



Universidad
Zaragoza

Tecnologías y Modelos para el Desarrollo de Aplicaciones Distribuidas

Documentación aplicación chat

Laura Oliva Maza - 702756

Índice

1. Introducción	1
2. Historias y requisitos	1
3. Descripción del sistema	3
4. Arquitectura del sistema	7
4.1. Modelo de datos	7
4.1.1. Modelo mensajes	7
4.1.2. Modelos bases de datos	8
4.1.3. Modelo de RabbitMQ	10
4.2. Vista de componentes y conectores	13
4.2.1. Tier 0: Cliente	14
4.2.2. Tier 1: Servidor	15
4.2.3. Tier 2: Datos	24
4.2.4. Microservicio	24
4.3. Vista de distribución	26
4.3.1. PC Cliente	27
4.3.2. Servidor	27
4.3.3. Servidor Censura	28

4.3.4. CloudAMQP	28
4.4. Comportamiento del sistema	29
5. Evaluación del sistema	33
5.1. Número de usuarios	33
5.2. Tiempo de procesado de mensajes	35
5.3. Limitaciones	37
6. Requisitos cumplidos	40
6.1. Requisito 1	40
6.2. Requisito 2	43
6.3. Requisito 3	46
6.4. Requisito 4	51
6.5. Requisito 5	52
7. Conclusiones	55

1. Introducción

El objetivo principal de este proyecto es desarrollar una aplicación distribuida de tipo chat. Para ello se ha desarrollado una aplicación en Java, utilizando Gradle y Spring Boot. Esta aplicación debe cumplir con unos requisitos e implementar unas historias que se explicarán en la siguiente sección.

En este informe se puede encontrar información sobre la arquitectura del sistema desarrollado, junto con los mensajes que se intercambian entre las diferentes componentes y el tipo de comunicación que se utiliza.

Por último, se han realizado unas pruebas sobre el sistema para comprobar sus limitaciones y calcular el tiempo que se invierte en procesar y enviar los mensajes.

2. Historias y requisitos

La aplicación de chat que se ha desarrollado debe implementar las siguientes historias de usuario:

1. Como usuario quiero poder enviar mensajes a otros usuarios para tener conversaciones puntuales.
 - Cada mensaje es texto plano y puede tener hasta 500 caracteres.
 - Mensajes “punto a punto” simplemente sabiendo el nombre de los usuarios a los que van dirigidos.
 - Si un usuario no está online cuando se le envía un mensaje, se le notifica automáticamente cuando se conecte.
 - Poder intercambiar mensajes simultáneamente con al menos 5 usuarios (tanto si están como si no están conectados al mismo tiempo).
 - Los mensajes llegan a todos los participantes en “push” (p.ej. sin tener que refrescar la aplicación).

- Ningún mensaje tarda más de 60 s en llegar a todos los destinatarios (siempre que estén online en el momento en que el mensaje se envió).
2. Como usuaria quiero poder compartir ficheros de mi equipo con otros usuarios para complementar las conversaciones puntuales.
 - Los ficheros intercambiados pueden ser de cualquier tipo y pesar hasta 1 MB.
 - Por lo demás, y para todo lo que tenga sentido, los mensajes de tipo fichero enviado tienen los mismos criterios de aceptación que los mensajes básicos de texto (historia 1).
 3. Como usuario quiero poder crear salas de chat permanentes para tener un registro de las conversaciones pasadas
 - El creador de la sala tiene permiso de administración sobre la misma: es el único que puede invitar/desinvitar a usuarios y borrar la sala cuando quiera.
 - Solo los usuarios invitados por el creador de la sala pueden acceder.
 - Salas identificadas y accesibles con una URL única.
 - Todos los mensajes enviados a través de la sala quedan registrados en el sistema hasta que se decida borrar esta sala.
 - En un chat permanente se pueden intercambiar los mismos tipos de mensajes que de manera puntual (p.ej. Historias 1 y 2).
 4. Como superusuario quier opoder enviar mensajes a todos los usuarios suscritos para anuncios de interés y publicidad de nuevas características del sistema
 - Solo mensajes de texto
 5. Como superusuaria quiero poder censurar algunos mensajes para cumplir con los requisitos legales exigidos en regímenes represivos con las libertades civiles pero con mercados económicamente rentables.

- Poder establecer que ciertas palabras o frases no pueden usarse y hacer que cualquier mensaje que las contenga llegue pero con estas palabras o frases censuradas.
- Que cada infracción quede registrada para, eventualmente, poder expulsar a los infractores que repitan mucho.
- Estas palabras y frases deben poder modificarse en cualquier momento. Por ejemplo es posible que a mitad de una conversación una palabra que estaba permitida deje de estarlo y los mensajes sucesivos que la contengan deben llegar censurados.

Además, este sistema debe soportar correctamente a varios usuarios simultáneos y se ha de desplegar y ejecutar en al menos tres máquinas virtuales de las cuales al menos una debe estar en un proveedor cloud.

En las siguientes secciones se va a explicar cómo se ha implementado el sistema, junto con las decisiones arquitecturales tomadas, para que se cumplan los requisitos mencionados.

3. Descripción del sistema

El sistema desarrollado se encuentra en el siguiente repositorio de GitHub:

https://github.com/LauraOliva/TMDAD_ProyectoMensajes

Para cumplir los requisitos explicados se ha desarrollado una aplicación web en la que los usuarios pueden introducir su nombre de usuario (Figura 1). Una vez el sistema haya verificado si es un usuario existente o un usuario nuevo y si se puede crear un nuevo usuario, se pasa a una nueva pantalla. En esta nueva pantalla el usuario puede introducir comandos (ventana de comandos) o mandar mensajes a su chat activo (ventana de mensajes) que, inicialmente, es nulo (Figura 2).

El usuario interactúa con la aplicación enviando mensajes a su sala activa o comandos. El usuario puede introducir los siguientes comandos:

- **HELP**: muestra por pantalla todos los comandos que el usuario puede ejecutar

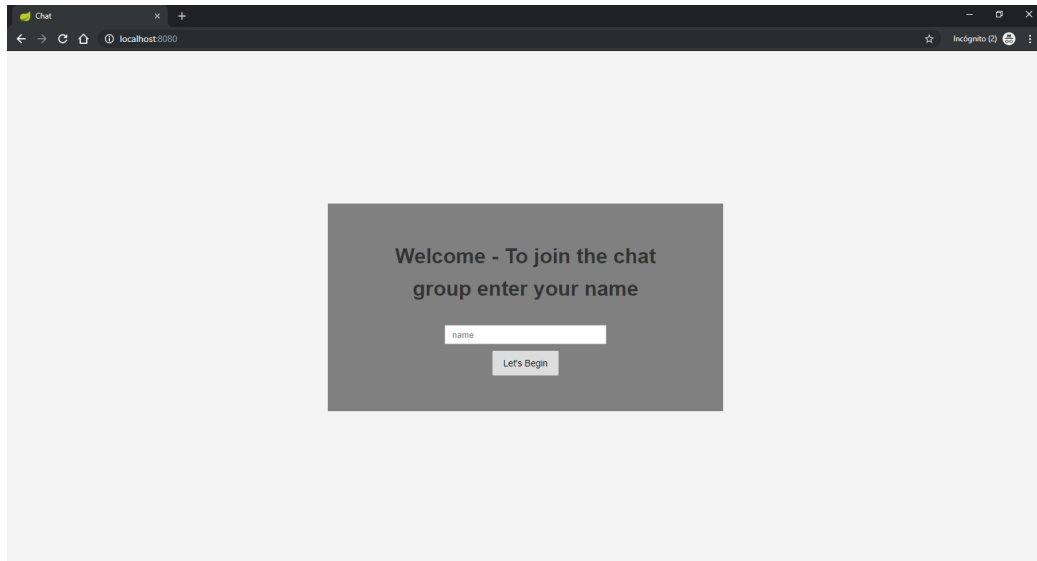


Figura 1: Pantalla inicio

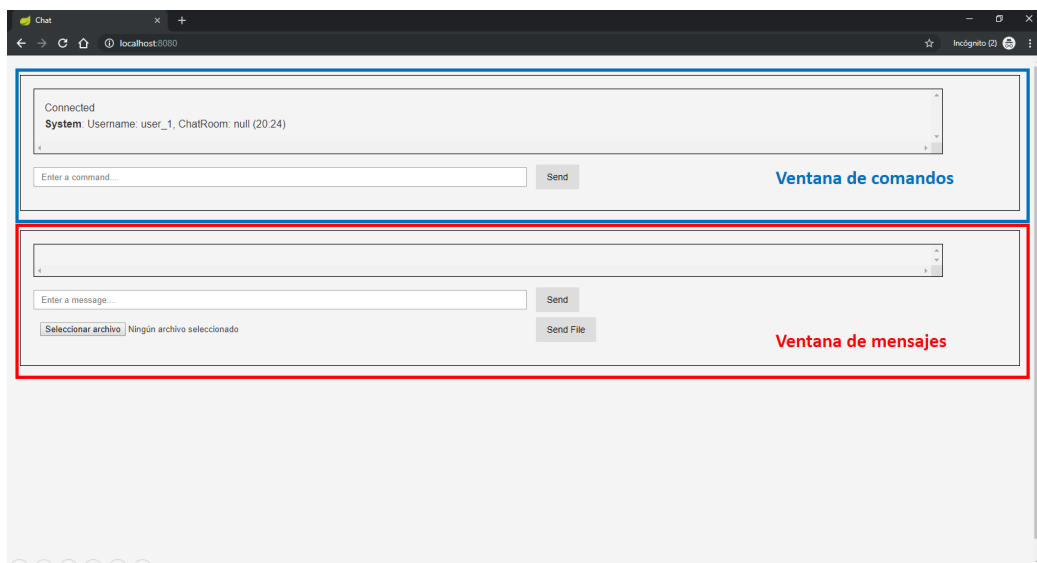


Figura 2: Pantalla principal

- **CREATEROOM** *id_room*: crea una sala con el nombre *id_sala*, solo si no existe ya una sala con ese nombre, cuyo administrador y único miembro es el usuario que ha ejecutado este comando.
- **INVITEROOM** *id_room id_user*: este comando solo puede ser ejecutado por el administrados de la sala *id_room*. El usuario *id_user* recibirá

una notificación con las indicaciones para unirse a la sala *id_room*.

- **JOINROOM** *id_room*: sólo los usuarios que hayan sido invitados a la sala *id_room* pueden ejecutar este comando. Una vez que el usuario ejecuta este comando se une a la sala *id_room*.
- **LEAVEROOM** *id_room*: sólo los usuarios que pertenezcan a la sala *id_room* pueden ejecutar este comando. Con este comando el usuario abandona la sala *id_room* y deja de recibir mensajes y notificaciones de esa sala.
- **GETROOMS**: devuelve todas las salas en las que esta el usuario que lo ha ejecutado.
- **GETUSERSROOM** *id_room*: sólo los usuarios que pertenezcan a la sala *id_room* pueden ejecutar este comando. Devuelve los usuarios que están en la sala *id_room*.
- **CLOSEROOM**: el usuario deja de recibir mensajes de su sala activa. Ahora recibirá notificaciones del estilo: “Nuevo mensaje en la sala x”
- **OPENROOM** *id_room*: sólo los usuarios que pertenezcan a la sala *id_room* pueden ejecutar este comando. La sala *id_room* pasa a ser la sala activa del usuario que ejecuta este comando. Ahora el usuario recibirá los mensajes de esta sala en la ventana de mensajes.
- **DELETEROOM** *id_room*: sólo el administrador de la sala *id_room* puede ejecutar este comando. Una vez ejecutado, se expulsa a todos los usuarios de esa sala, se eliminan los mensajes de la base de datos y se elimina la sala.
- **KICKROOM** *id_room id_user*: sólo el administrador de la sala *id_room* puede ejecutar este comando. Con este comando el administrador de la sala *id_room* expulsa al usuario *id_user* de la sala. El usuario *id_user* dejará de recibir mensajes de la sala.
- **CHATUSER** *id_user*: este comando inicia o abre una conversación entre dos usuarios a la que no se puede unir nadie. Este comando crea una sala privada de dos usuarios.

- **BROADCAST** *msg*: sólo el administrador del sistema puede utilizar este comando. Con este comando el administrador del sistema envía el mensaje *msg* a todos los usuarios del sistema.
- **ADDCENSURE** *word*: sólo el administrador del sistema puede utilizar este comando. Con este comando el administrador añade la palabra *word* a la lista de palabras censuradas (base de datos). Desde el momento que el administrador añade esta palabra, si un usuario intenta enviar esta palabra, se añadirá una incidencia a la base de datos y se censurará la palabra en el mensaje por “*****”.
- **REMOVECENSURE** *word*: sólo el administrador del sistema puede utilizar este comando. Con este comando el administrador elimina la palabra *word* de la lista de palabras censuradas (base de datos). Desde el momento en el que el administrador ejecuta este comando, los usuarios pueden volver a enviar mensajes que contengan la palabra *word* sin que sean censurados.
- **GETCENSURE**: sólo el administrador del sistema puede utilizar este comando. Con este comando el administrador obtiene la lista de palabras censuradas.

Para poder enviar mensajes o ficheros el usuario debe iniciar una conversación con otro usuario, crear una sala, unirse a una sala de la que tenga invitación o abrir una sala a la que ya se haya unido previamente. Una vez el usuario tenga una sala activa, este recibirá los mensajes de esa sala en la ventana de mensajes. Si un usuario se une a una sala, pero decide cerrarla, dejará de recibir los mensajes de esa sala en la ventana de mensajes y empezará a recibir notificaciones cada vez que se envíe un mensaje a esa sala. Además, aunque el usuario tenga una sala activa, éste recibirá notificaciones de las otras salas a las que pertenece.

Con estos comandos y con la ventana de mensajes, este sistema cubre todas las historias de usuario explicadas en la sección anterior. En la siguiente sección se va a explicar cómo se ha diseñado este sistema para que el usuario pueda realizar las acciones comentadas. En la sección 5 se hará una evaluación de este sistema y en la sección 6 se explicará cómo la aplicación implementada cubre las historias de usuario y los requisitos explicados en la sección 2.

4. Arquitectura del sistema

Para mostrar la arquitectura del sistema en este apartado se van a mostrar diferentes diagramas:

- Modelos de las bases de datos
- Modelo de mensajes
- Modelo de Rabbit
- Diagrama de componentes y conectores
- Diagrama de despliegue
- Diagrama de secuencia

Con estos diagramas se pretende explicar el funcionamiento y la estructura del sistema y justificar las decisiones arquitecturales tomadas.

4.1. Modelo de datos

Lo primero de todo se va a hablar del modelo de los mensajes que se envían desde el cliente y de la estructura de los datos que se almacenan.

4.1.1. Modelo mensajes

El cliente y el servidor se comunican con mensajes de tipo json. Estos mensajes tienen dos campos:

- *type*: define lo que se va a hacer con el mensajes
- *content*: contenido del mensaje. Dependiendo del tipo este contenido se tratará de una forma u otra.

El mensaje puede ser de cinco tipos diferentes que, dependiendo de si los recibe el cliente o el servidor, tendrán una función u otra. Empecemos con los mensajes que se envía del cliente al servidor:

- **CHAT**: envia el mensaje a la centralita de la sala a la que va dirigido (sala activa del usuario que envia en mensaje) e inserta el mensaje en la base de datos.
- **VERIFY**: este tipo de mensajes son enviados del cliente al servidor. El contenido de los mensajes de este tipo es el nombre de usuario. Si el servidor recibe este tipo de mensajes comprueba si el usuario cuyo nombre de usuario se encuentra en el contenido del mensaje, ha utilizado previamente el sistema y, si no lo ha utilizado, comprueba que se pueda crear un usuario nuevo y lo inserta en la base de datos. Además con este mensaje se añade la cola del usuario a la centralita del administrador del sistema.
- **COMMAND**: este tipo de mensajes son enviados del cliente al servidor. El contenido de este mensaje será uno de los comandos explicados en la sección anterior.

El servidor recibirá un mensaje de uno de los tipos mencionados y contestará al cliente (a través de rabbit y websockets) con un mensaje de uno de los siguientes tipos:

- **CHAT**: significa que este usuario ha recibido un nuevo mensaje en su sala activa por lo que se añade un nuevo mensaje a la ventana de mensajes.
- **NOTIFICATION**: este tipo de mensajes son enviados del servidor al cliente. El contenido de este mensaje será mostrado en la ventana de notificaciones.
- **CLEAN**: este tipo de mensajes son enviados del servidor al cliente. Se utiliza para limpiar la ventana de mensajes.

4.1.2. Modelos bases de datos

Para almacenar los datos del sistema se han creado tres esquemas de bases de datos MySQL diferentes:

- Base de datos chat: en esta base de datos se almacena la información de los usuarios, las salas y los mensajes enviados por los usuarios a las salas.
- Base de datos ficheros: en esta base de datos se almacenan la información de los ficheros.
- Base de datos censura: en esta base de datos se almacenan las palabras y los mensajes censurados.

En la base de datos chat, hay tres tablas:

- **Usuario:** en esta tabla se almacena la información del usuario, como su nombre de usuario (*username*), su sala activa (*activeroom*) y un booleano para indicar si es el administrador del sistema o no (*root*). Un usuario puede pertenecer a muchas salas, pero sólo tendrá una sala activa. En la ventana de mensajes solo se mostrarán los mensajes de su sala activa. Si recibe un mensajes de una sala a la que pertenece pero que no es su sala activa, recibirá una notificación del estilo: *Nuevo mensaje en sala x*.
- **Chatroom:** en esta tabla se almacena la información de las salas como su nombre (*name*), el nombre del usuario que ha creado la sala y, por lo tanto, es el administrador (*admin*) y un booleano para indicar si es una sala que permite varios usuarios o no (*multiple_users*).
- **Mensajes:** en esta tabla se almacena la información de los mensajes como un identificador que aumenta automáticamente (*id_mensajes*), el nombre de la sala a la que va dirigido ese mensaje (*dst*), el nombre del usuario que envía ese mensaje (*sender*), el contenido del mensaje (*msg*), un timestamp que indica cuando se ha enviado ese mensaje (*timestamp*) y el tipo de mensaje (*type*). Solo se almacenan los mensajes de tipo CHAT ya que de esta forma se garantiza persistencia. Además, estos mensajes se utilizan luego para comprobar si un usuario pertenece a una sala o si ha sido invitado a una sala.

En la base de datos de censura podemos encontrar dos tablas:

- **Palabras:** en esta tabla se almacenan las palabras que se va a censurar si aparecen en un mensaje. En esta tabla hay tres columnas: id, palabra y timestamp.
- **Mensaje:** en esta tabla se almacenan los mensajes censurados porque contenían una de las palabras almacenadas en la tabla *Palabras*. Para tener un seguimiento de los usuarios que utilizan palabras censuradas, en esta tabla se almacena: id, el mensaje sin censurar (*msg*), una lista de las palabras censuradas que contiene ese mensaje (*palabras*), un timestamp y el nombre del usuario que ha mandado ese mensaje.

El administrador del sistema es el único que puede ver o realizar cambios sobre el contenido de la tabla de palabras, puesto que es el único que puede ejecutar los comandos ADDCENSURE, REMOVECENSURE y GETCENSURE.

Por último, la base de datos de ficheros consta de una única tabla (**files**) para almacenar la información de los ficheros enviados a cualquier sala y por cualquier usuario. En esta tabla se guarda: un identificador único de fichero (*id*), un array de bytes (*data*), el nombre del fichero (*file_name*) y el tipo del fichero (*file_type*)

4.1.3. Modelo de RabbitMQ

Para enviar mensajes a salas o a usuarios, notificaciones o mensajes broadcast se utiliza RabbitMQ. RabbitMQ es un sistema de colas, en el que una cola puede subscribirse a una centralita y recibir los mensajes que lleguen a ella. Una centralita recibe un mensaje y lo envía a todas las colas que estén enlazadas o suscritas a ella.

Teniendo en cuenta esto, se va a tener en cuenta varios casos:

- Enviar notificación a un usuario
- Enviar mensaje a una sala
- Enviar mensaje broadcast

Para satisfacer el primer caso se crea una cola por cada usuario. De esta forma si se quiere enviar una notificación a un usuario, simplemente habrá que mandársela a su cola.

Para satisfacer el segundo caso se crea una centralita por cada sala. De esta forma cada vez que un usuario se une a la sala con el comando JOIN-ROOM se enlazarla la cola de ese usuario a la centralita de la sala.

Para satisfacer el tercer caso se crea una centralita a la que las colas de todos los usuarios del sistema estarán enlazadas. De esta forma si el administrador envía un mensaje utilizando el comando BROADCAST, este mensaje se mandará a la centralita y la centralita se encargará de enviárselo a todos los usuarios del sistema. Esta centralita se llamará *rootExchange*.

Para que el sistema funcione correctamente, si un usuario está conectado, cada vez que llega un mensaje, se consume y se envía mediante web sockets a la sesión correspondiente del usuario. Si el usuario no está conectado, los mensajes se irán acumulando en la cola y se enviarán al usuario una vez se conecte.

Se va a suponer la siguiente situación. En el sistema hay 4 usuarios: *user_1*, *user_2*, *user_3* y *user_4*. Cada usuario tiene una cola y todas las colas están unidas a la centralita *exchangeRoot*. En este caso, *user_2*, *user_3* y *user_4* están en la sala *room_1*, por lo que hay una centralita llamada *room_1* y las colas de dichos usuarios están unidas a ella. Por último, se va a suponer que los usuarios *user_1* y *user_2* han comenzado una conversación privada por lo que se ha creado una sala privada y, por tanto, la centralita *user_1-user_2* para esa conversación. En la **figura 3** se puede observar las centralitas que se han creado y cómo estarían unidas o suscritas las colas.

Ahora se va a suponer el caso en el que el usuario *user_1* quiere enviar un mensaje a la sala privada creada con *user_2*. En este caso *user_1* publicará un mensaje en la centralita *user_1-user_2* y la centralita enviará el mensaje a las colas suscritas o unidas a ella, es decir, a *user_1* y a *user_2*. En la figura 4 se puede observar como sería el proceso o el comportamiento de *rabbit* en este caso. Como se puede observar el mensaje sólo lo reciben las colas o los

usuarios que están suscritos a esa centralita, el resto de usuarios no reciben nada.

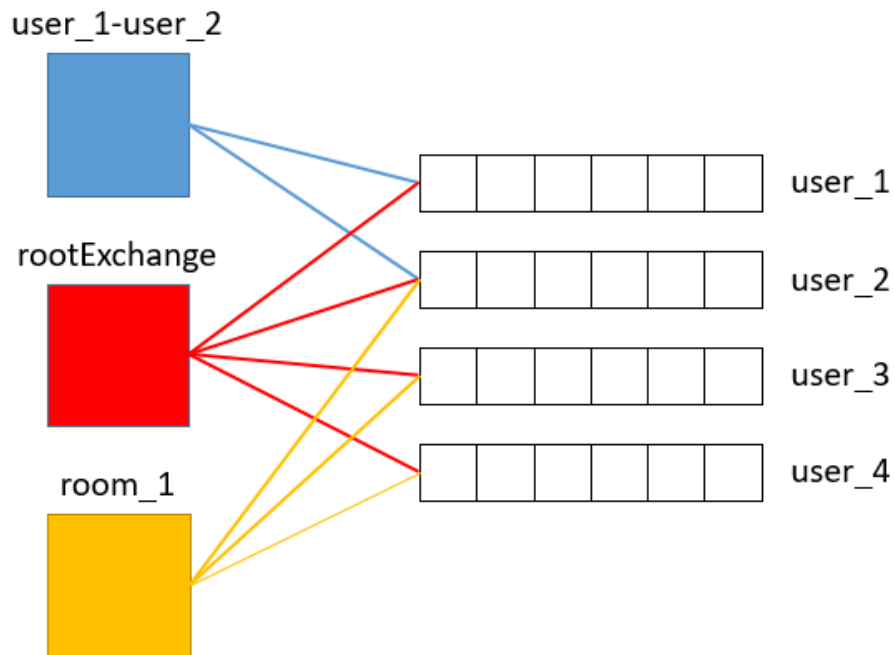


Figura 3: Estructura rabbitMQ

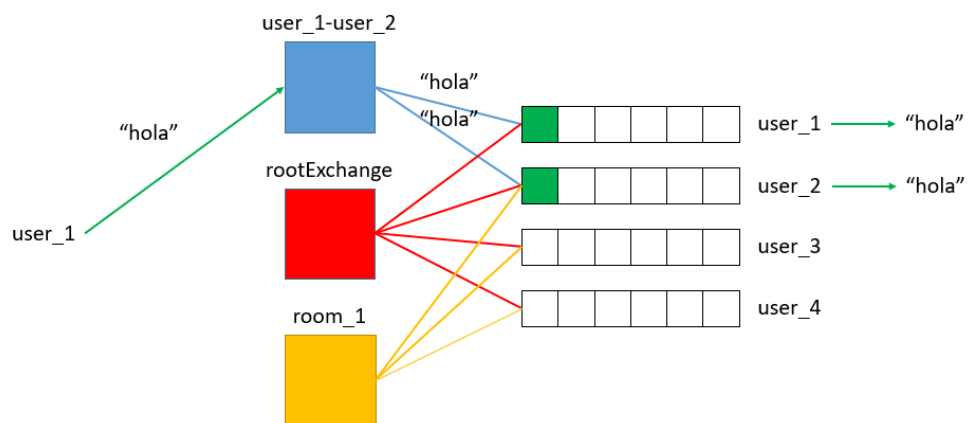


Figura 4: Ejemplo uso rabbitMQ

Las limitaciones que supone utilizar *rabbit* al igual que sus ventajas se comentarán en la sección 5.3. Después de las pruebas de esfuerzo realizadas para evaluar el sistema.

4.2. Vista de componentes y conectores

En esta subsección se van a explicar las diferentes componentes que forman el sistema y los mensajes y tipo de comunicación que utilizan para comunicarse entre ellas. En la figura 5 se puede observar el diagrama de componentes. A continuación, se va a explicar con detalle este diagrama y cada una de las distintas componentes de este sistema explicando para que se utiliza, quién la utiliza, cómo se comunican y que mensajes intercambian.

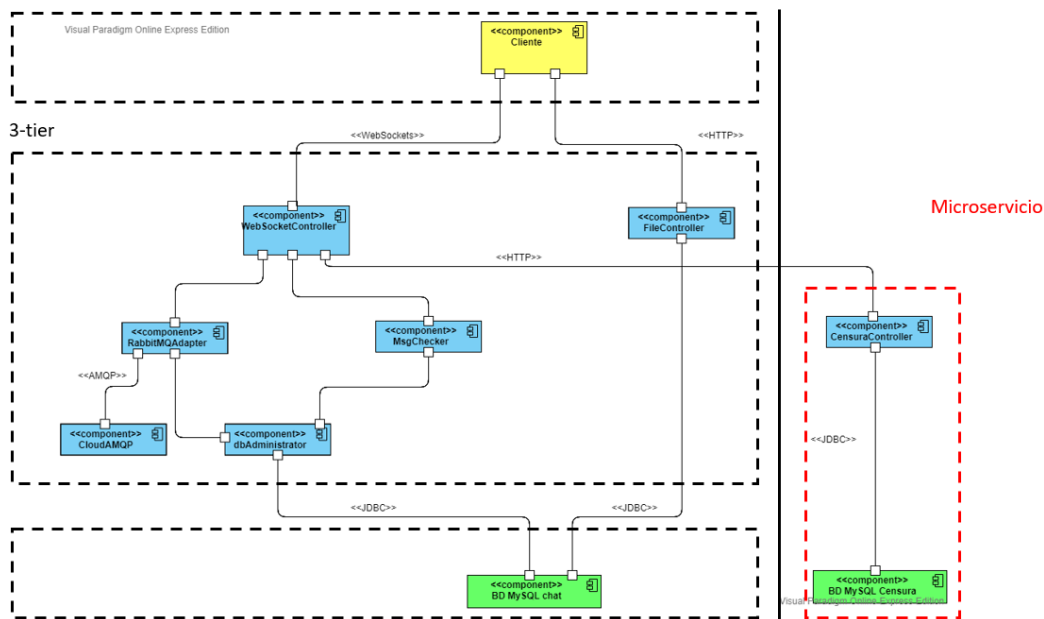


Figura 5: Diagrama de componentes

Como se puede observar en la figura 5 el sistema tiene una estructura **3-tier** y un **microservicio**.

Empecemos comentando la estructura 3-tier. Aquí tenemos 3 niveles o tiers diferentes:

- **Tier 0:** en este nivel se encuentra el componente cliente que se comunicará con dos componentes del nivel 1, el componente *WebSocketController* y *FileController*.
- **Tier 1:** en este nivel se encuentran todos los componentes del servidor. Este componente se comunica con el nivel 0 (cliente) y con el nivel 2 (datos). Además también se comunica con el microservicio. Esta componente se encarga de realizar toda la lógica del dominio.
- **Tier 2:** en este nivel se encuentra la base de datos. En esta base de datos se encuentran los esquemas de chat y ficheros. Esta componente recibe peticiones (*queries*) de las componentes *WebSocketController* y *FileController* del nivel 1.

Este sistema se comunica con un microservicio de censura mediante llamadas HTTP. Este servicio tiene una componente que recibe estas llamadas y otra componentes que es la base de datos en la que se almacenara toda la información sobre la censura.

Otra posibilidad para la estructura de este sistema era una arquitectura 4-tier en la que el nivel 0 fuera el cliente, el nivel 1 el servidor, el nivel 2 el servidor de censura y el nivel 3 las bases de datos. Puesto que en el nivel 2 sólo se encontraría el servidor de censura y puesto que componentes del nivel 1 se comunicarían con los del nivel 3, se descartó esta posibilidad.

También se podría haber definido como una arquitectura 3-tier en la que se juntaran los niveles 1 y 2 de la arquitectura 4-tier comentada. Esta arquitectura se descartó puesto que el servidor de censura se encuentra en otra máquina, por lo que la comunicación con esta componete no es de clase a clase si no que es HTTP por lo que se ha de encontrar en otro nivel.

4.2.1. Tier 0: Cliente

En este tier se encuentra el cliente. Esta componete se compone por el html y los javascript que va a ser ejecutados por el navegador del cliente.

Este nivel se comunica con el nivel 1 mediante web sockets y mediante HTTP.

En la comunicación de WebSockets se intercambian mensajes que siguen el modelo explicado en la sección 4.1.1.

En cuanto a la comunicación HTTP, el cliente puede realizar dos tipos de peticiones:

- **POST /uploadFile:**
 - Parámetros: file: fichero que se desea almacenar.
 - Descripción: el fichero *file* se almacena en la base de datos de ficheros
 - Devuelve: la URL necesaria para descargar el fichero almacenado.
- **GET /downloadFile/{fileId}:**
 - Parámetros: fileId: identificador del fichero que se desea descargar. Este parámetro está incluido en la ruta.
 - Descripción: el fichero con identificador *fileId* se recupera de la base de datos.
 - Devuelve: el fichero recuperado.

En el supuesto caso en el que el cliente quiera enviar un fichero, primero se realizara la petición HTTP /uploadFile con dicho fichero y luego se enviará un mensaje mediante WebSockets con la URL recibida.

4.2.2. Tier 1: Servidor

En este tier se encuentran todas las componentes que realizan las funciones necesarias para que los mensajes y ficheros del chat se almacenen en la base de datos correspondientes y para que lleguen los mensajes a los usuarios correctos o para que si el usuario ejecuta un comando, obtenga el comportamiento deseado.

Para entender este tier se va a explicar cada una de sus componentes por separado y los mensajes que se intercambian entre las componentes de este tier.

Vamos a empezar por el *WebSocketController*. Esta componente recibe los mensajes del cliente, que siguen el modelo explicado en la sección 4.1.1. Cuando recibe un mensaje, lo envía a la componente *MsgChecker* para que compruebe que la acción que ese mensaje requiere ejecutar, la puede ejecutar el usuario que está enviando ese mensaje. Para ello lo primero que se hace es obtener el nombre de usuario a partir de la sesión. Para ello esta componente tiene un *HashMap* en el que se almacenan las sesiones, teniendo como clave el nombre de usuario. Una vez se ha obtenido el nombre de usuario se envían estos datos a la componente *MsgChecker* junto con el mensaje recibido.

Antes de continuar explicando el comportamiento o la funcionalidad de la componente *WebSocketController* hay que explicar *MsgChecker* ya que dependiendo de la respuesta que esta clase le da a *WebSocketController* esta se comportará de una forma u otra. *MsgChecker* devuelve una lista de string en la que el primer elemento es la respuesta y el resto de elementos son los datos necesarios para realizar la operación que conlleva esa respuesta.

Lo primero que comprueba *MsgChecker* es el tipo de mensaje:

- **CHAT**: recupera de la base de datos de chat la sala activa del usuario y devolverá **CHATOK** y el nombre de la sala activa.
- **VERIFY**: comprueba si el usuario ya está en la base de datos y, si no esta, comprueba si se puede insertar un nuevo usuario (si el número de usuarios del sistema es menor que 100) y en caso de que se pueda crea un nuevo usuario y lo inserta. En caso de que sea un nuevo usuario y no se pueda insertar un nuevo usuario, *MsgChecker* devolverá **VERIFYNOK**, en cualquier otro caso devolverá **VERIFYOK** y como segundo elemento el nombre del usuario. Si *WebSocketController* recibe **VERIFYOK** unirá la cola del usuario a la centralita de **ROOT** y le enviará un mensaje de tipo *NOTIFICATION* a la cola del usuario.
- **COMMAND**: para todos los comandos lo primero que hace es comprobar si tiene el número de parámetros adecuados o los permisos adecuados. Para cada uno de los comandos realiza unas consultas y/o modificaciones en la base de datos de chat.

Para cada uno de los comandos se va a explicar lo que haría *MsgChecker*, la respuesta que le daría a *WebSocketController* y lo que haría *WebSocketController*.

Lo primero de todo se van a explicar las respuestas que están ligadas a varios comandos. Si *WebSocketController* recibe una de estas respuestas lo único que hace es notificar al usuario enviando el mensaje correspondiente a su cola en *Rabbit*. Estas respuestas son las siguientes:

- **NOTKNOWN**: el comando no es conocido.
- **WRONGCOMMAND**: el número de parámetros del comando es incorrecto.
- **NOTADMIN**: un usuario que no es administrador de la sala está intentando ejecutar un comando que solo puede ejecutar el administrador de la sala (INVITEROOM, DELETEROOM, KICKROOM).
- **NOTROOT**: un usuario, que no es el administrador del sistema, está intentando ejecutar el comando BROADCAST.
- **NOACTIVEROOM**: un usuario está intentando enviar un mensaje de tipo CHAT pero no tiene ningún chat activo.
- **ROOMNOTEXISTS**: un usuario está intentando acceder (OPENROOM o JOINROOM), invitar (INVITEROOM), borrar (DELETEROOM) o echar a alguien (KICKROOM) de una sala que no existe.
- **ROOMEXISTS**: un usuario está intentando crear (CREATEROOM) una sala que ya existe.
- **USERINROOM**: el administrador de una sala está intentando invitar (INVITEROOM) a un usuario que ya está en esa sala.
- **USERNOTEXISTS**: un usuario está intentando enviar un mensaje privado (CHATUSER) a un usuario que no existe.
- **USERNOTROOM**: el administrador está intentando invitar (INVITEROOM) o echar (KICKROOM) a un usuario que no existe.

- **NOTINVITED**: un usuario está intentando unirse a una sala (**JOINROOM**) a la que no ha sido invitado.

A continuación se va a explicar la respuesta que *MsgChecker* daría a *WebSocketController*, dependiendo del comando que el usuario quiera ejecutar y lo que haría cada una de estas componentes.

- **HELP**:
 - *MsgChecker*: crea un string que contiene la ayuda para ejecutar los comandos.
 - Respuesta: **HELPOK** + string creado
 - *WebSocketController*: envia un mensaje de tipo **NOTIFICATION** al usuario que ha ejecutado este comando con el string que recibe como segundo elemento de la respuesta recibida.
- **CREATEROOM** *id_room*:
 - *MsgChecker*: comprueba si en la base de datos existe una sala con el nombre *id_room*. Si no existe, crea una nueva sala cuyo administrador y único usuario sea el usuario que ha ejecutado este comando y devuelve **CREATEOK**. Además, la nueva sala pasa a ser la sala activa del usuario. Si no, devuelve **ROOMEXISTS**
 - Respuesta: **CREATEOK** + *id_room*
 - *WebSocketController*: envia un mensaje de tipo **NOTIFICATION** al usuario y dos mensajes de tipo **CHAT** a la nueva sala.
- **INVITEROOM** *id_room id_user*:
 - *MsgChecker*: comprueba si el usuario que lo ejecuta es el administrador de la sala *id_room* y si el usuario *id_user* existe y no está en la sala. Si el usuario no existe devuelve **USERNOTE-XISTS**. Si el usuario está en la sala devuelve **USERINROOM**. En cualquier otro caso devuelve **INVITEOK**.
 - Respuesta: **INVITEOK** + *id_user* + *id_room*

- *WebSocketController*: envia un mensaje de tipo *NOTIFICATION* al usuario con el comando necesario para unirse a la sala y un mensajes de tipo *CHAT* a la sala *id_room*.

■ **JOINROOM** *id_room*:

- *MsgChecker*: comprueba si el usuario que lo ejecuta ha sido invitado a la sala *id_room*. En este caso, devuelve **JOINOK** y se modifica la sala activa del usuario en la base de datos. En caso contrario, devuelve **NOTINVITED**.
- Respuesta: **JOINOK** + *id_room*
- *WebSocketController*: une la cola del usuario a la centralita de la sala *id_room*. Además envia un mensaje de tipo *NOTIFICATION* y otro de tipo *CLEAN* a la cola usuario y un mensajes de tipo *CHAT* a la centralita sala *id_room*.

■ **LEAVEROOM** *id_room*:

- *MsgChecker*: comprueba si el usuario que lo ejecuta pertenece a la sala *id_room*. Si esta sala es su sala activa, cambia la sala activa del usuario a null en la base de datos. Si el usuario pertenece a la sala *id_room* devuelve **LEAVEOK**. En caso contrario devuelve **USERNOTROOM**.
- Respuesta: **LEAVEOK** + *id_room*
- *WebSocketController*: desune la cola del usuario a la centralita de la sala *id_room*. Además envia un mensaje de tipo *NOTIFICATION* y otro de tipo *CLEAN* (solo si la sala activa del usuario era *id_room*) a la cola usuario y un mensajes de tipo *CHAT* a la centralita sala *id_room*.

■ **GETROOMS**:

- *MsgChecker*: crea un string con los nombres de las salas a las que pertenece el usuario que ejecuta el comando, separadas por comas. En caso de que el usuario no pertenezca a ninguna sala este string contendrá "No tienes salas". Siempre devuelve **ROOMSOK**.

- Respuesta: **ROOMSOK** + string creado
 - *WebSocketController*: envía un mensaje de tipo *NOTIFICATION* a la cola usuario con el string recibido como segundo elemento.
- **GETUSERSROOM** *id_room*:
- *MsgChecker*: crea un string con los nombres de los usuarios a las que pertenecen a la sala *id_room*, separadas por comas. En caso de que el usuario no pertenezca a la sala este string contendrá “No estas en la sala”. Siempre devuelve **USERSROOMOK**.
 - Respuesta: **USERSROOMOK** + string creado
 - *WebSocketController*: envía un mensaje de tipo *NOTIFICATION* a la cola usuario con el string recibido como segundo elemento.
- **CLOSEROOM**:
- *MsgChecker*: comprueba si, en la base de datos, el usuario tienen alguna sala activa. En caso de que su sala activa sea null, devuelve **NOACTIVEROOM**. En caso contrario devuelve **CLOSEOK**
 - Respuesta: **CLOSEOK** + string creado
 - *WebSocketController*: envía un mensaje de tipo *CLEAN* a la cola usuario.
- **OPENROOM** *id_room*:
- *MsgChecker*: comprueba si el usuario pertenece a esa sala. En caso de que no pertenezca devuelve **NOTINVITED**. En caso contrario, se actualiza su sala activa en la base de datos a *id_room* y se recuperan los mensajes de esa sala.
 - Respuesta: **OPENOK** + *id_room*
 - *WebSocketController*: une la cola del usuario que ejecuta ese comando a la centralita de la sala *id_room*.
- **DELETEROOM** *id_room*:

- *MsgChecker*: comprueba si el usuario es el administrador de la sala y si la sala existe. En caso de que no lo sea devuelve **NOTADMIN**. Si la sala no existe devuelve **ROOMNOTEXISTS**. En caso contrario, elimina la sala y los mensajes de la base de datos y crea una lista con los nombres de los usuario que pertenecen a esa sala.
- Respuesta: **OPENOK** + *id_room* + lista usuarios
- *WebSocketController*: envía un mensaje de tipo *NOTIFICATION* a la cola usuario. Desune las colas de todos los usuarios de la lista de usuario y envía un mensaje de tipo *NOTIFICATION* a sus colas. Por último, elimina la centralita.

■ **KICKROOM** *id_room id_user*:

- *MsgChecker*: comprueba si el usuario es el administrador de la sala, si la sala existe y si el usuario *id_user* pertenece a la sala. En caso de que no lo sea devuelve **NOTADMIN**. Si la sala no existe devuelve **ROOMNOTEXISTS**. Si el usuario no pertenece a la sala devuelve **USERNOTROOM**. En caso contrario, si la sala activa del usuario *id_user* es la sala *id_room* modifica la sala activa del usuario *id_user* en la base de datos para que sea null y devuelve **KICKROK**
- Respuesta: **KICKROK** + *id_user* + *id_room*
- *WebSocketController*: envía un mensaje de tipo *NOTIFICATION* a la cola usuario. Desune la cola del usuario *id_user* y envía un mensaje de tipo *CHAT* a la sala *id_room*.

■ **CHATUSER** *id_user*:

- *MsgChecker*: comprueba si el usuario *id_user* existe y si existe una sala privada para estos dos usuarios. Si no existe el usuario *id_user* devuelve **USERNOTEXISTS**. Si no existe una sala privada para estos dos usuarios se crea una nueva sala, la sala activa del usuario que ejecuta el comando pasa a ser esta sala y devuelve **CHATUSERCREATE**. En caso de que si que exista una sala

para estos dos usuarios, se recuperan los mensajes, se modifica la sala activa del usuario que ejecuta este comando y devuelve **CHATUSERMSG**.

- Respuesta: **CHATUSERCREATE** + nombre sala creada + *id_user*
- *WebSocketController*: une la cola de cada uno de los usuarios a la centralita de la nueva sala creada y envía mensajes de tipo *CHAT* a esta sala para notificar que se ha creado y que se ha unido el usuario que ha ejecutado este comando.
- Respuesta: **CHATUSERMSG** + nombre sala creada
- *WebSocketController*: no hace nada

■ **BROADCAST** *msg*:

- *MsgChecker*: comprueba si el usuario es el administrador del sistema. En caso de que no lo sea devuelve **NOTROOT**. En caso contrario devuelve **BROADCASTOK**.
- Respuesta: **KICKKROK** + *msg*
- *WebSocketController*: envía el mensaje *msg* a la centralita de root.

■ **ADDCEASURE** *word*:

- *MsgChecker*: comprueba si el usuario es el administrador del sistema. En caso de que no lo sea devuelve **NOTROOT**. En caso contrario devuelve **ADDCEASUREOK**.
- Respuesta: **ADDCEASUREOK** + *word*
- *WebSocketController*: envía una petición HTTP al microservicio de censura para añadir la palabra *word* a la base de datos de censura. Envía un mensaje de tipo *NOTIFICATION* al administrador del sistema.

■ **REMOVECEASURE** *word*:

- *MsgChecker*: comprueba si el usuario es el administrador del sistema. En caso de que no lo sea devuelve **NOTROOT**. En caso contrario devuelve **REMCCEASUREOK**.

- Respuesta: **REMCENSUREOK** + *word*
- *WebSocketController*: envía una petición HTTP al microservicio de censura para eliminar la palabra *word* de la base de datos de censura. Envía un mensaje de tipo *NOTIFICATION* al administrador del sistema.

■ **GETCENSURE:**

- *MsgChecker*: comprueba si el usuario es el administrador del sistema. En caso de que no lo sea devuelve **NOTROOT**. En caso contrario devuelve **GETCENSUREOK**.
- Respuesta: **GETCENSUREOK**
- *WebSocketController*: envía una petición HTTP al microservicio de censura para obtener una lista de las palabras censuradas de la base de datos de censura. Envía un mensaje de tipo *NOTIFICATION* al administrador del sistema con esta lista.

A lo largo de esta explicación se comenta que se comprueba si un usuario pertenece a una sala. Para comprobar si un usuario está en una sala determinada o si ha sido invitado se obtienen los timestamps de los mensajes del chat, de cuando el usuario se unió a la sala o de cuando el usuario fue invitado y de cuando el usuario abandonó la sala (si es que existen). Comparando estos timestamps se puede comprobar si un usuario pertenece a una sala o si un usuario a sido invitado a una sala. Todos los mensajes de tipo *CHAT* que son enviados a las centralitas o a las colas se almacenan en la base de datos.

De este tier faltarían por comentar dos componentes: *RabbitMQAdapter*, *DBAdministrator* y *FileController*.

RabbitMQAdapter se encarga de la conexión con *CloudAMQP*. Para ello esta componente representa a una clase que tiene las funciones necesarias para realizar la conexión, para crear o eliminar colas o centralitas, para unir o desunir colas a centralita y para enviar mensajes a colas o centralitas. Para ello cuando se crea una cola, se le asigna una función para que cuando reciba un mensaje, lo envíe a la sesión del usuario en caso de que este conectado. Además, cuando un usuario envía un mensaje a una centralita, este mensaje

se almacenará en la base de datos. De esta forma se asegura que un mensaje sólo se almacena en la base de datos si se ha enviado.

DBAdministrator se encarga de la conexión con la base de datos. En concreto con el esquema de chat. Esta componente representa a una clase que tiene las funciones necesarias para crear o eliminar usuarios, salas o chat y para realizar las consultas que se consideren necesarias como consultar si un usuario ha sido invitado a una sala o si un usuario pertenece a una sala.

FileController se encarga de almacenar los ficheros que recibe del cliente y de recuperar un fichero determinado cuando recibe la petición de descarga. En la sección 4.2.1 ya se han explicado las diferentes peticiones que se pueden realizar a esta componente. Aquí solo cabe destacar que, este no se conecta directamente con la base de datos, si no que llama a las funciones de una clase que es la encargada de realizar la conexión de esta componente con la base de datos.

4.2.3. Tier 2: Datos

En este nivel se encuentra la base de datos MySQL. Esta base de datos tiene dos esquemas, el esquema de chat y el esquema de ficheros. Ambos esquemas están explicados en la sección 4.1.2.

Esta componente recibe *queries* y realiza las modificaciones necesarias o devuelve los datos correspondientes con esa query. La conexión y la comunicación se realizan mediante JDBC.

4.2.4. Microservicio

En la figura 5 se puede observar como el componente *WebSocketController* del tier 1 explicado en la sección 4.2.2, se comunica con este microservicio vía HTTP. Este microservicio tiene un componente llamado *CensuraController* que recibe estas llamadas.

En la sección 4.2.2 también se ha comentado que estas llamadas o peticiones se realizaban el utilizar los comandos ADDCENSURE, REMOVECENSURE y GETCENSURE. En esta sección se va a especificar que llamadas se pueden realizar a esta componente y como se gestionan.

El componente *CensuraController* puede gestionar las siguientes peticiones:

■ **POST /censureFilter:**

- Parámetros:
 - msg: mensaje que se desea censurar
 - sender: usuario que ha enviado ese mensaje
- Descripción: dado el mensaje *msg*, recupera las palabras que se deben censurar, almacenadas en la base de datos, y comprueba si dicho mensaje tiene alguna de esas palabras. En caso afirmativo sustituya esas palabras por “****” y almacena el mensaje sin censurar, junto con las palabras censuradas y *sender* en la base de datos de censura.
- Devuelve: el mensaje censurado en caso de que se haya aplicado censura o el mensaje original en caso contrario.

■ **POST /addCensure:**

- Parámetros: word: palabra que se desea añadir a la base de datos de palabras que se han de censurar.
- Descripción: añade a la base de datos de censura la palabra *word*.
- Devuelve: un mensaje afirmando o denegando si se ha insertado correctamente la palabra en la base de datos.

■ **POST /removeCensure:**

- Parámetros: word: palabra que se desea eliminar de la base de datos de palabras que se han de censurar.
- Descripción: elimina de la base de datos de censura la palabra *word* en caso de que esté almacenada.
- Devuelve: un mensaje afirmando o denegando si se ha eliminado correctamente la palabra de la base de datos.

■ **GET /censureWords:**

- Descripción: recupera de la base de datos las palabras que se han de censurar.
- Devuelve: devuelve una lista con estas palabras.

El componente *BD MySQL Censura* es igual que la descrita en la sección 4.2.3, es una base de datos MySQL. La única diferencia es que esta base de datos tiene un único esquema en el que se almacenan los ficheros. Este esquema está explicado en la sección 4.1.2.

4.3. Vista de distribución

En esta sección se va a hablar de cómo el sistema está distribuido, en cuántas máquinas se está ejecutando y qué se está ejecutando en cada máquina. También se va a hablar de cómo está relacionado con el diagrama de componentes (figura 5). Para ello, esta sección va a estar centrada en el diagrama de despliegue que se puede observar en la figura 6.

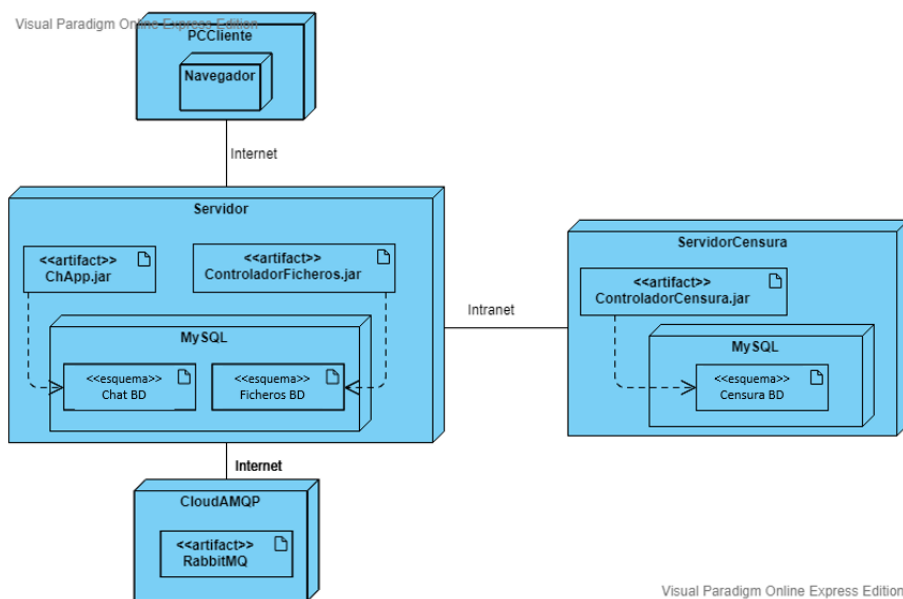


Figura 6: Diagrama de despliegue

En la figura 6 se puede observar el diagrama de despliegue antes de que el cliente se conecte. Una vez el cliente se conecte, el sistema estaría completo.

En la figura 6 se pueden ver cuatro máquinas diferentes:

- PCCliente: esta es la máquina del cliente. El cliente se conecta al sistema a través del navegador, utilizando Internet.
- Servidor: en esta máquina se encuentran dos archivos JAR diferentes, el de la aplicación chat (llamada ChApp) y el de los ficheros. En esta máquina también se encuentra la base de datos.
- ServidorCensura: en esta máquina o nodo se encuentra el servidor de censura y la base de datos. Esta máquina y la máquina del servidor están conectadas a través de Intranet.
- CloudAMQP: esta máquina esta en cloud y en esta máquina se encuentra RabbitMQ.

4.3.1. PC Cliente

Esta es la máquina del cliente. Esta máquina se conectará a la máquina del servidor a través de Internet. Una vez están conectadas se descargarán en la máquina del cliente los ficheros necesarios para ejecutar la aplicación (html, javascript y css).

4.3.2. Servidor

En esta máquina o nodo se encuentran dos aplicaciones: ChApp y ControladorFicheros y una base de datos MySQL con dos esquemas. Esta máquina o nodo tiene la IP 192.168.1.134.

La aplicación ChApp incluye el servidor web Tomcat. Esta aplicación está en el puerto 8080. Esta aplicación incluye los siguientes componentes:

- *WebSocketController*
- *MsgChecker*
- *RabbitAdapter*
- *DBAdministrator*

La aplicación ControladorFicheros también incluye el servidor web Tomcat. Esta aplicación está en el puerto 8000. Esta aplicación incluye el componente *FileContoller*.

Por último, las bases de datos MySQL se encuentran en el puerto 3306. La aplicación ChApp utiliza el esquema de Chat BD, mientras que la aplicación ControladorFicheros utiliza el esquema Ficheros BD. Esta base de datos incluye el componente *BD MySQL chat*.

Esta máquina tiene dos aplicaciones y una base de datos con dos esquemas por lo que las aplicaciones de esta máquina se podrían ejecutar en dos máquinas distintas porque como ya se ha comentado en la sección 4.2.2 las componenets de estas dos aplicaciones se comunican mediante HTTP por lo que para separarlas simplemente habría que modificar la dirección IP de dichas peticiones.

4.3.3. Servidor Censura

Esta maquina tiene la IP 192.168.1.138. En esta máquina se encuentra la aplicación de ControladorCensura que incluye el servidor web Tomcat escuchando las peticiones en el puerto 9000.

En cuanto a las componentes, la aplicación ControladorCensura contendría las componentes del tier 0 del micro-servicio, es decir, la componentes *CensuraController*.

En esta máquina también hay una base de datos MySQL que se encuentra en el puerto 3306. La aplicación ControladorCensura utiliza el esquema *Censura BD* de esta base de datos.

4.3.4. CloudAMQP

Esta máquina se encuentra en la nube, al ser una aplicación de terceros no se puede saber con certeza que aplicaciones se ejecutan dentro de esa máquina o cómo se ejecutan. En esta máquina se encontraría la componente *CloudAMQP*, encargada de gestionar el sistema de colas.

4.4. Comportamiento del sistema

Para explicar mejor el comportamiento de sistema se van a utilizar varios diagramas de secuencia en los que se muestra qué componentes o clases actúan, que funciones se utilizan y en qué orden y que mensajes se intercambian. En estos diagramas pueden aparecer objetos que no aparecían en el diagrama de componentes (imagen 5). Esto es debido a que, como ya se ha mencionado, hay objetos o clases que se utilizan para realizar las comunicaciones entre dos componentes por lo que estos no aparecen en el diagrama de componentes.

En estos diagramas se van a mostrar tres casos diferentes:

- Un usuario envía un mensaje a su sala activa (figura 7)
- Un usuario envía un fichero a su sala activa (figura 8)
- Un usuario ejecuta el comando: **CREATEROOM** *id_room* (figura 9)

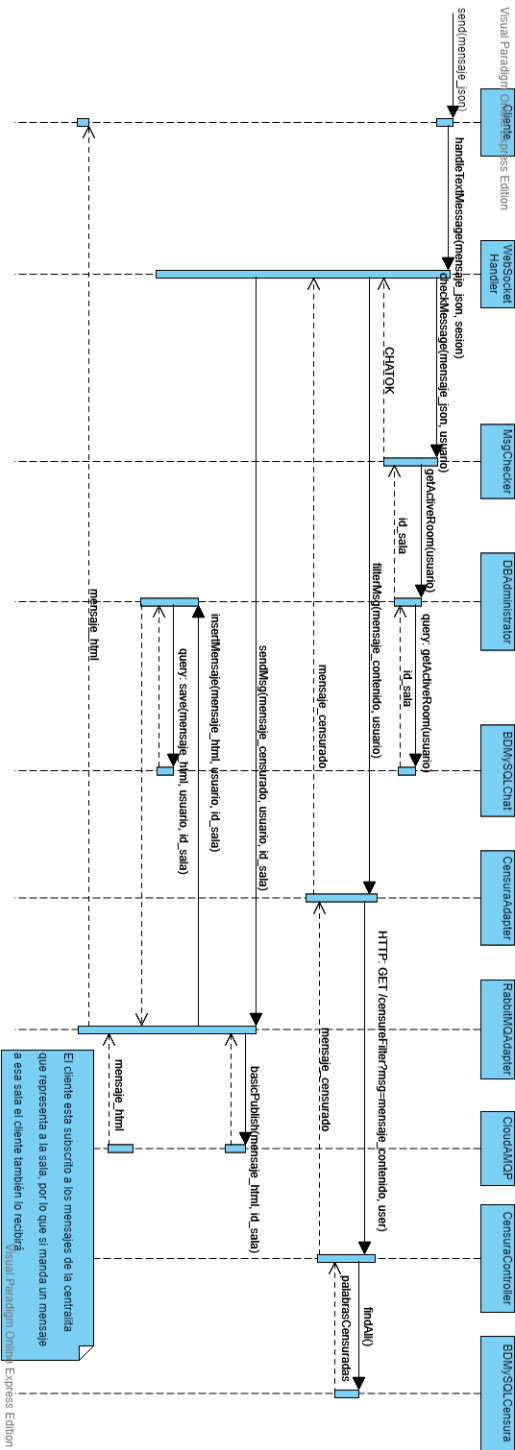


Figura 7: Diagrama de secuencia: enviar mensaje

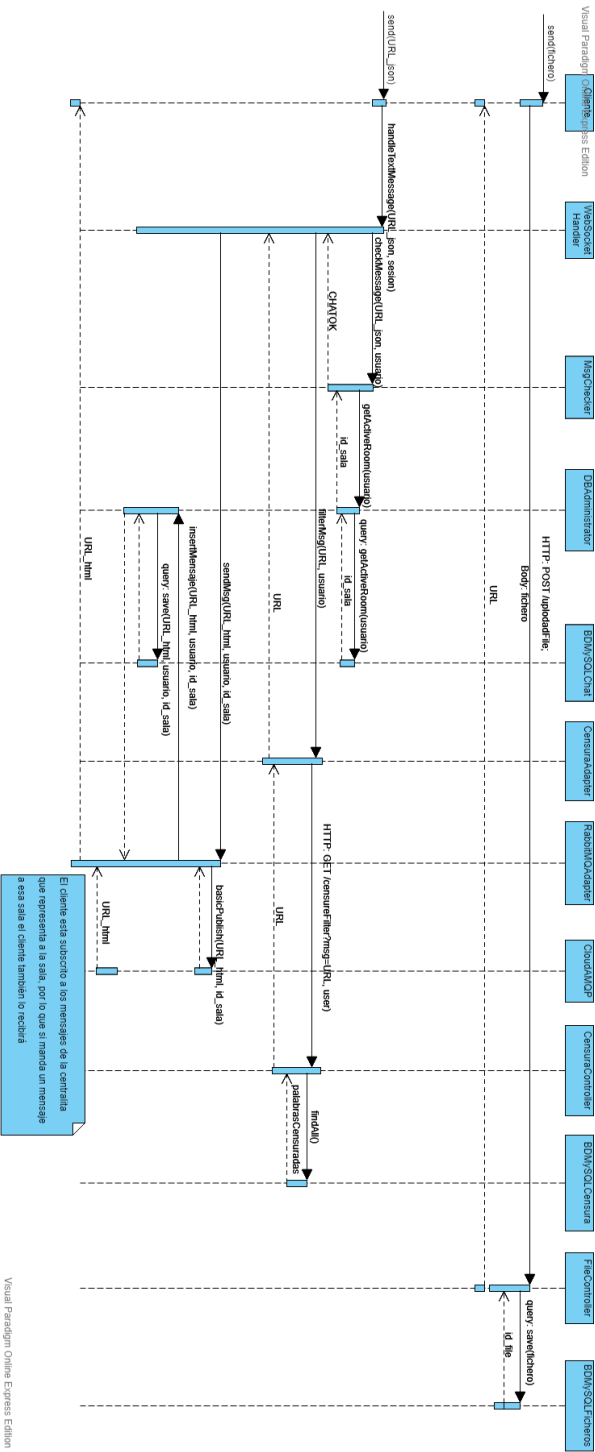


Figura 8: Diagrama de secuencia: enviar fichero

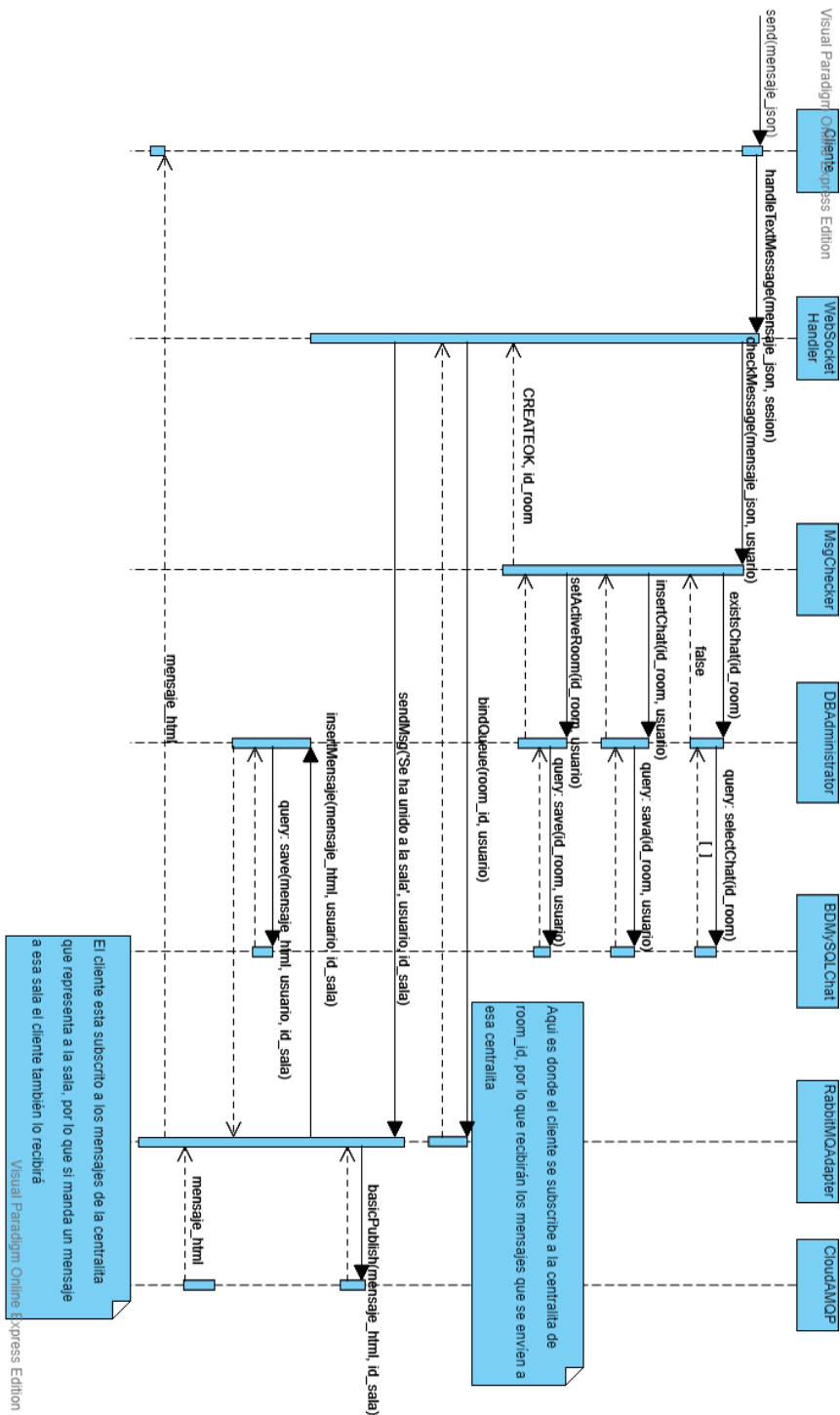


Figura 9: Diagrama de secuencia: crear sala

5. Evaluación del sistema

Para evaluar el rendimiento del sistema se han realizado dos pruebas diferentes:

- Prueba 1: comprobar cuantos usuarios puede aguantar el sistema
- Prueba 2: comprobar el tiempo que tarda en procesarse un mensaje conforme aumenta el número de usuarios activos en el sistema

Para realizar estas pruebas se ha creado una aplicación que envía mensajes mediante web sockets al servidor. Esta aplicación, dependiendo del número de la prueba, envía mensajes de un tipo u otro. Si son mensajes de tipo CHAT, los envía cada 2 segundos.

Además, se ha creado un script en el bash de Windows que lanza 100 pestañas en modo incógnito del navegador Chrome. Como este navegador puede resultar muy pesado en algunos ordenadores, se han abierto 50 pestañas en un ordenador y 50 pestañas en otro. Ambos ordenadores que actúan de clientes son distintos al ordenador en el que se encuentra el servidor, uno de ellos con Windows 8 y el otro con Windows 10. Ambos con 8GB de RAM y una tarjeta gráfica AMD.

5.1. Número de usuarios

Esta prueba esta basada en una de las limitaciones de utilizar *Rabbit* y es que, en su versión gratuita, sólo permite crear 100 colas (figura 10), por lo que el sistema sólo puede tener 100 usuarios.

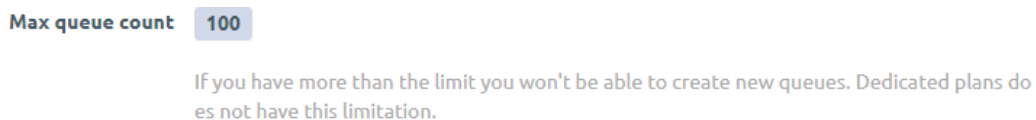


Figura 10: Número máximo de colas Rabbit

Teniendo en cuenta esto se ha limitado el número de usuarios que se pueden registrar a 100 de modo que si hay 100 usuarios en el sistema y se intenta registrar un nuevo usuario, el sistema no le dejará. Este comportamiento se puede observar en la figura 11.

Para realizar esta prueba se ha ejecutado un script que registra automáticamente 100 usuarios. Para comprobar que no se puede registrar un usuario más, se ha intentado registrar manualmente el usuario 101. Los resultados de esta prueba se pueden observar en las figuras 11 y 10.

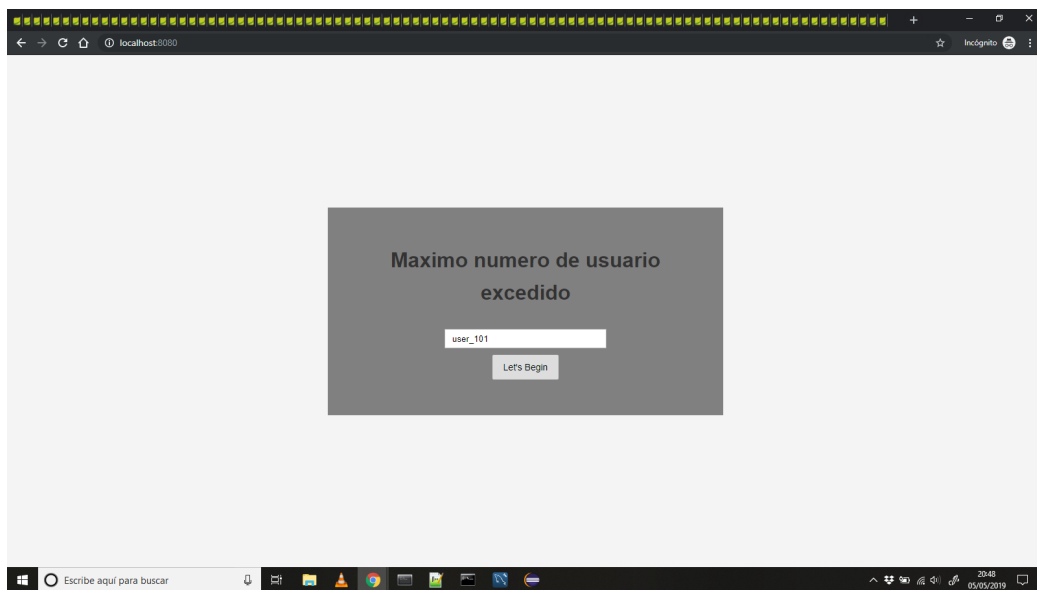


Figura 11: Número máximo de usuarios excedido

```
user_90: access
user_80: access
user_68: access
user_75: access
user_91: access
user_92: access
user_3: access
user_93: access
user_7: access
user_94: access
user_20: access
user_23: access
user_24: access
user_40: access
user_95: access
user_99: access
user_58: access
user_100: access
user_74: access
user_19: access
user_72: access
user_1: access
user_97: access
user_30: access
user_69: access
user_78: access
user_76: access
user_96: access
user_98: access
user_14: access
user_5: access
user_37: access
user_67: access
user_101: denied
```

Figura 12: Usuarios aceptados y usuario 101 denegado

5.2. Tiempo de procesamiento de mensajes

La segunda prueba realizada mide el tiempo que tarda en procesarse un mensaje dependiendo del número de usuarios activos que haya y del número de salas. En esta prueba, un usuario activo es un usuario que envía un mensaje cada dos segundos. El tiempo de procesamiento de un mensaje es el tiempo que

tarda un mensaje desde que lo recibe el servidor (*WebSocketController*) hasta que es enviado utilizando *RabbitMQ*, pasando por *MsgChecker*, almacenando o modificando los datos necesarios en la base de datos y pasando por el servidor de censura. Este comportamiento o este procesamiento del mensaje está representado en el diagrama de secuencia de la figura 7

En la primera parte de esta prueba, se ha creado una única sala a la que se han invitado al resto de usuarios y todos se han unido. El número de usuarios activos, enviando mensajes a esa sala, se ha ido incrementando cada tres segundos.

En la segunda parte de esta prueba, cada usuario que se conectaba creaba su propia sala y enviaba mensajes a esa sala. El número de usuarios activos se ha ido incrementando cada tres segundos.

En la figura 13 se pueden observar los resultados. Como se puede ver, el número de salas no afecta en el tiempo de procesamiento, ya que el procesamiento del mensaje es independiente al número de salas (como se puede observar en el diagrama de secuencia de la figura 7). Lo que sí que influye en el tiempo de procesamiento es el número de usuarios activos, puesto que conforme aumenta el número de usuarios activos aumenta el número de peticiones a las que tiene que atender el servidor.

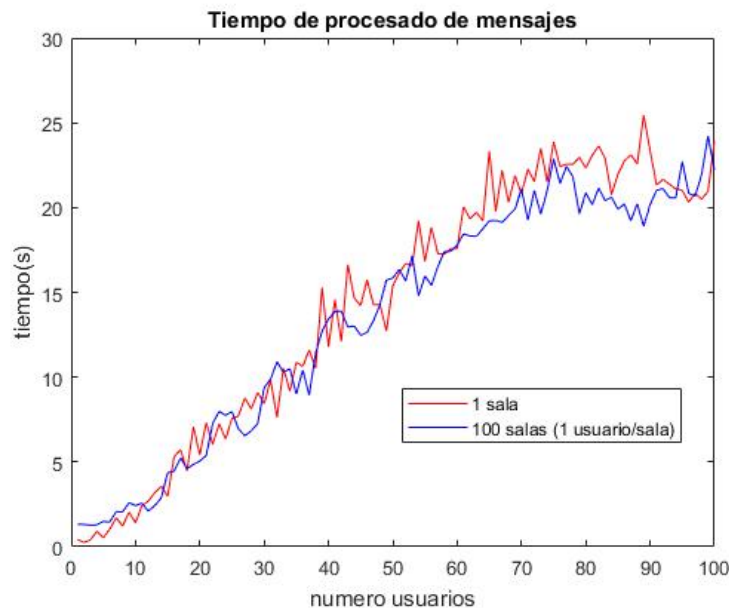


Figura 13: Tiempo de procesamiento de mensaje

5.3. Limitaciones

En esta sección se va a hablar de las limitaciones que tiene este sistema debido a las decisiones tomadas a lo largo del proyecto.

La mayor limitación radica en el uso de *Rabbit* y es que en su versión gratuita hay un límite de 100 colas, de 10000 mensajes acumulados en una cola y de 1000000 de mensajes enviados. Lo que significa que no puedes tener más de 100 usuarios en el sistema, ni tener acumulados más de 10000 mensajes ni enviar más de 1000000 mensajes (figura 14). Estos números, aunque algunos puedan parecer grandes, son muy pequeños para una aplicación de mensajería.

La otra opción sería obtener una versión de pago de *Rabbit*. Estas versiones o planes se pueden ver en la figura 15. En todos estos planes el número de colas y el número de mensajes es ilimitado, varían en la velocidad de los mensajes y en el número de conexiones. Teniendo en cuenta estos planes habría que pensar si sale rentable contratar una versión de pago de *Rabbit* o si habría que buscar otro tipo de solución para enviar los mensajes que

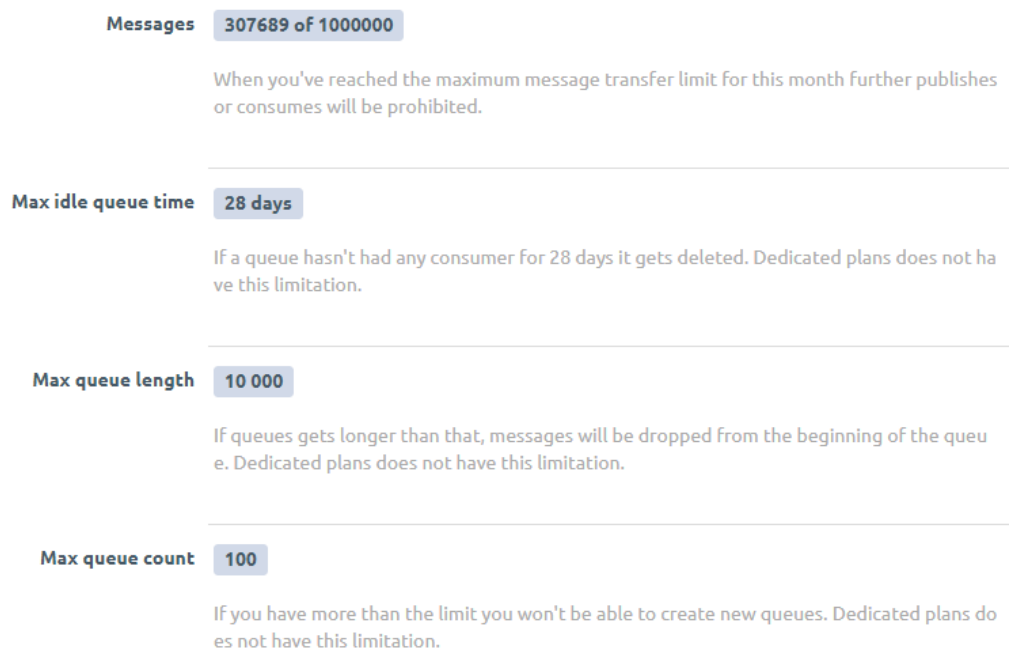


Figura 14: Limitaciones de rabbit

sustituyera a *Rabbit*.

La principal ventaja de *Rabbit* es que es muy cómodo de utilizar, puesto que te abstraes por completo de cómo funciona. Utilizando *Rabbit* con muy pocas funciones puedes tener un sistema de mensajería.

Otra posible limitación de este sistema, es la forma en la que se comprueba si un usuario pertenece a una sala o no. Como ya se ha comentado, esto se hace mirando los mensajes de la base de datos. El principal inconveniente de hacerlo de esta forma, es que conforme aumenta el número de mensajes almacenados en esa base de datos, también lo hace el tiempo para hacer este tipo de consulta. Con la creación de índices se podría reducir dicho tiempo, pero si aun así este tiempo siguiera siendo elevado habría que buscar otras soluciones como re-estructurar la base de datos de chat.

Otras posibles limitaciones vienen dadas por las conexiones entre las máquinas y las diferentes aplicaciones.







	Loud Lion - 1 node 400 k ~Max MSGs/s 160 k ~Max connections \$ 3499 PER MONTH Get Now
	Heavy Hippo - 1 node 200 k ~Max MSGs/s 80 k ~Max connections \$ 1999 PER MONTH Get Now
	Awesome Ape - 1 node 100 k ~Max MSGs/s 40 k ~Max connections \$ 999 PER MONTH Get Now
	Power Panda - 1 node 40 k ~Max MSGs/s 20 k ~Max connections \$ 499 PER MONTH Get Now
	Roaring Rabbit - 1 node 20 k ~Max MSGs/s 8 k ~Max connections \$ 299 PER MONTH Get Now
	Big Bunny - Only available with 1 node • 10k msg/s in burst 1 k ~Max MSGs/s 1 k ~Max connections \$ 99 PER MONTH Get Now

Figura 15: Planes de rabbit

Si se perdiera la conexión con la máquina en la que se encuentra la censura, los mensajes tardaría más en llegar, puesto que el sistema se queda esperando una respuesta de censura, y los mensajes no serían censurados. Una posible solución a este problema sería crear una replica de censura y, en caso de no recibir respuesta, enviar las peticiones a esta replica.

Si el cliente no se pudiera comunicar con la aplicación de ficheros, este no podría enviar ficheros a su sala activa, pero el resto del sistema seguiría funcionando. Este es el caso en el que solo se ha “caído” la aplicación de ficheros, si por algún motivo se perdiera la conexión con la máquina del servidor la aplicación no funcionaría. Lo mismo pasaría si la máquina del servidor perdiera la conexión con *Rabbit*, el sistema de mensajería dejaría de funcionar.

6. Requisitos cumplidos

En esta sección se van a explicar cómo la aplicación desarrollada cumple los requisitos establecidos en la sección 2.

6.1. Requisito 1

Este requisito dice lo siguiente: *Como usuario quiero poder enviar mensajes a otros usuarios para tener conversaciones puntuales.*

Para cumplir este requisito existe el comando **CHATUSER** *id_user* que crea una sala privada en la que sólo pueden hablar los dos usuarios. Todos los mensajes enviados en la ventana de mensajes tienen una longitud máxima de 500 caracteres limitada por el tamaño de la caja de mensajes, que solo permite escribir 500 caracteres.

Para enviar un mensaje a otro usuario el usuario debe:

1. Empezar o abrir una conversación con el usuario con el que quiere hablar → **CHATUSER** *id_user*
2. Enviar un mensaje → Escribir mensaje en la ventana de mensajes

Para esta prueba se han registrado dos usuarios *user_1* y *user_2*. Como se puede observar en las imágenes siguientes, para que *user_1* empiece una conversación privada con *user_2*, el primero debe ejecutar el comando **CHATUSER** *user_2* (Figura 16). Tras haber ejecutado este comando, esta sala

privada pasa a ser su sala activa y todos los mensajes que envíe por la ventana de mensajes, se enviarán a esta sala (Figura 17). Una vez *user_1* inicie la conversación *user_2* empezará a recibir notificaciones indicando que *user_1* le está enviando mensajes por una sala privada. Para ver estos mensajes y poder contestarle *user_2* tiene que ejecutar el comando **CHATUSER** *user_1* (Figura 18) que cargará todos los mensajes que se hayan enviado por esa sala privada y establecerá esta sala como sala activa del usuario *user_2* (Figura 19).

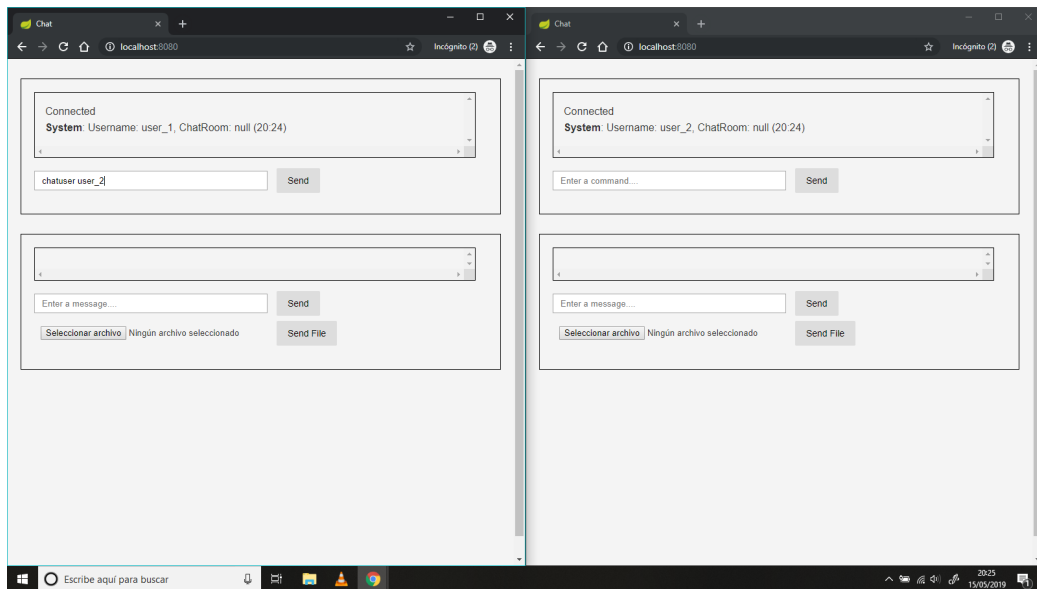


Figura 16: *user_1* inicia una conversación privada con *user_2*

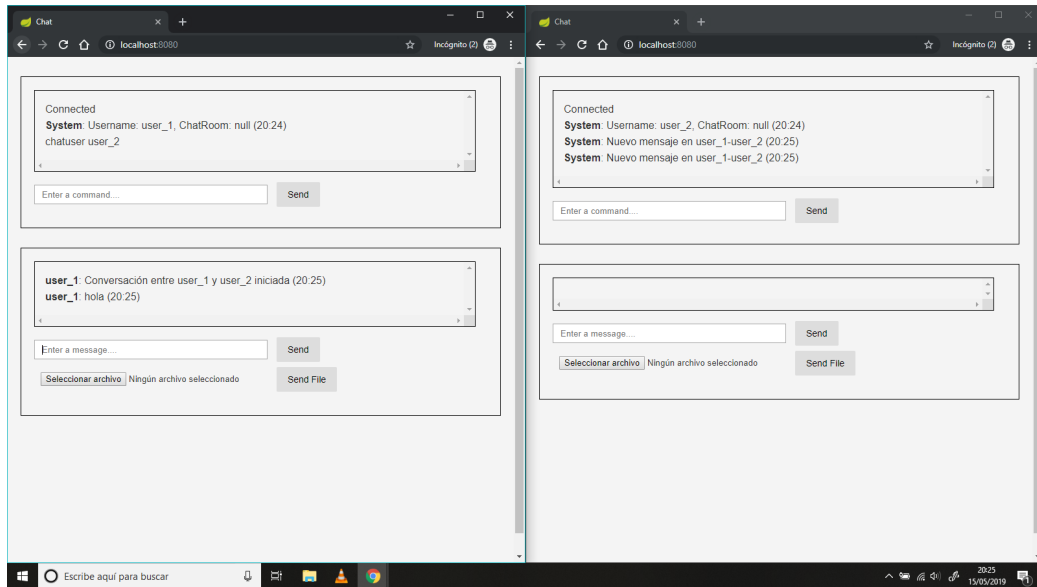


Figura 17: *user_1* envía mensaje a *user_2* por la sala privada

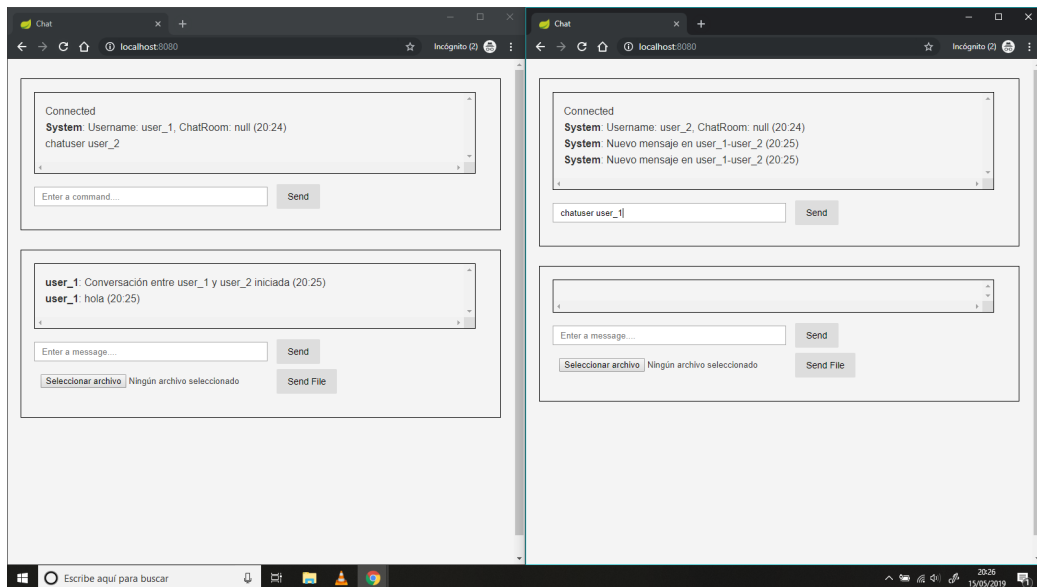


Figura 18: *user_2* recibe notificaciones de *user_1* y abre la conversación privada

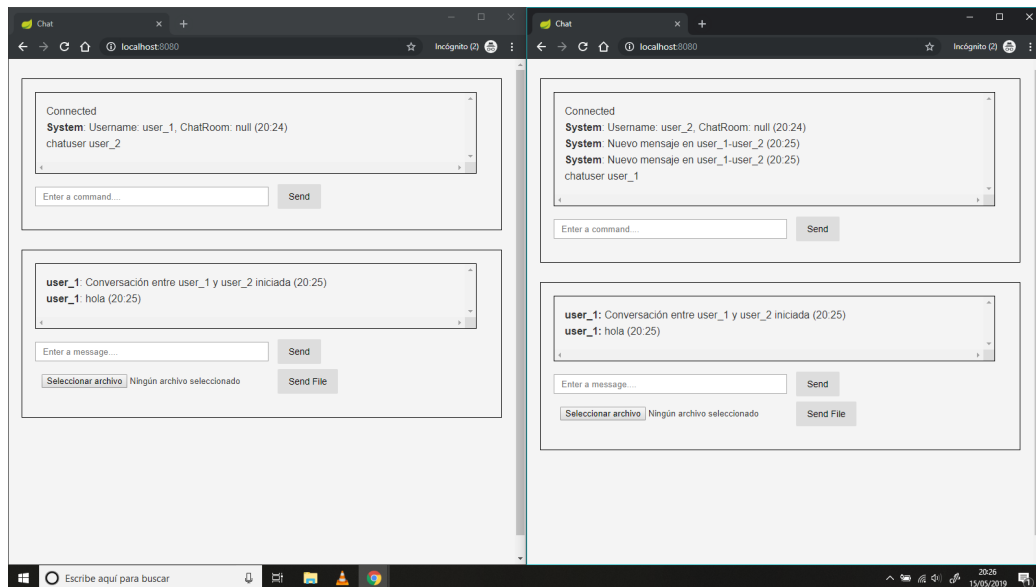


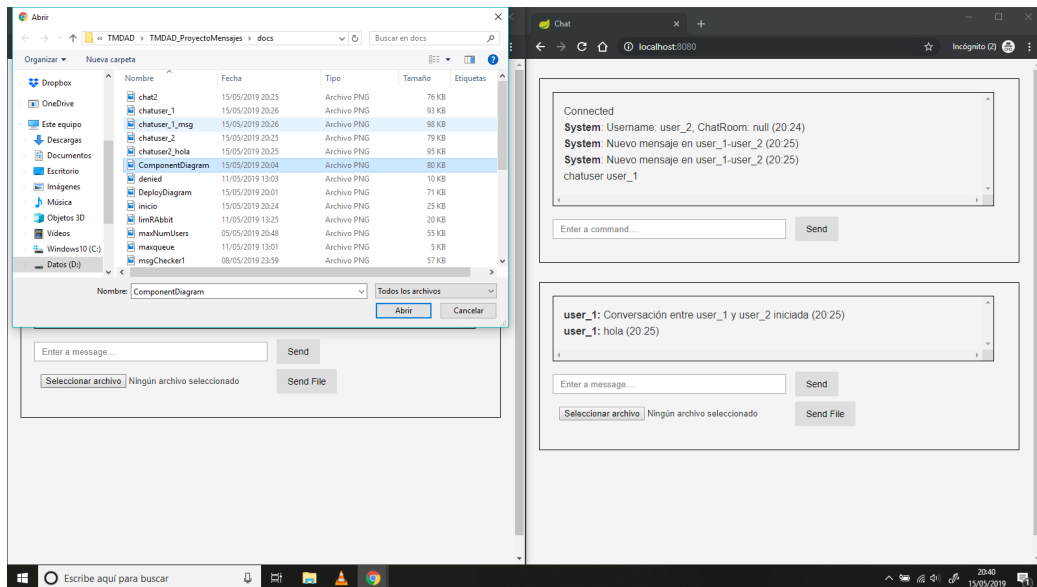
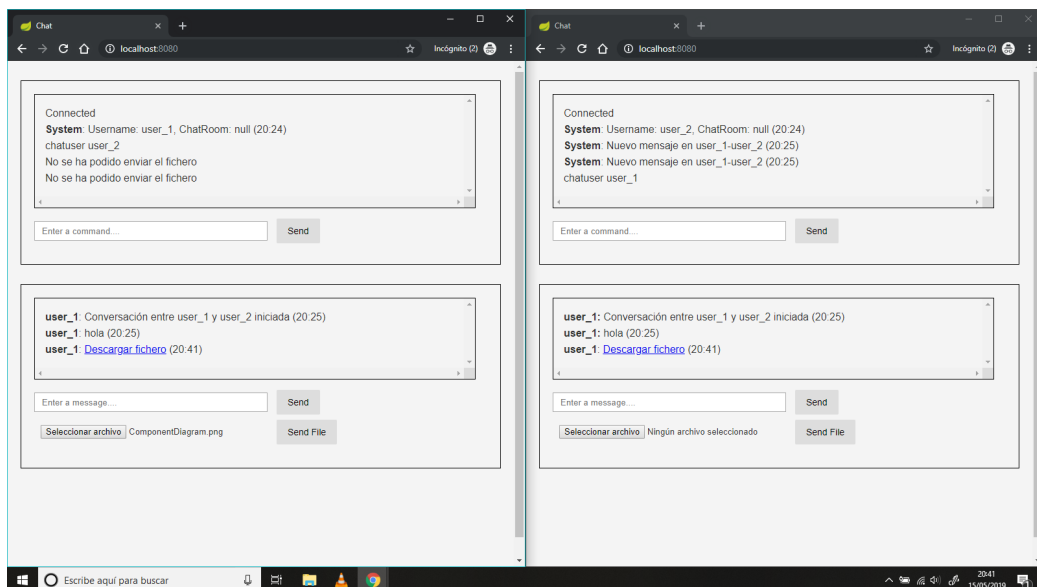
Figura 19: Los mensajes de la sala privada con *user_1* se han cargado en la ventana de mensajes de *user_2*

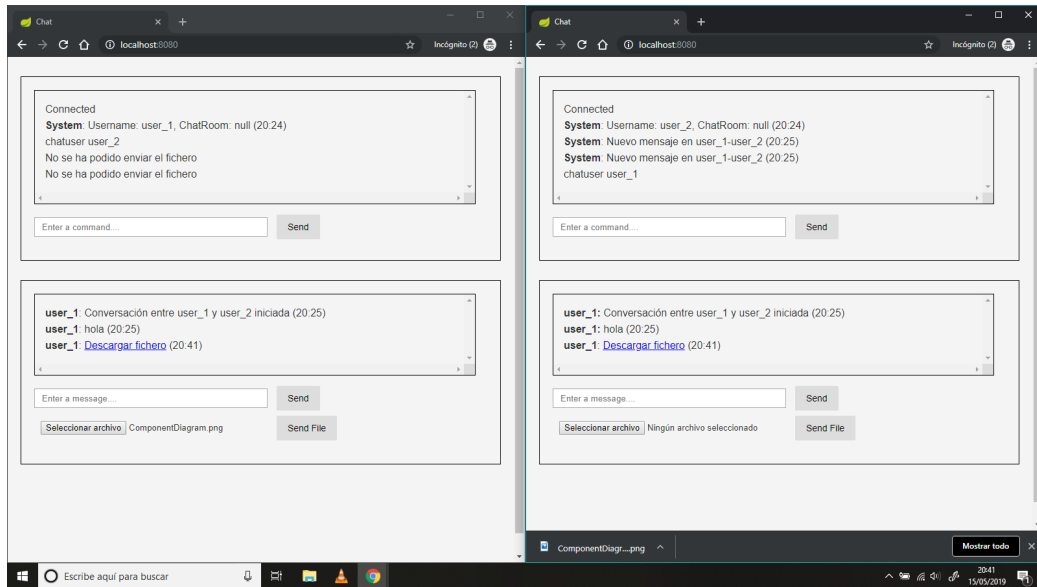
6.2. Requisito 2

Este requisito dice lo siguiente: *Como usuaria quiero poder compartir ficheros de mi equipo con otros usuarios para complementar las conversaciones puntuales.*

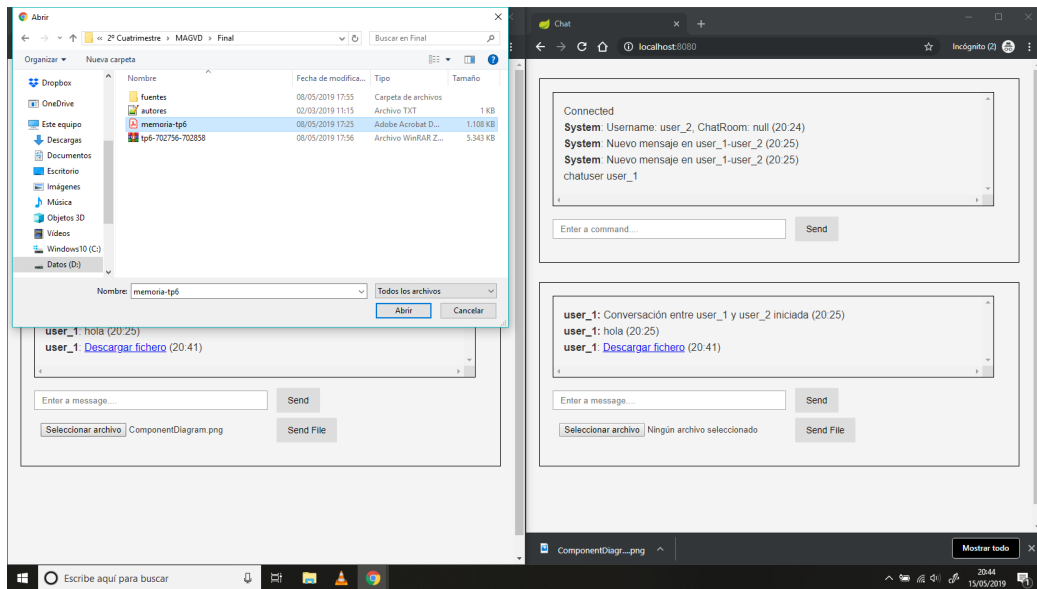
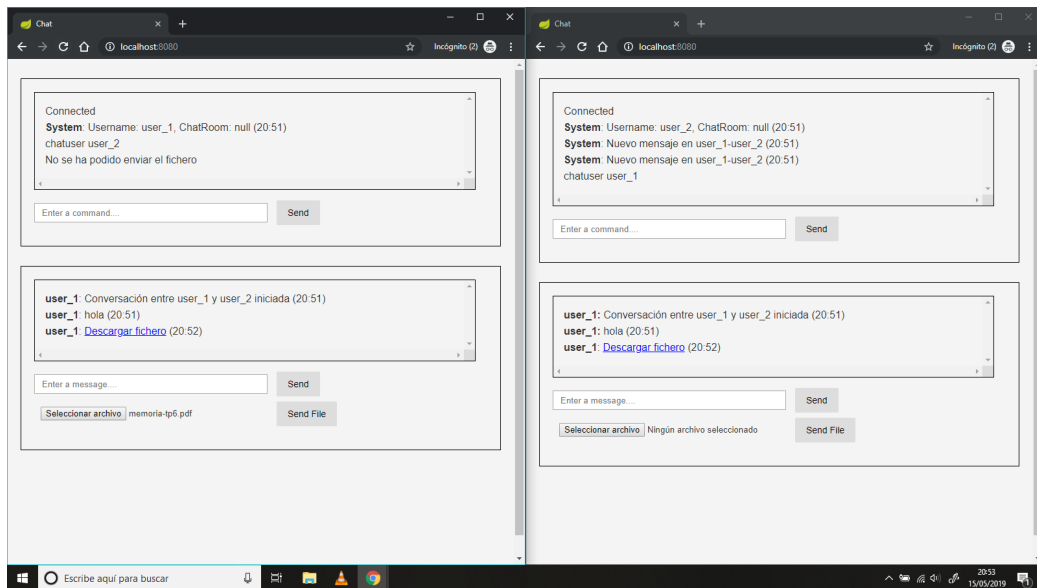
Para cumplir este requisito existen los botones “Seleccionar archivo” y “Send File” en la ventana de mensajes. Con estos botones el usuario seleccionará el archivo de desee enviar y lo enviará su sala activa. El tamaño máximo de estos ficheros es de 1MB. Este tamaño se ha limitado con el servidor REST de ficheros. Si el fichero excede este tamaño, el usuario recibirá una notificación y no se enviará este fichero.

Para mostrar el comportamiento de estos botones se va a volver a la sala privada creada en la sección anterior, con los usuario *user_1* y *user_2*. Supongamos que el usuario *user_1* envía el fichero *ComponentDiagram* por esa sala. Para ello el usuario *user_1*, seleccionará el fichero (Figura ??) y le dará al botón “Send File”. Ambos usuarios recibirán un enlace (Figura ??). Si el usuario *user_2* pulsa ese enlace, se descargará el fichero (Figura ??).

Figura 20: *user_1* selecciona el archivo *ComponentDiagram*Figura 21: *user_1* y *user_2* reciben un enlace para descargar el fichero

Figura 22: *user_2* descarga fichero

Ahora supongamos que el usuario *user_1* intenta enviar el archivo *memoria_tp6*, cuyo tamaño es mayor que 1MB (Figura 23). Tal y como se ha mencionado, en la figura x, se puede observar como este usuario recibe una notificación indicando que el archivo no se ha podido enviar (Figura 24).

Figura 23: *user_1* selecciona el archivo *memoria_tp6*Figura 24: *user_1* recibe notificación

6.3. Requisito 3

Este requisito dice lo siguiente: *Como usuario quiero poder crear salas de chat permanentes para tener un registro de las conversaciones pasadas.*

Para tener salas de chat permanentes, como se puede ver en el diagrama de secuencia de la figura 7, cada vez que se envía un mensaje se almacena en la base de datos. Además, al abrir una sala a la que un usuario pertenece con el comando **OPENROOM** *id_room*, se recuperan todos los mensajes de esa sala.

Para esta prueba, además de los usuarios *user_1* y *user_2*, se han registrado dos nuevos usuarios *user_3* y *user_4*. El usuario *user_1* crea la sala *room_1* con el comando **CREATEROOM** (Figura ??) e invita al resto de usuarios con el comando **INVITEROOM** (Figura ??). El resto de usuarios se une a la sala con el comando **JOINROOM** (Figura ??). Si el usuario *user_2* envía un mensaje, el resto de usuarios lo recibe (Figura ??). Si el usuario *user_3* decide cerrar la sala con el comando **CLOSEROOM**, éste empezará a recibir notificaciones cada vez que se envíe un mensaje a esa sala (Figura ??). Cuando quiera que la sala *room_1* vuelva a ser su sala activa, simplemente tendrá que ejecutar el comando **OPENROOM** *room_1*. Sin embargo, si el usuario *user_4* decide abandonar la sala y dejar de recibir mensajes, éste ejecutaría el mensaje **LEAVEROOM** *room_1* (Figura ??). De esta forma este usuario dejará de recibir mensajes de esta sala y, en caso de que se quiere volver a unir, necesitará una nueva invitación (Figura ??).

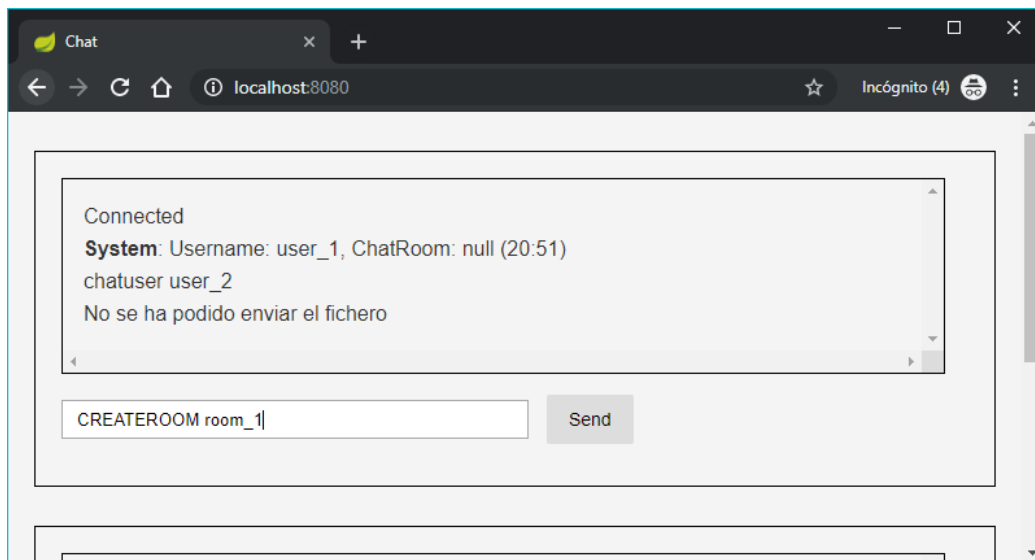
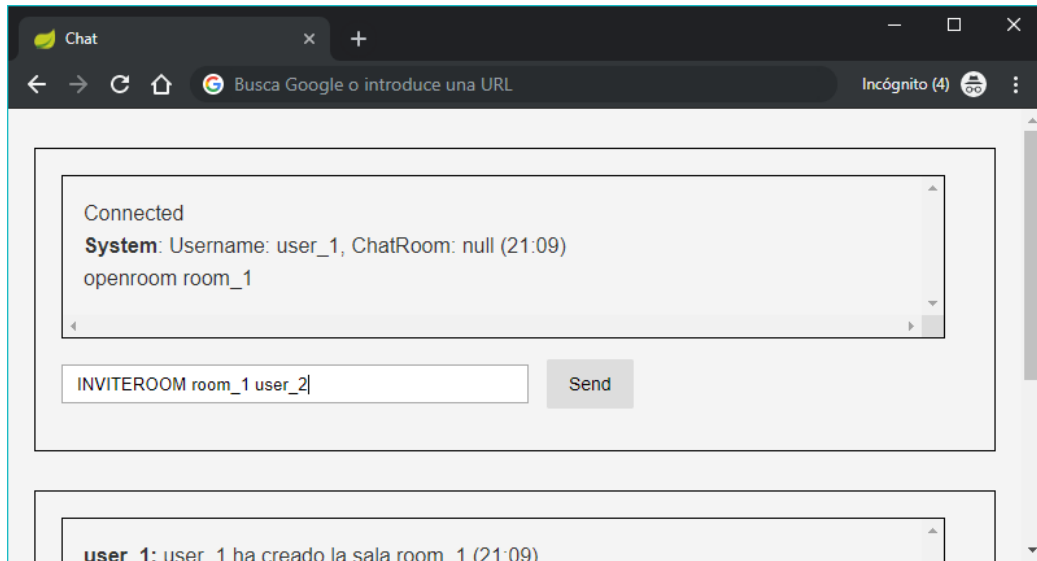
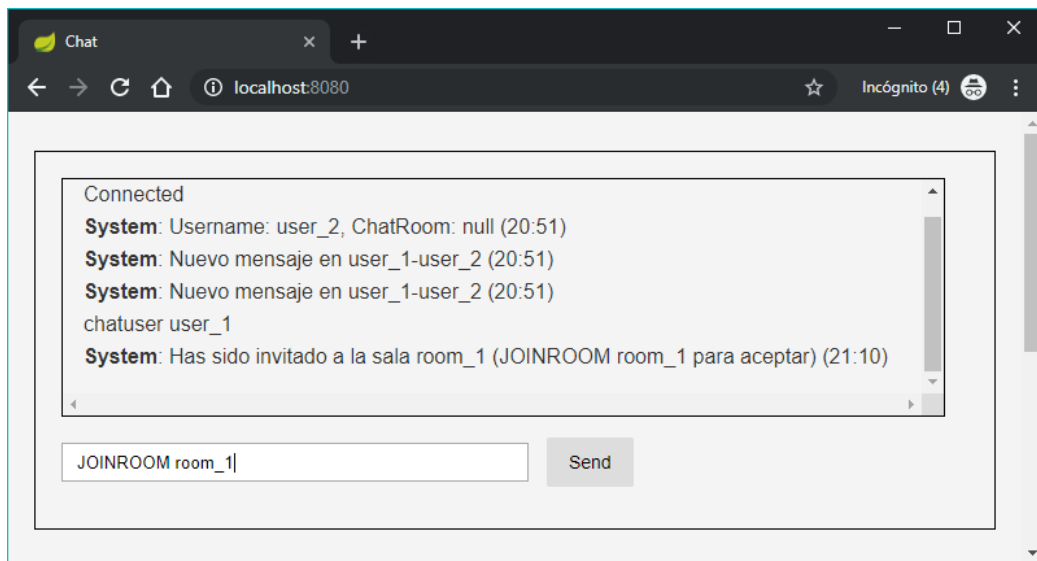
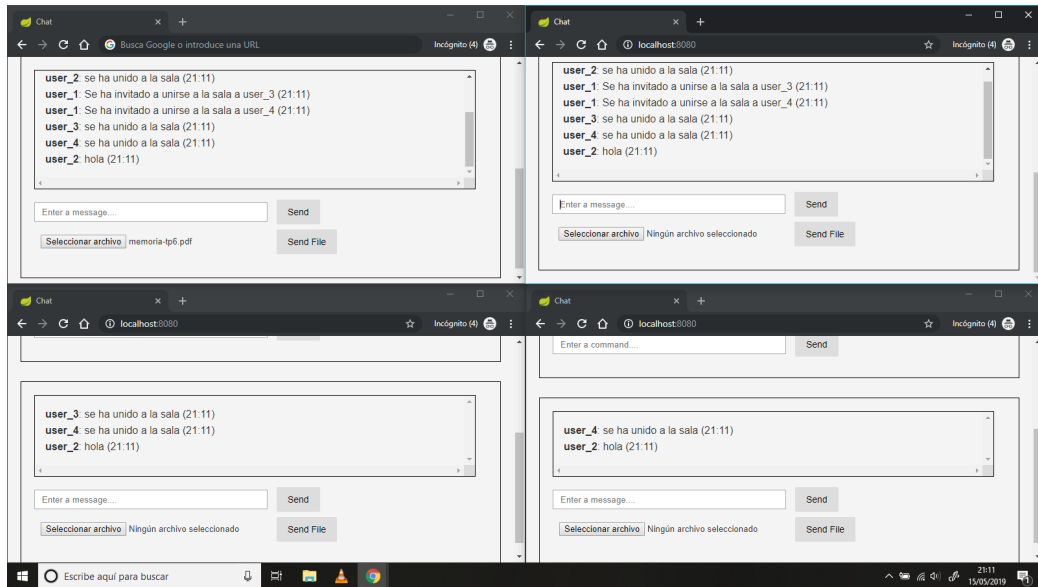
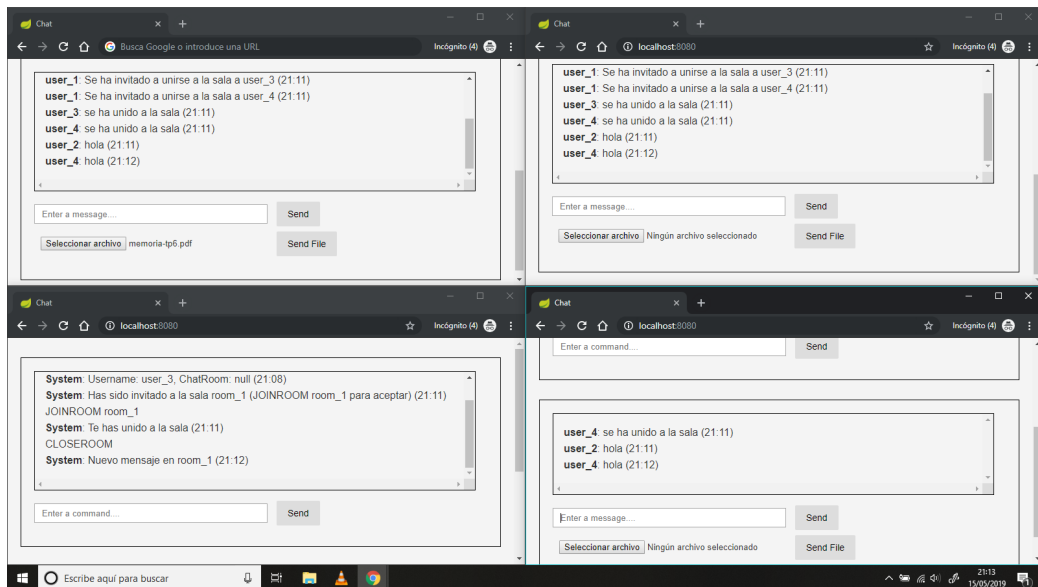
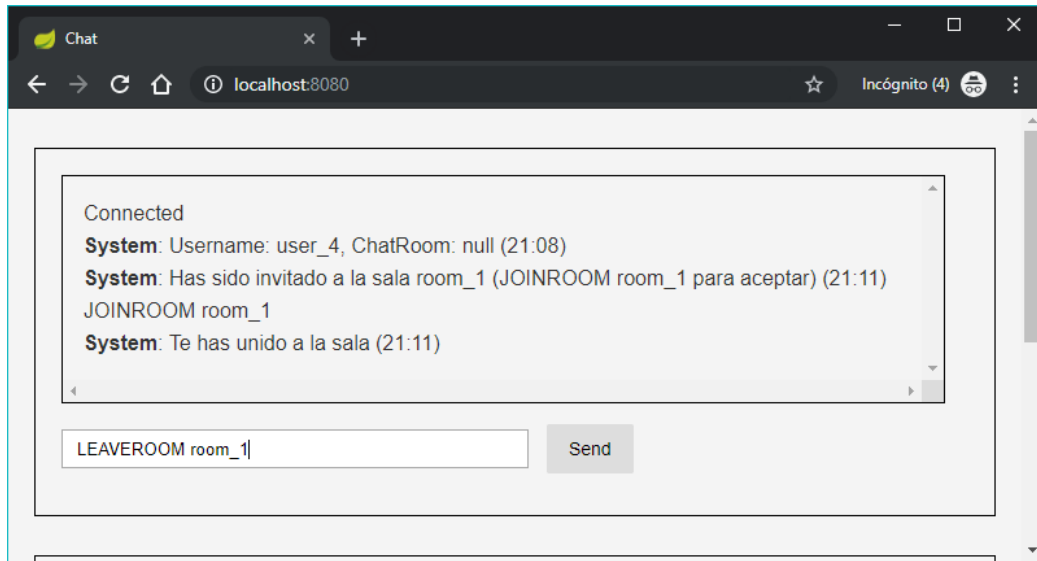
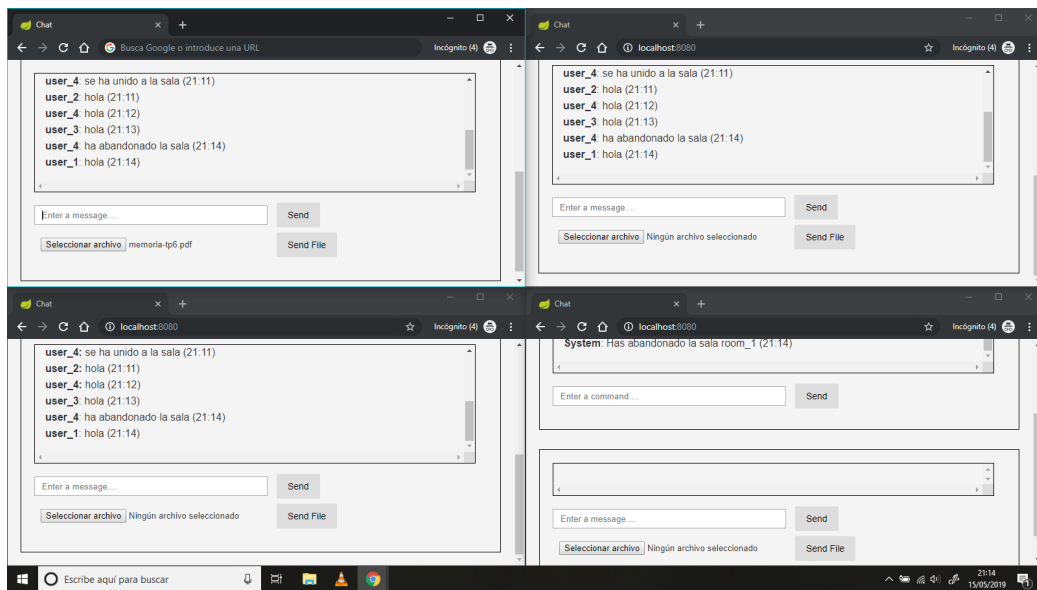


Figura 25: *user_1* crea la sala *room_1*

Figura 26: *user_1* invita al resto de usuariosFigura 27: *user_2* se une a la sala *user_2*

Figura 28: *user_2* envía mensaje a la salaFigura 29: *user_3* cierra la sala y empieza a recibir notificaciones

Figura 30: *user_4* abandona la salaFigura 31: *user_4* deja de recibir los mensajes de la sala *room_1*

6.4. Requisito 4

Este requisito dice lo siguiente: *Como superusuario quiero poder enviar mensajes a todos los usuarios suscritos para anuncios de interés y publicidad de nuevas características del sistema.*

Para ello existe el usuario *root* que es el administrador del sistema. Este usuario es el único que puede utilizar el comando **BROADCAST** *msg* para enviar un mensaje a todos los usuarios. Como ya se ha explicado, si el superusuario utiliza este comando se envía un mensaje a la centralita de *root* a la que las colas de todos los usuarios están suscritas.

Para mostrar el funcionamiento de este comando volvamos al caso de la sección anterior en el que teníamos 4 usuarios. Si el usuario *root* decide enviar un mensaje con el comando **BROADCAST** (Figura 32), los 4 usuarios recibirán este mensaje (Figura 33).

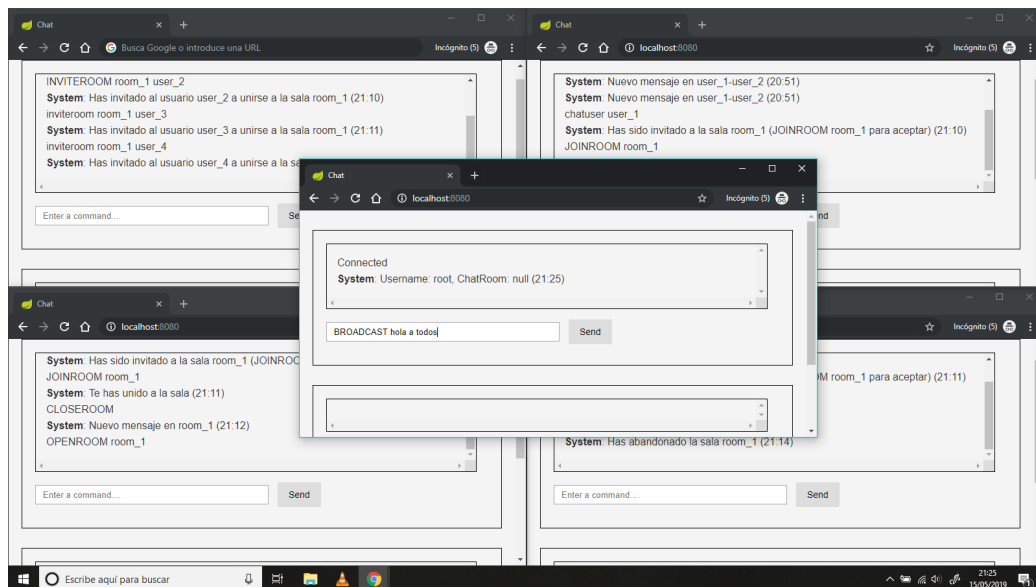


Figura 32: *root* envía mensaje broadcast

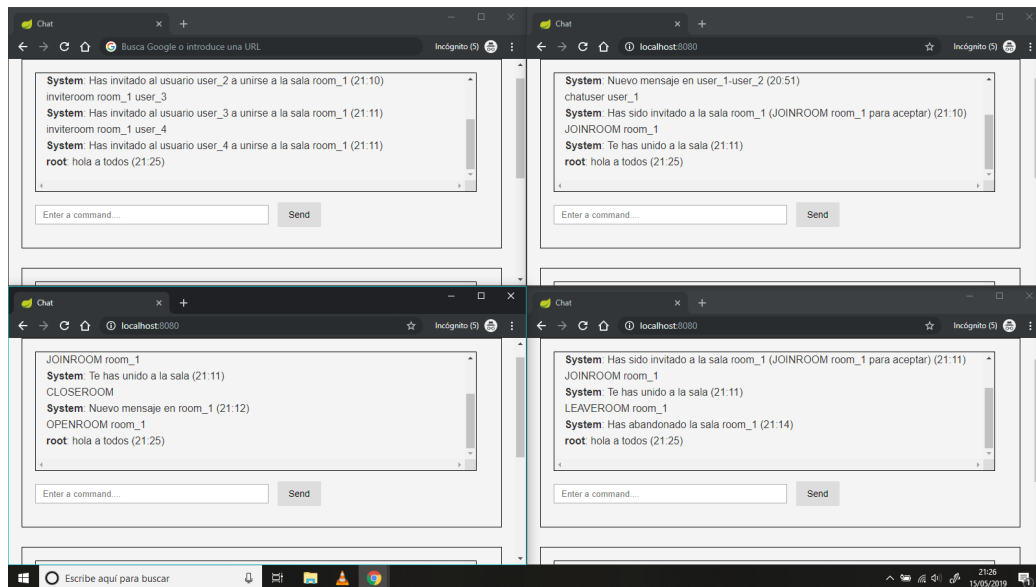


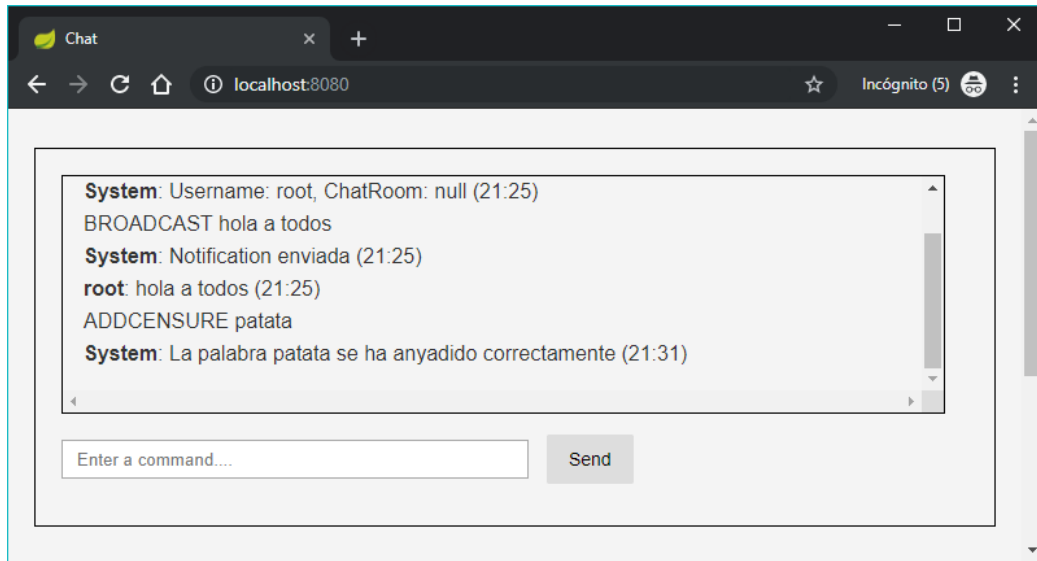
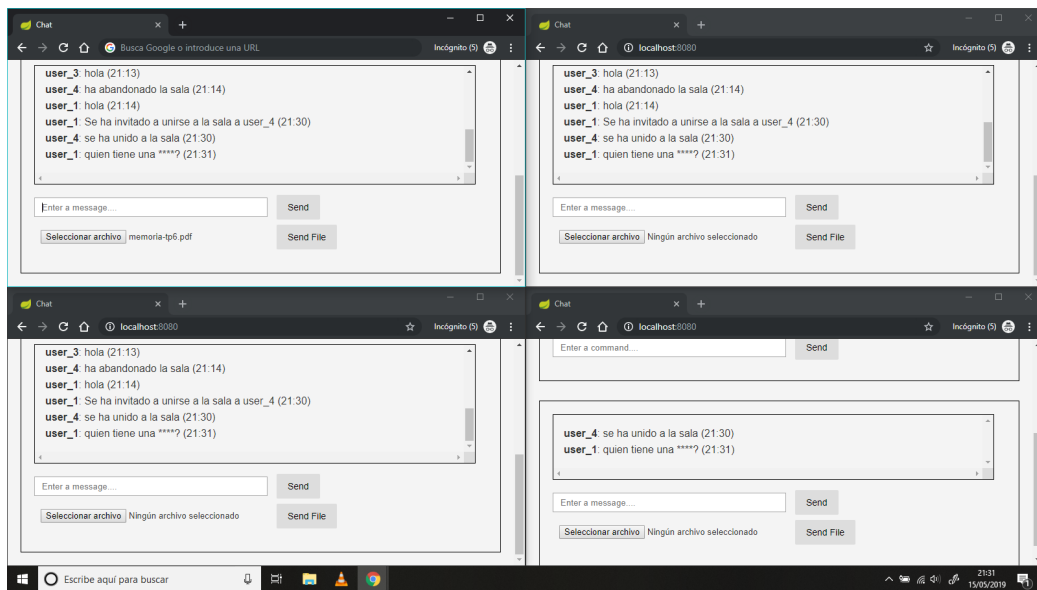
Figura 33: Todos los usuarios reciben el mensaje

6.5. Requisito 5

Este requisito dice lo siguiente: Como superusuario quiero poder censurar algunos mensajes para cumplir con los requisitos legales exigidos en regímenes represivos con las libertades civiles pero con mercados económicamente rentables.

Para ello existen los comandos **ADDCENSURE**, **REMOVECENSURE** y **GETCENSURE**, que sólo pueden ser usados por el superusuario (*root*) y permiten gestionar las palabras que se van a censurar, además cada vez que se envía un mensaje a una sala, los mensajes son filtrados para ver si contienen alguna de las palabras censuradas. Si es así se sustituye por “****”.

Para mostrar su funcionamiento vamos a suponer el caso en el que los 4 usuarios de ejemplos anteriores están en la misma sala. Si el superusuario censura la palabra “patata” (Figura 34) y un usuario decide enviar un mensaje con esta palabra, el mensaje aparecerá censurado (Figura 35). Sin embargo, si el superusuario decide que esa palabra ya no está censurada y la elimina de la lista (Figura 36), si un usuario envía un mensaje con esa palabra ya no será censurado (Figura 37).

Figura 34: *root* censura “patata”Figura 35: *user_1* envía un mensaje a la sala *room_1* que contiene la palabra “patata” y es censurado

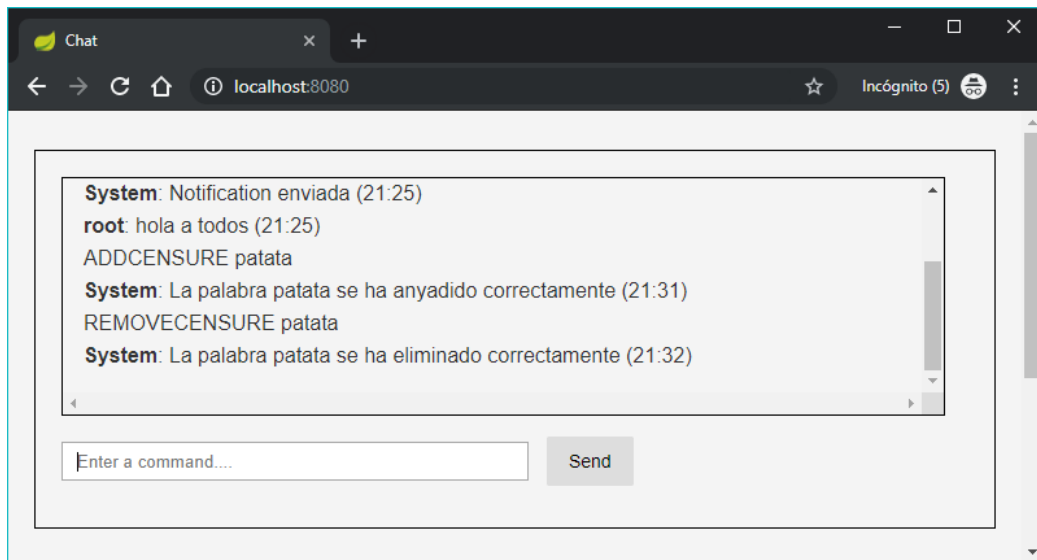


Figura 36: *root* elimina “patata” de la lista de palabras censuradas

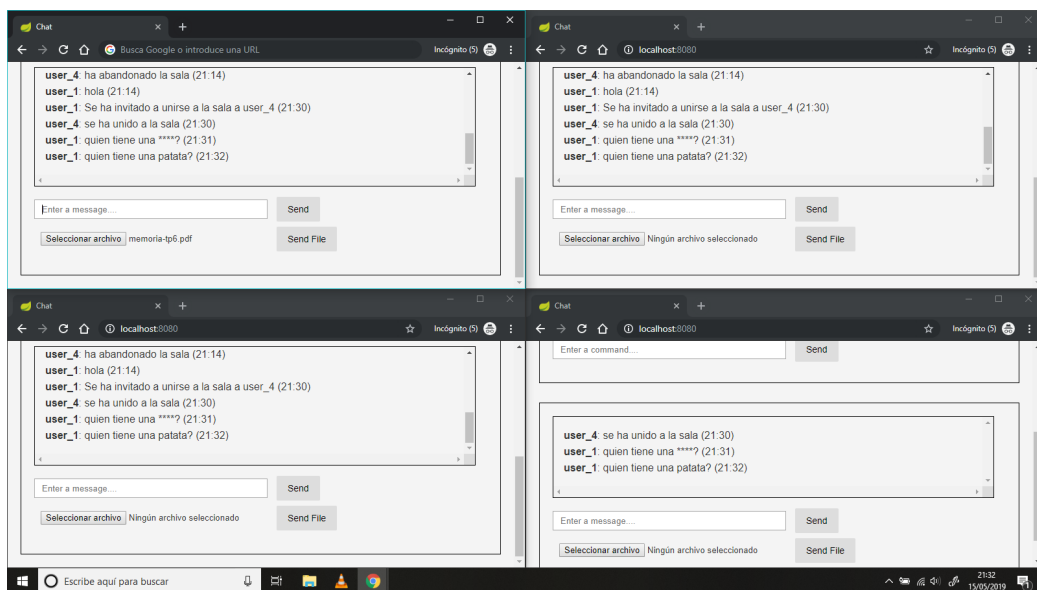


Figura 37: *user_1* envía un mensaje a la sala *room_1* que contiene la palabra “patata” y ya no es censurado

7. Conclusiones

Con el desarrollo de esta aplicación de chat se ha conseguido implementar un sistema distribuido y comprender cómo se comunican los diferentes nodos o máquinas de este sistema. También ha servido para repasar los diferentes diagramas aprendidos a lo largo de otras asignaturas.

Además, con este trabajo se ha logrado comprender cómo funcionan los sistemas de mensajería de cola de mensajes, como *Rabbit*, así como los servicios REST y la comunicación utilizando *web sockets*.

Otra parte importante que se ha aprendido en este trabajo es la creación de pruebas automáticas, que permiten realizar una evaluación completa del sistema.

Como la implementación de la interfaz no era importante, este trabajo ha permitido que nos centremos más en la parte de implementar un sistema distribuido junto con sus comunicaciones. Este trabajo también ha permitido conocer las ventajas de utilizar un framework como *Spring Boot* con la que se ahorra bastante tiempo.

Referencias

- [1] Repositorio github. https://github.com/LauraOliva/TMDAD_ProyectoMensajes. Accessed: 2018-05.
- [2] Rubén Béjar y Francisco J Lopez-Pellicer Pedro Álvarez. Apuntes de la asignatura TMDAD, 2019.
- [3] Spring boot, websockets application. <https://www.callicoder.com/spring-boot-websocket-chat-example/>. Accessed: 2018-04.
- [4] Spring boot, websockets. <https://www.nexmo.com/blog/2018/10/08/create-websocket-server-spring-boot-dr/>. Accessed: 2018-04.
- [5] Spring boot, ficheros. <https://www.callicoder.com/spring-boot-file-upload-download-jpa-hibernate-mysql-database-example/>. Accessed: 2018-04.
- [6] Spring boot, base de datos. <https://spring.io/guides/gs/accessing-data-mysql/>. Accessed: 2018-05.
- [7] Cloud amqp. <https://www.cloudamqp.com/>. Accessed: 2018-05.