la CPU del sistema alcanza un umbral de carga, el sistema deberá añadir más recursos de computación para atender a nuevos usuarios.
HU2	Si la CPU total del clúster está por debajo de un umbral, el sistema reagrupará a los clientes de forma que un nodo se pueda eliminar sin caída del servicio.
HU3	Si se produce una desconexión entre el cliente web y el servidor, por caída del WebSocket, los mensajes enviados deben encolarse y enviarse al recuperar la conexión.
HU4	Si se produce una desconexión entre el cliente web y el servidor, por caída del WebSocket, los mensajes pendientes de recibirse deben ser reclamados.
HU5	Los usuarios deben poder enviar mensajes que contengan: • Texto • Imágenes • Ficheros
HU6	Si se produce una actualización de la aplicación no debe producirse perdida de servicio.

Figura 26. Casos de uso

5. Diseño e Implementación

La aplicación ha sido desarrollada para ser desplegada en un clúster de Kubernetes. Como hemos mencionado anteriormente, el Backend está basado en Vert.x.

Vert.x se basa en el uso de un Bus de eventos para comunicar los diferentes elementos del Backend, y en Hazelcast para compartir de forma distribuida elementos como mapas de usuarios, conexiones de bases de datos, etc.

El esquema básico de la aplicación es el siguiente:

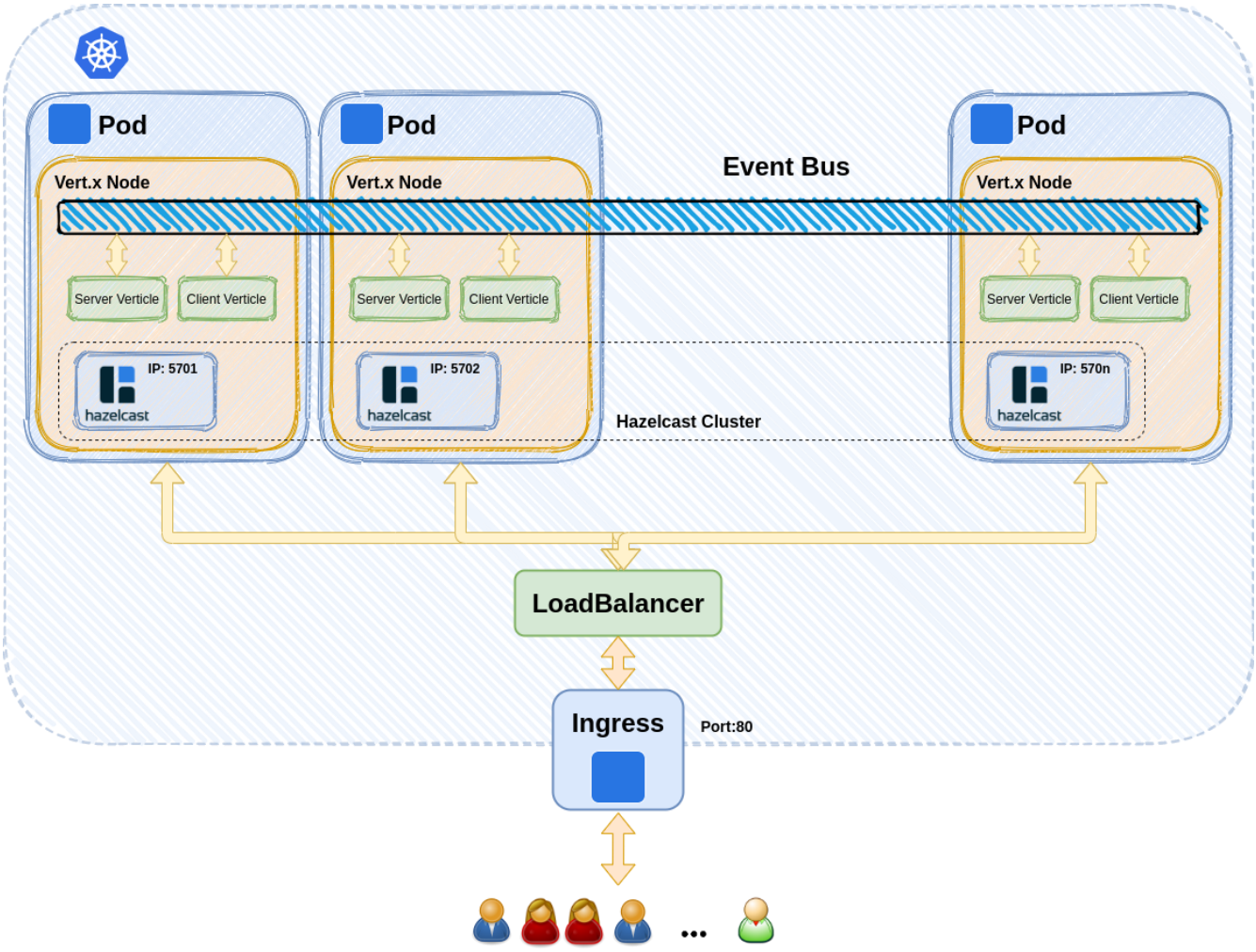

Figura 37. Esquema básico de la aplicación

Se dispone de un grupo de pods escalable horizontalmente que despliegan el backend de la aplicación. El punto de entrada al sistema es un balanceador de carga, que repartirá el tráfico entre los diferentes nodos, y que facilitará, además, la redirección de tráfico durante el escalado.

Los puntos básicos de la solución son:

- **Persistencia**: Todos los mensajes enviados al servidor son almacenados en una base de datos MongoDB.
  En el momento que un usuario accede a la sala de chat el servidor recupera los últimos mensajes y se los envía al usuario. Cuando se produce una reconexión, le envía solo aquellos mensajes que el usuario no ha recibido. Se profundiza más en este mecanismo en la sección de tolerancia a fallos.

- **Elasticidad**: Es la capacidad de un sistema de aumentar o liberar los recursos de una infraestructura de forma dinámica para adaptarse a las necesidades del sistema, para lograr optimizar el uso de recursos. Gracias a que el sistema se despliega sobre un clúster de Kubernetes, haciendo uso del ***Horizontal Pod Autoescaler*** (HPA), lograremos tener un sistema elástico.

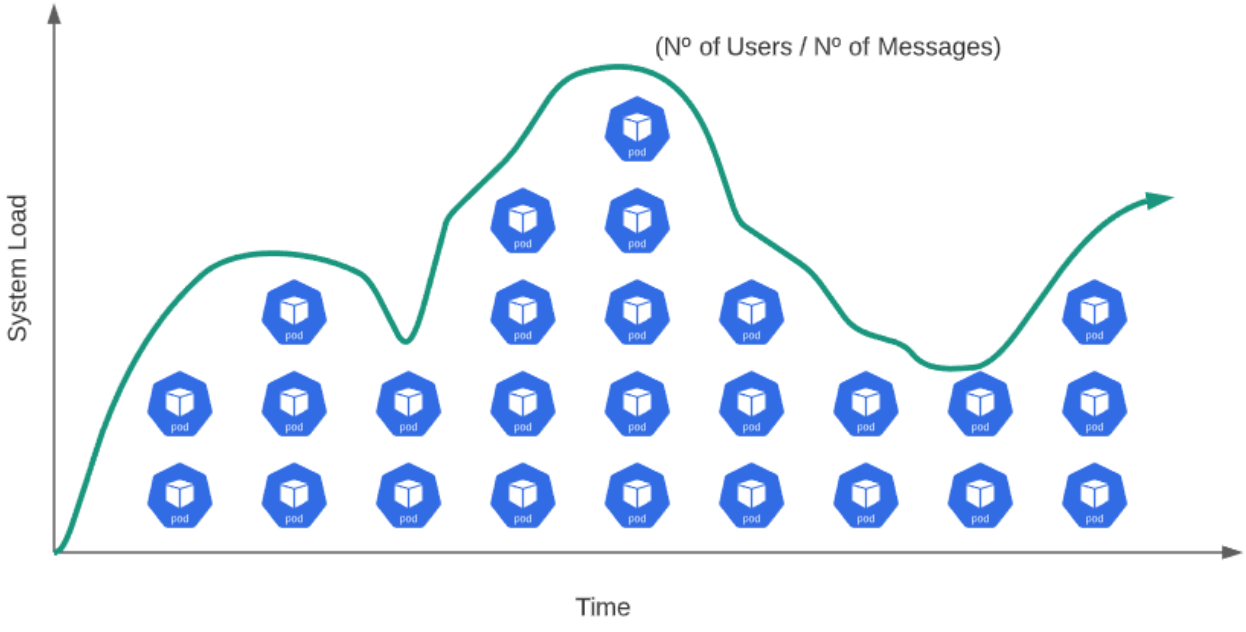

Figura 48. Elasticidad

Para más detalles sobre cómo se implementa la elasticidad y como funciona HPA puedes visitar el repositorio de desarrollo [1].

- **Tolerancia a fallos:** Es la propiedad de las aplicaciones para continuar su funcionamiento si uno o más de sus elementos fallan. Dado que nuestra solución está desplegada sobre un clúster de Kubernetes, en una arquitectura elástica, es muy importante permitir lo que se conoce como "graceful shutdown", la capacidad de desconectar algún nodo sin que haya perdida de servicio, permitiendo que los usuarios reciban dicho servicio desde otro de los nodos disponibles.

  Lo más importante en un sistema de chat es que los usuarios no pierdan ningún mensaje, en ninguno de los casos de fallo planteado en la sección de casos de uso:
  - En caso de fallo de Backend.
  - En caso de fallos de red.
  - En casos de eliminación de pods por reescalado.

  En todos estos casos, descritos con más detalle en la documentación técnica [2], es la librería de mensajes (ver documentación de desarrollo [3]) desarrollada la encargada de gestionar las reconexiones, recuperación de mensajes perdidos, etc.

  Esquema básico del despliegue en Kubernetes:

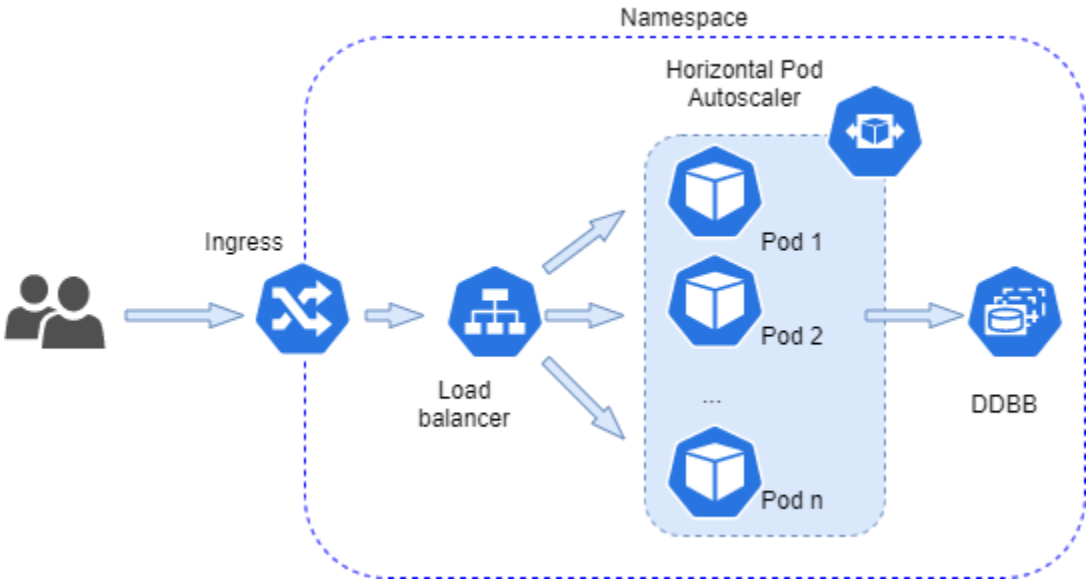

Figura 59. Esquema Kubernetes

[1] https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant GroupChat/blob/master/Documents/Development.md#backend
[2] https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant-GroupChat#gettingstarted
[3] https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant-GroupChat/blob/master/Documents/Development.md

Se puede ver el detalle de todos los elementos en el repositorio "***Kubernetes architecture elements"*** [4]

5.1. Backend basado en Vert.x

La estructura básica de la aplicación Backend es la que se muestra en la siguiente figura:

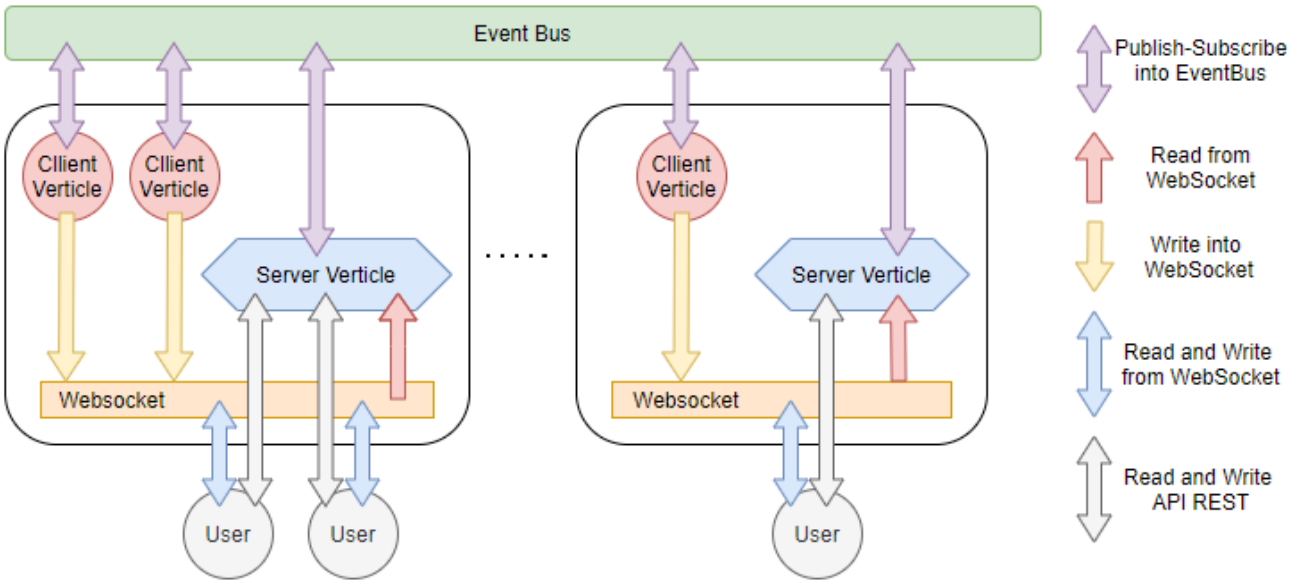

Figura 20. Esquema Vertx

Cada nodo tiene una instancia de un Verticle Server. Este verticle escucha conexiones entrantes de usuarios a través de un Websocket y expone además una serie de endpoints API Rest.

La comunicación es bidireccional entre los verticles, tanto clients como servers, y el Event Bus de Vert.x, tal y como indicamos anteriormente, ya que siguen un patrón publish/subscribe:

- El Server Verticle recibe los mensajes de los usuarios a través del Websocket y los publica en el bus de eventos, indicando usuario y sala a la que pertenecen.
- El Client Verticle lee del bus de eventos los mensajes que corresponden a su sala y los envía a través del Websocket a su usuario.

[4] https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant-GroupChat/blob/master/EFTGCA-VertxBackend/README.md#k8s

Cuando un usuario envía una solicitud de registro en una sala, el servidor valida el identificador del usuario (nombre en la sala) e instancia un Client verticle nuevo que se suscribe al bus de eventos de Vert.x.

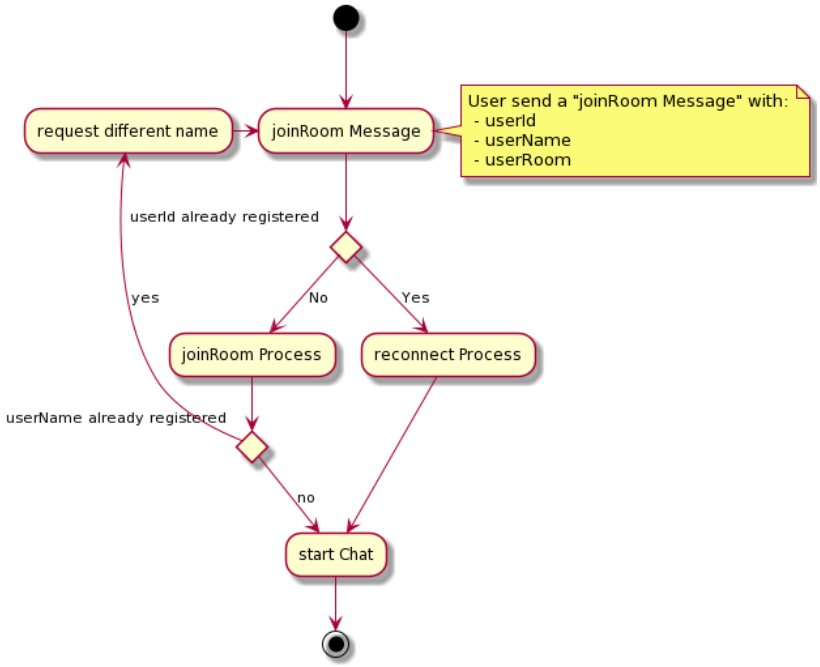

Figura 21. Registro en una sala

Visita el repositorio "***Elastic & FaultTolerant GroupChat Application - Vert.x based Application"*** [5] para más información.

[5] https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant-GroupChat/blob/master/EFTGCA-VertxBackend/README.md

5.2. Librería cliente JavaScript

Para acompañar al servidor se ha implementado una librería auxiliar de mensajes en Node. Dicha librería expone una serie de métodos y dispara una serie de eventos que permiten, desde cualquier aplicación cliente, comunicarse con el servidor (enviar y recibir mensajes, atender eventos, tales como reconexiones o mensajes entrantes, etc.)

Los mensajes siguen la especificación JSON-RPC 2.0.

Visita el repositorio "***Elastic & FaultTolerant GroupChat Application Messages Lib"*** [6] para más información.

5.3. Frontend basado en Angular

Para demostrar la aplicación práctica de la librería de mensajes y cómo se puede utilizar desde otras aplicaciones se ha implementado un proyecto de ejemplo en Angular, que presenta una sala de chat.

La aplicación propone una pantalla en la que se puede introducir texto, y dispone de dos botones para anexar imágenes o ficheros. Además de una barra de estado como la que se ve en la siguiente imagen en la que se puede ver el nombre de la sala actual, en del usuario y chequear el estado de la conexión con el servidor en función del color del icono que aparece a la derecha:

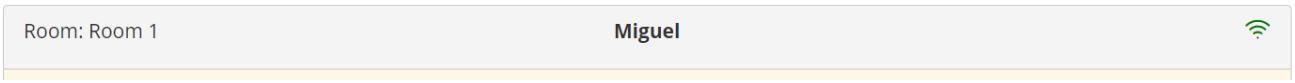

Figura 22. Barra de estado del chat con conexión OK (en verde)

[6] https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant-GroupChat/blob/master/EFTGCA-MessagesLib/README.md

En las siguientes imágenes se puede ver como un usuario accede a la sala, y puede enviar y recibir mensajes de otros usuarios, tanto de texto como imágenes o ficheros:

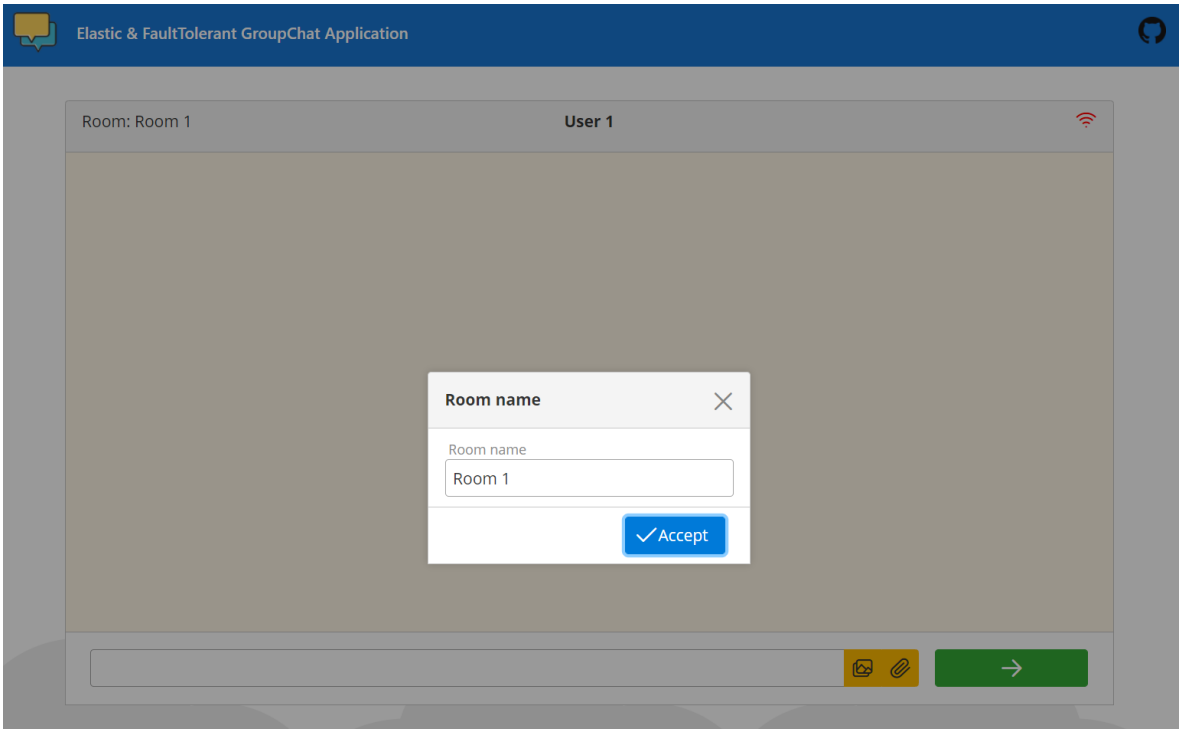

Figura 23. Selección de Sala

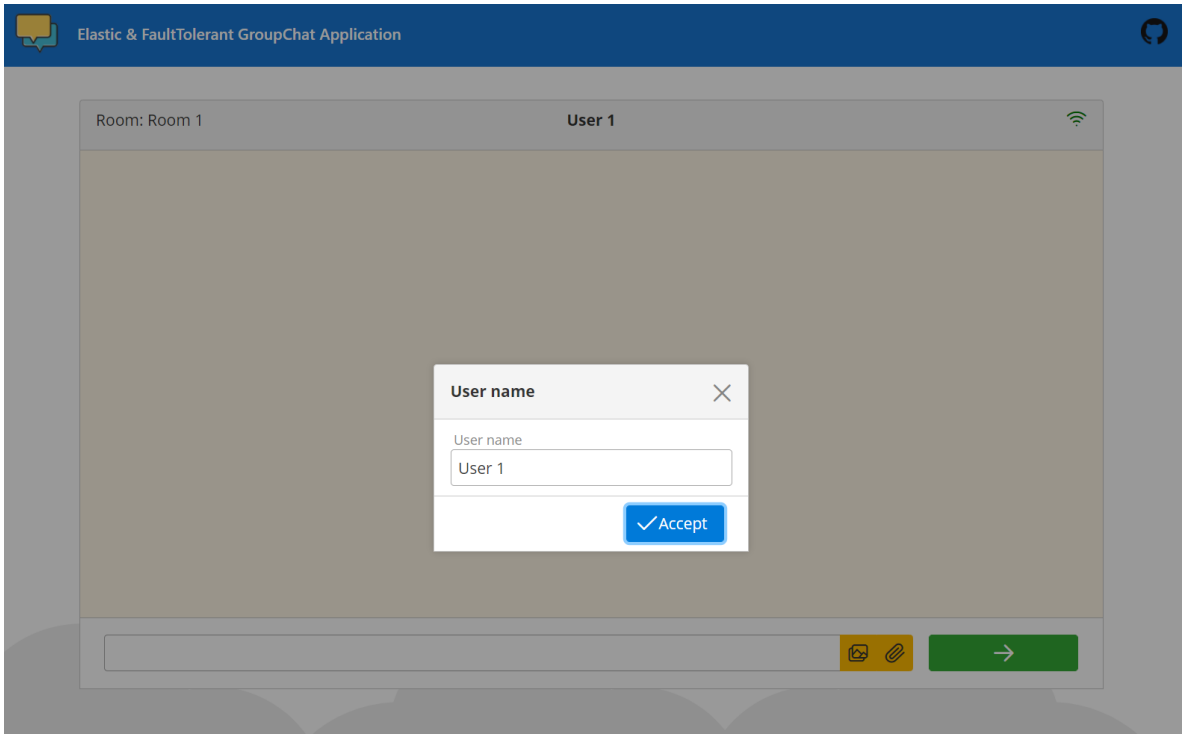

Figura 24. Selección de Nombre de Usuario

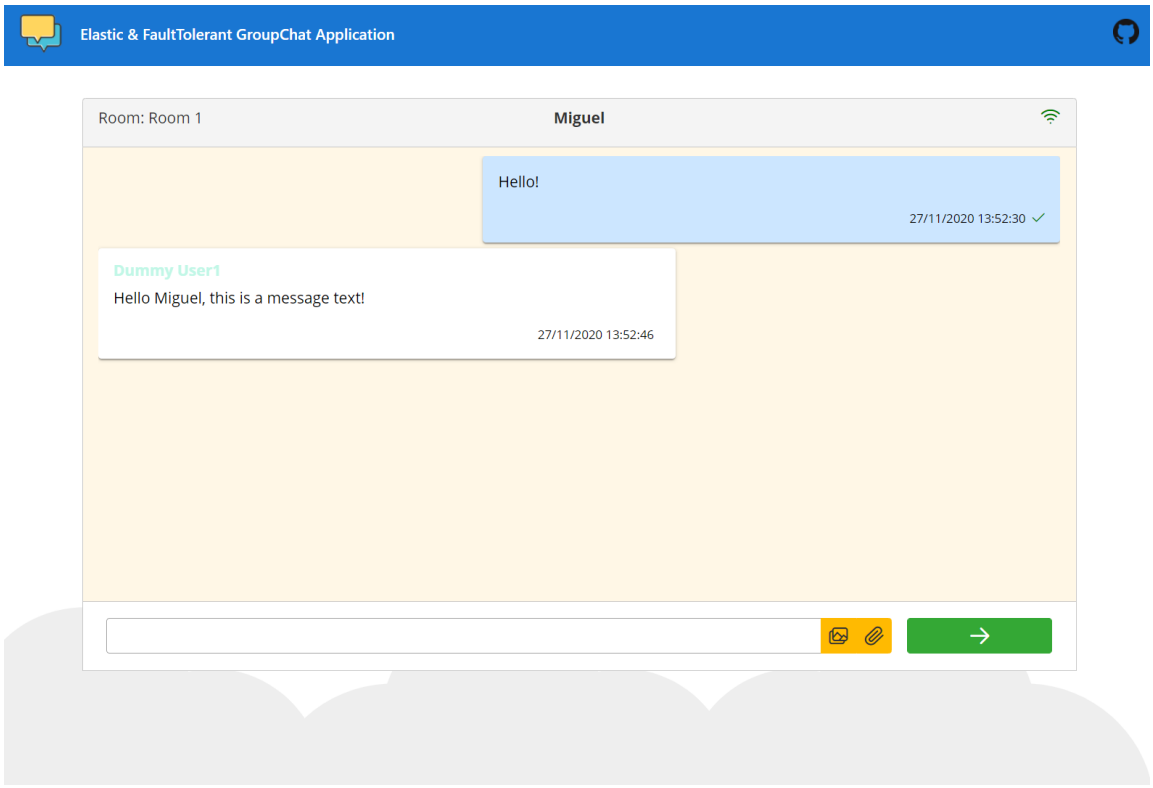

Figura 25. Intercambio de mensajes de texto

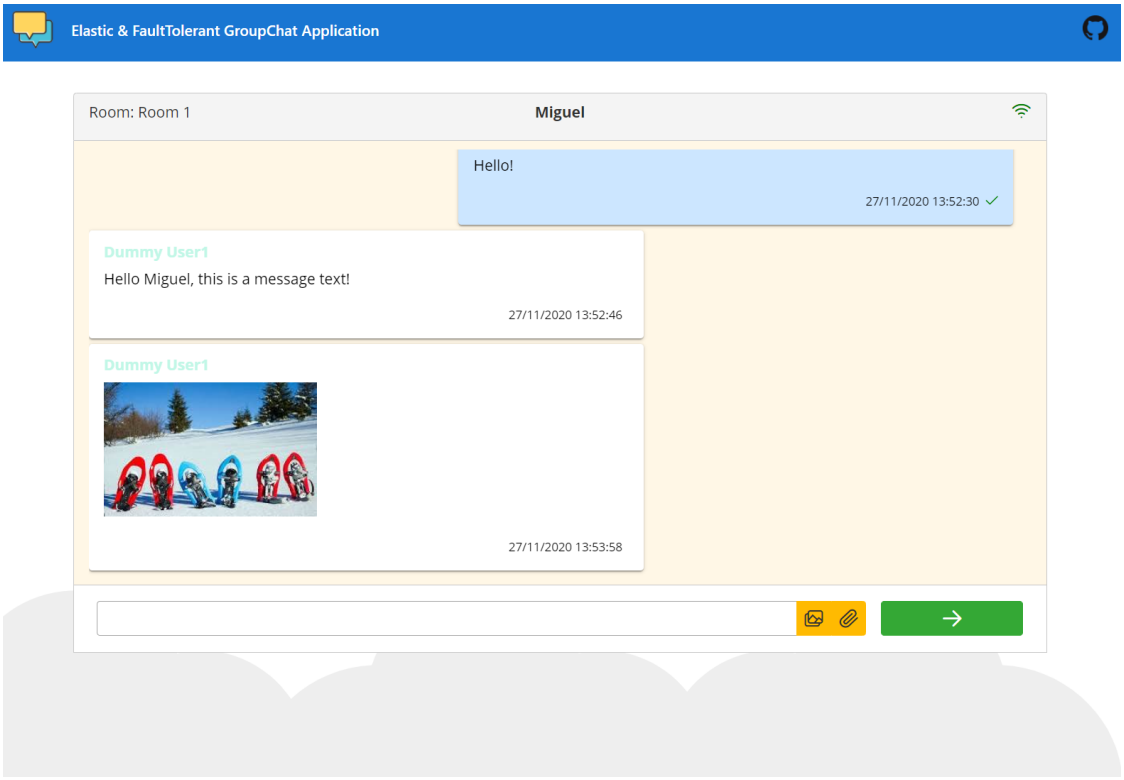

Figura 26. Como se recibe una imagen

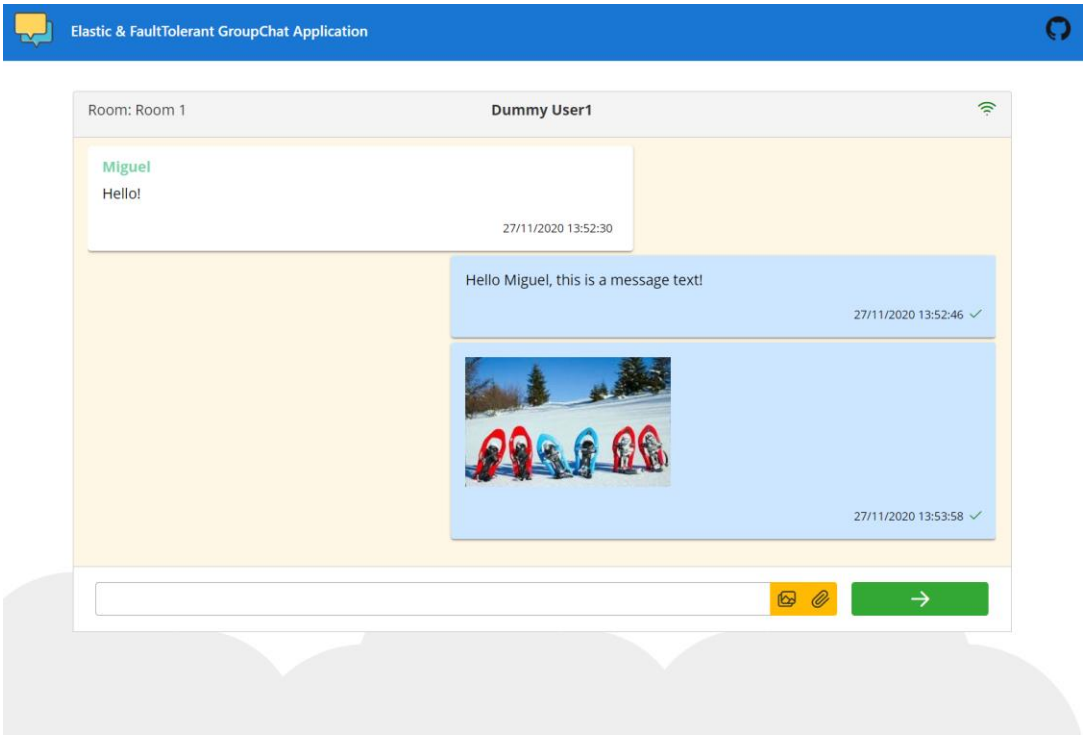

Figura 27. Como se envía una imagen

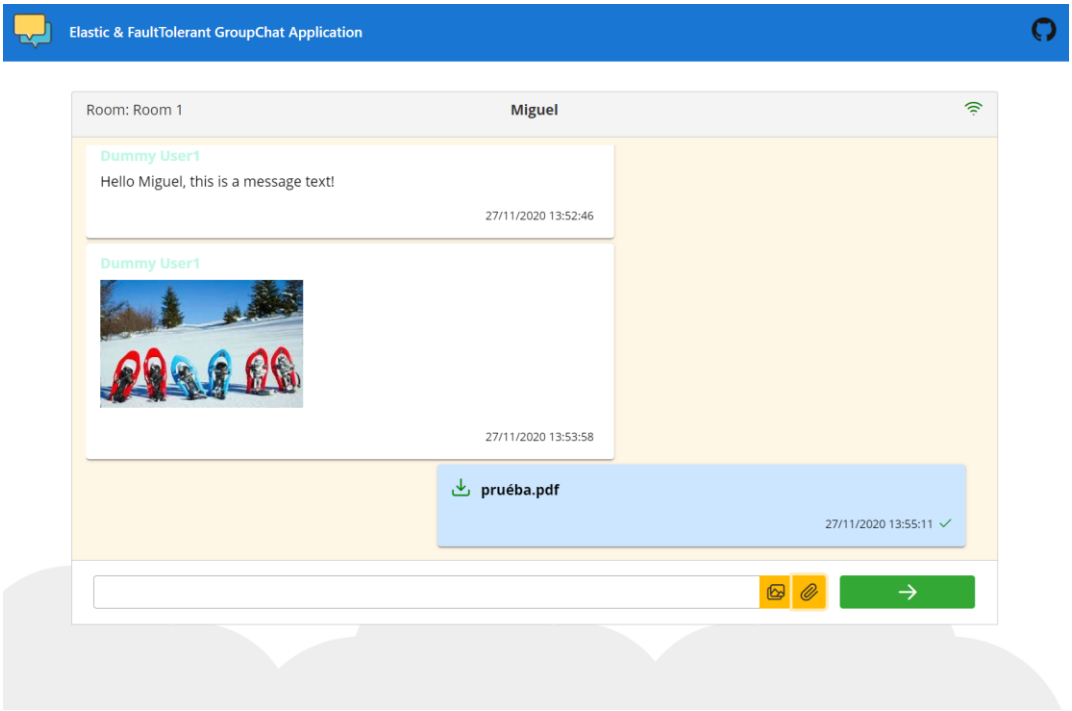

Figura 28. Como se envía un fichero

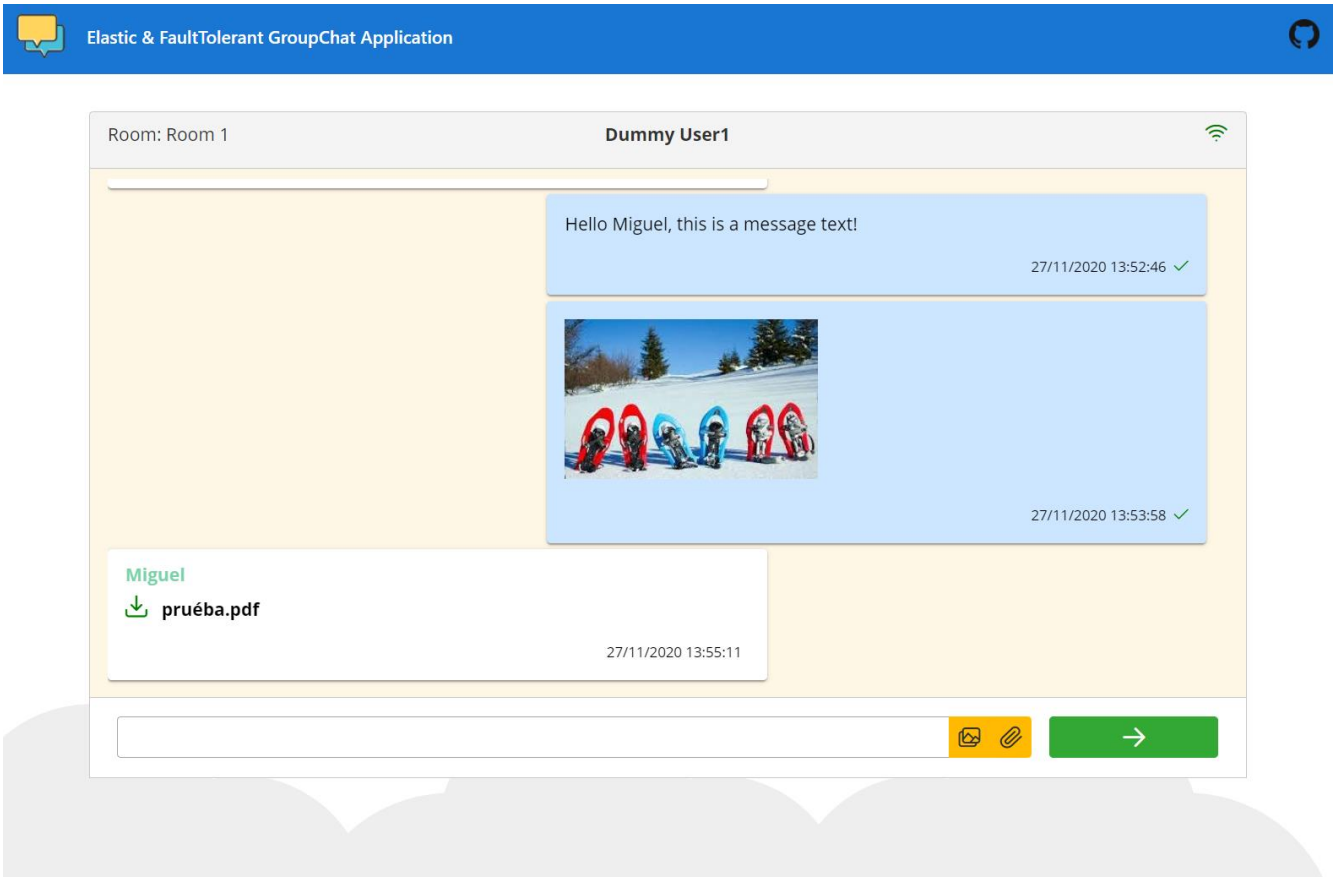

Figura 29. Como se recibe un fichero para descargar

Para la importación de la librería en el proyecto solo hay que añadirla a las dependencias del mismo en el fichero *package.json*

```
"dependencies": {
  "eftgca-messages": "^3.0.0",
   …
```

Para importarla en el módulo en que se desee usar:

```
import * as ChatMessagesManager from "eftgca-messages"
```

Visita el repositorio "***Angular Frontend example for Elastic & FaultTolerant GroupChat Application***" [7] para más información.

[7] https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant-GroupChat/blob/master/EFTGCA-Front/README.md

6. Funcionamiento

El funcionamiento básico de la aplicación lo aporta la combinación entre la librería de mensajes y el servidor de chat.

La librería encapsula todos los procedimientos de envío y recepción de mensajes y se encarga también de la conexión inicial al WebSocket del servidor y de las reconexiones en caso de fallo, caída del pod, escalado, etc.

Los usuarios pueden enviar mensajes haciendo uso de la librería diseñada para tal fin. Los mensajes, dependiendo del tipo se envían a través de API Rest, en el caso de imágenes y ficheros, o a través de WebSocket. Los diferentes casos se describen a continuación:

Cuando un usuario envía un nuevo mensaje de texto, la librería lo envía a través del WebSocket. El servidor lo publica en el bus de eventos, al que están suscritos los clientes, con información de la sala a la que pertenece. Los clientes extraen el mensaje y lo transmiten a los usuarios a través del Websocket. Cuando el mensaje se recibe, la librería emite el evento correspondiente para que la aplicación cliente pueda manejar el evento como desee.

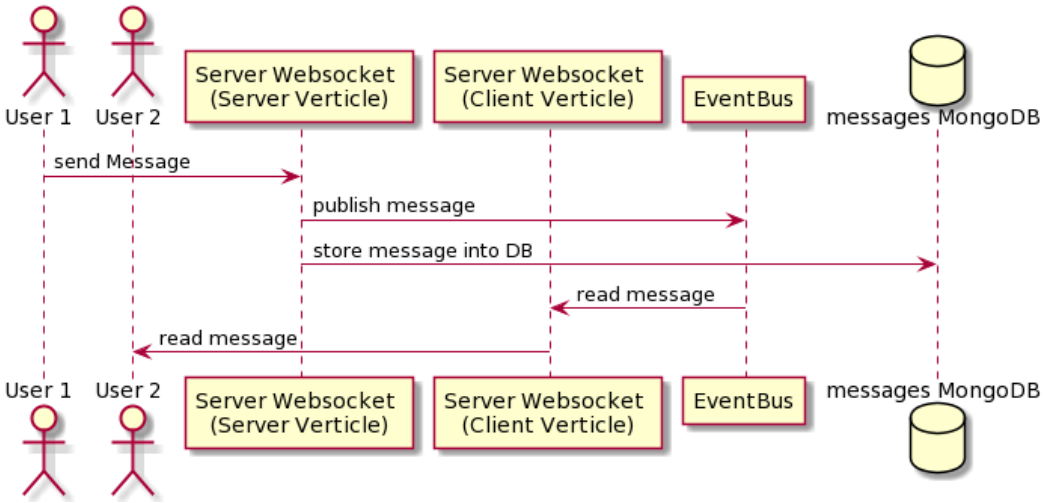

Figura 30. Flujo de envío de mensaje de texto

Cuando un usuario envía una imagen o fichero lo hace a través de API Rest, mediante una operación POST, para evitar saturar el Websocket, que tiene limitaciones en cuanto al tamaño de los mensajes.

Para el usuario este hecho es transparente, ya que la librería expone métodos que recubren el sistema de envío. Cuando el servidor recibe una imagen o fichero los almacena como si se tratasen de mensajes normales (en el caso de imágenes su descripción en base64, y en el caso de fichero el path en que se ha guardado). El servidor envía una notificación al resto de usuarios informando de que hay un mensaje con imagen o fichero y estos la descargan a través del API Rest con una operación GET. En el caso de las imágenes se hace de forma automática. En el caso de los ficheros se proporciona un enlace de descarga. A nivel de usuario final esto queda recubierto por la librería, que solo expondrá un método para enviar imágenes o ficheros y emitirá un evento de nueva imagen o fichero cuando se haya completado el flujo.

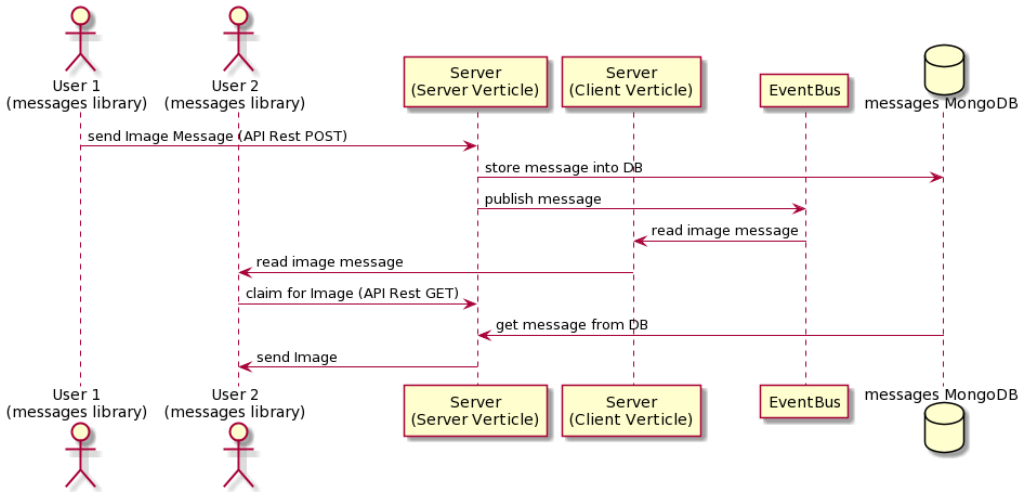

Figura 31. Flujo de envío de imágenes

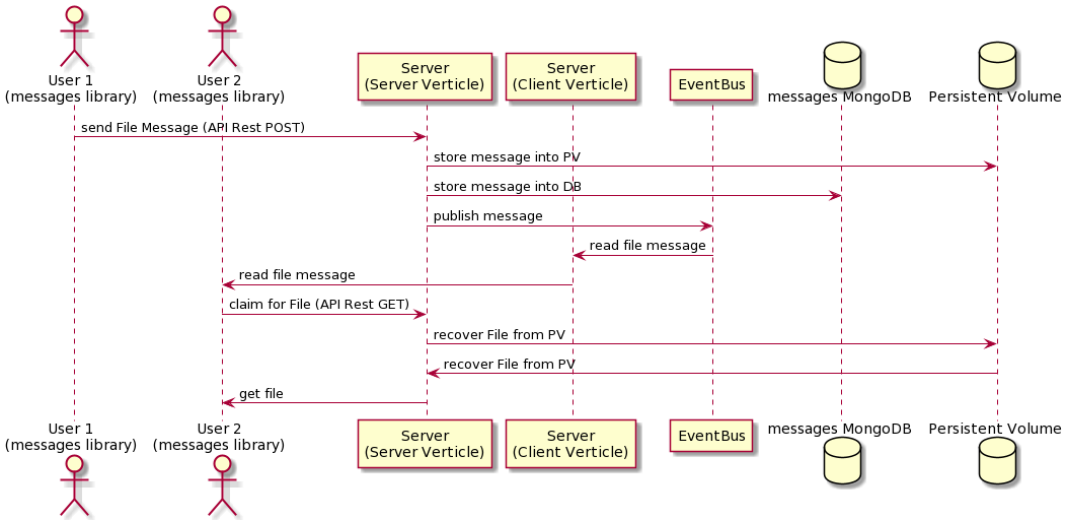

Figura 32. Flujo de envío de ficheros

7. Pruebas

En este capítulo se describen las pruebas realizadas sobre la aplicación, que he orientado en dos direcciones. Pruebas de elasticidad para analizar la escalabilidad del sistema sin pérdida de servicio, y pruebas de caos testing para verificar la tolerancia a fallos. En ambos casos se comprueba que no hay perdidas de mensajes.

7.1. Elasticidad

Como hemos mencionado anteriormente, la elasticidad horizontal es la capacidad de aprovisionar y liberar dinámicamente instancias de cómputo, en nuestro caso nodos del clúster Kubernetes.
Las pruebas de elasticidad se implementan de dos formas diferentes:

- Mediante un script que usa la librería de mensajes para crear una serie de usuarios y realizar envíos de mensajes de forma masiva para generar carga sobre el sistema. Y posteriormente validar que no se ha perdido ninguno de ellos.

- Mediante **Gremlin** [8], un servicio cloud que permite realizar una variedad de pruebas sobre infraestructuras Kubernetes, y que en nuestro caso usaremos para simular carga de CPU sobre los nodos del clúster local.

Visita el repositorio "***EFTGCA-VertxAppTests***" [9] para más información sobre las ejecuciones y procedimientos, y también para los resultados de los test.

[8] https://www.gremlin.com
[9] https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant-GroupChat/tree/master/EFTGCA-VertxAppTests

7.2. Tolerancia a fallos

Dado que ningún sistema o plataforma en producción es infalible a caídas, es necesario someter a nuestra aplicación a una serie de pruebas que nos permitan asegurar que no hay perdida de servicio (en nuestro caso perdida de mensajes) aunque se produzca la caída de alguno de los nodos del clúster. Para realizar estas pruebas sometemos a los sistemas a test de caos, que consisten en "matar" de forma aleatoria pods del clúster sobre los que se está ejecutando la aplicación, y comprobar como esta resiste a dichos cambios.

Mediante un script que usa la librería de mensajes se crean una serie de usuarios y se realizan envíos de mensajes, para posteriormente validar que no se ha perdido ninguno.
Se realizan dos tipos de prueba, en función del tipo de despliegue, en local y cloud.

- **Despliegue en local:** se implementa la prueba mediante un pod desplegado en el clúster, que se encarga de matar de forma periódica y aleatoria alguno de los pods sobre los que se está ejecutando la aplicación.

- **Despliegue en el cloud de Okteto**: las pruebas se realizan mediante el despliegue de un pod de Litmus, que se facilita por parte de Okteto. Toda la información sobre como desplegar y realizar las pruebas está en el documento: "***Chaos Testing with Litmus and Okteto Cloud***"[10] del repositorio del proyecto.

Visita el repositorio "***EFTGCA-VertxAppTests*****"** [11] para más información sobre las ejecuciones y procedimientos, y también para los resultados de los test.

[10] https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant-GroupChat/blob/master/EFTGCA VertxAppTests/ChaosTestingOkteto.md
[11] https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant-GroupChat/tree/master/EFTGCA-VertxAppTests

8. Conclusiones

Se pone de manifiesto que la mayor dificultad en el desarrollo de este tipo de sistemas escalables es la gestión de los datos y de la información entre los diferentes nodos que componen el sistema. En este caso nos hemos apoyado en el toolkit Vert.x, que proporciona todas las herramientas necesarias para lograr nuestro objetivo, tales como el bus de eventos para compartir los mensajes, el clúster Hazelcast para compartir la información entre los diferentes nodos, o el acceso a una base de datos de forma concurrente desde todos los nodos.

Cada día surgen nuevos toolkits y frameworks similares que facilitan estas tareas, dado el auge de los microservicios y la containerización de las aplicaciones.

A nivel personal, mediante el desarrollo del proyecto he podido profundizar en aspectos relacionados con el desarrollo de aplicaciones distribuidas, y otros relacionados con el desarrollo de software en general, poniendo en práctica muchos de los conceptos teóricos vistos durante el curso, tales como:

- Desarrollo de sistemas distribuidos, en este caso utilizando con Vert.x.
- Desarrollo con Java, Node, etc.
- Bases de datos NoSQL como MongoDB.
- Protocolos avanzados basados en bus de eventos o Websockets, API Rest, etc.
- Empaquetado y publicación de aplicaciones en DockerHub y NPM.
- CI/CD con GitHubActions.
- Computación en la nube, en este caso con Oketo.
- Gestión del código fuente, aplicación en Github.
- Contenedores y Orquestadores.

Ha sido muy interesante poder profundizar en estos temas mediante la realización de una aplicación práctica, a la que aún le queda mucho recorrido, pero que considero que puede ser un buen punto de partida, quizá no como aplicación “usable”, pero si a nivel docente pudiendo revisar cada uno de los aspectos tratados y su aplicación.

9.Trabajos futuros

Como futuros trabajos o puntos en los que se podría profundizar un poco más y que se pueden desarrollar sobre la aplicación, yo propondría:

- Implementación con otro tipo de tecnología para sistemas distribuidos (Spring + Hazelcast…) y comparativa de rendimiento.
- Se podrían implementar otra serie de características, como borrado de los mensajes, edición de los mensajes…
- **Securizar** la aplicación mediante el uso de tokens de seguridad, certificados, etc.
- **Observabilidad**: La mejora del sistema de logs de la aplicación. Profundizar sobre temas de Observabilidad y aplicarlos para "monitorizar" en todo momento su comportamiento, comprender el uso que le puedan dar los usuarios y anticiparse a posibles fallos y casos de uso.
- **Mejora de CI/CD.** Algún tipo de despliegue Blue/Green o Canary release, como aplicación práctica de lo visto en el máster.
- **Seguridad y Gestión de usuarios**: Se podría desarrollar un sistema para la gestión de los usuarios y flujos de autenticación OAuth2, también como aplicación de lo visto durante el máster. Implementación de usuarios y accesos con Amazon Cognito por ejemplo.
- Despliegue en diferentes **proveedores cloud y comparativa de rendimientos**.

10. Bibliografía

- Web del W3C con la especificación de la API:
  - http://dev.w3.org/html5/websockets
- RFC 6455 Protocolo Websocket:
  - http://tools.ietf.org/html/rfc6455
- Documentación Vert.x:
  - https://vertx.io/docs/
- Documentación Hazelcast:
  - https://hazelcast.com/resources/?topic=getting-started-and-deployment
- JSON-RPC 2.0 Specification:
  - https://www.jsonrpc.org/specification
- Wikipedia MongoDB:
  - https://es.wikipedia.org/wiki/MongoDB
- MongoDB Manual:
  - https://docs.mongodb.com/manual/
- JavaScritpt MDN:
  - https://developer.mozilla.org/es/docs/Web/JavaScript
- Node API docs:
  - https://nodejs.org/docs/latest-v12.x/api/
- Angular Docs:
  - https://angular.io/docs
- Rolling updates con Kubernetes:
  - https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/
- Chaos monkey:
  - https://github.com/Netflix/chaosmonkey
- Chaos testing con Okteto:
  - https://okteto.com/blog/chaos-engineering-with-litmus/
- Litmus caos:
  - https://hub.litmuschaos.io/
- Gremlin:
  - https://www.gremlin.com/docs/quick-starts/get-started-with-gremlin-free/

11. Anexos

Como anexo a la memoria se indican los enlaces a los diferentes repositorios que componen la solución al proyecto:

- Repositorio principal del proyecto:

  https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant-GroupChat

- Guías de instalación del proyecto:
  - Despliegue en un clúster local Kubernetes:

    https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant-GroupChat/blob/master/Documents/GettingStarted.md#kubernetes
  - Despliegue en Okteto Cloud:

    https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant-GroupChat/blob/master/Documents/GettingStarted.md#okteto
  - Despliegue de proyecto de ejemplo Angular:

    https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant-GroupChat/blob/master/Documents/GettingStarted.md#front

- Guías de desarrollo de cada parte de la solución:
  - Backend Basado de Vert.x:

    https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant-GroupChat/blob/master/Documents/Development.md#backend
  - Librería de mensajes:

    https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant-GroupChat/blob/master/Documents/Development.md#messagelib
  - Chaos Testing con Litmus y Okteto Cloud:

    https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant-GroupChat/blob/master/Documents/ChaosTestingOkteto.md
  - Scripts de testing de la solución:

    https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant-GroupChat/tree/master/EFTGCA-VertxAppTests
  - Aplicación de ejemplo desarrollada con Ángular:

    https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant-GroupChat/blob/master/Documents/Development.md#front
  - CI/CD con GitHub Actions:

    https://github.com/MasterCloudApps-Projects/ElasticFaultTolerant-GroupChat/blob/master/Documents/Development.md#actions