Universidad Rey Juan Carlos

TITULACIÓN EN INGENIERÍA DEL SOFTWARE

Curso Académico 2017/2018

Trabajo Fin de Grado

Comparativa de tecnologías de servidor para servicios basados en websocket

Autor : Michel Maes Bermejo

Tutor : Micael Gallego Carrillo

Resumen

Aquí viene un resumen del proyecto. Ha de constar de tres o cuatro párrafos, donde se presente de manera clara y concisa de qué va el proyecto. Han de quedar respondidas las siguientes preguntas:

- ¿De qué va este proyecto? ¿Cuál es su objetivo principal?
- ¿Cómo se ha realizado? ¿Qué tecnologías están involucradas?
- ¿En qué contexto se ha realizado el proyecto? ¿Es un proyecto dentro de un marco general?

Lo mejor es escribir el resumen al final.

Índice general

Capítulo 1

Introducción y motivación

Hoy en día, un desarrollador de software tiene múltiples herramientas (entre lenguajes y librerías) para abordar cualquier proyecto que tenga entre manos.

Es una práctica común usar una tecnología concreta sobre la que sentimos predilección o las que creemos que pueden resolver mejor nuestro problema. En ocasiones, nos equivocamos en nuestra elección y descartamos opciones mucho más efectivas.

Este problema de desinformación puede abordarse mediante el estudio de las distintas tecnologías que proponen una solución al mismo, pero dado que el ámbito del desarrollo software es muy amplio, vamos a centrarnos en las tecnologías de servidor para servicios basados en WebSockets.

Estas tecnologías proporcionan una comunicación en tiempo real con clientes muy diversos (aplicaciones móviles, navegadores, otro servidores). Un ejemplo actual son los servicios de mensajería instantánea cómo WhatsApp o Telegram, cuyo crecimiento de usuarios se ha disparado en los últimos años. Hoy en día este tipo de aplicaciones tienen un impacto drástico en la vida diaria, siendo casi una herramienta imprescindible, por lo que prevenir una caída de servicio ante un alto número de clientes es fundamental.

La motivación de este proyecto surge de la necesidad de comprender mejor estas tecnologías y proporcionar argumentos sólidos que justifiquen el uso de una u otra, dependiendo de las necesidades de nuestro proyecto y de los recursos de los que dispongamos.

Para ello, tomaremos como punto de partida las tecnologías reactivas, que siguiendo el Manifiesto Reactivo[1] cuentan entre sus características:

- Tiempos de respuestas rápidos
- Tolerantes a fallos
- Adaptación a variaciones en la carga de trabajo
- Uso de mensajes asíncronos para la comunicación (no bloqueantes)

Para este proyecto, nos centraremos en Java, un lenguaje consolidado que cuenta con librerias y frameworks que nos ayudarán a abordar esta comparativa.

[1] `http://www.reactivemanifesto.org/`

Capítulo 2

Objetivos

El objetivo principal de este proyecto será realizar una comparativa entre distintas tecnologías que den solución a la comunicación en tiempo real mediante el uso de WebSockets. Dicha comparativa se realizará en base al rendimiento y el consumo de recursos de cada una de las tecnologías comparadas, ante diferentes niveles de carga y haciendo uso de un número variado de servidores.

Con este fin, se implementará un servidor de mensajería instantánea (que a partir de ahora denominaremos simplemente Chat) para cada tecnología y un cliente que se conectará a ese servidor simulando varios usuarios enviando mensajes que podrá medir el tiempo que tarda un mensaje desde que se envía hasta que se recibe.

Otro objetivo relevante del proyecto será su extensibilidad, de forma que cualquier desarrollador pueda implementar su aplicación de chat, sumarla a la comparativa y así contribuir al proyecto.

El proyecto base corresponde al realizado por el mismo autor de el proyecto que nos ocupa, en el que se realizó una comparativa entre las aplicaciones de Akka, Vert.x, SpringBoot y NodeJS, las cuales se compararón haciendo uso de una sola máquina. Este proyecto pretende ser una continuación y expansión del anterior, concretamente:

- Distribuir cada aplicación para que pueda ser lanzada en varias máquinas que formen un clúster.
- Actualizar las librerias a su última versión a fin de contar con las herramientas más recientes.

- Mejorar el cliente existente para que sea capaz de recoger métricas de una aplicación distribuida.

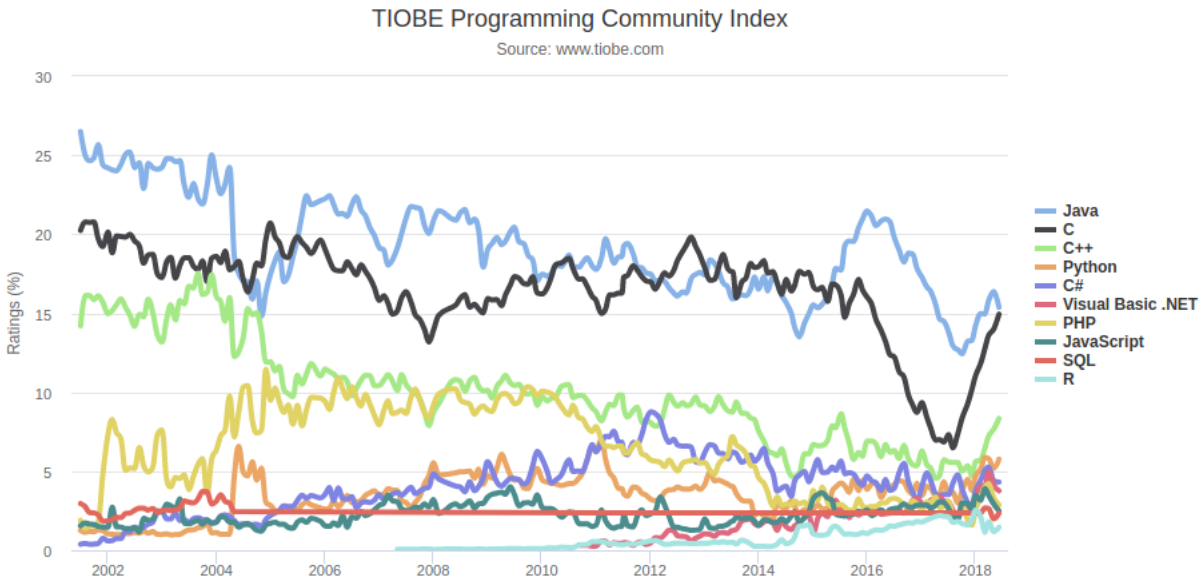
Las tecnologías que compararemos en este proyecto serán:

- Akka
- Vert.x
- SpringBoot + RabbitMQ
- SpringBoot + Hazelcast

Capítulo 3

Tecnologías, Herramientas y Metodologías

De la multitud de lenguajes de programación que existen válidos para afrontar el desarrollo de un servidor basado en WebSocket nuestra primera opción ha sido seleccionar Java, que dispone de múltiples librerías interesantes para abordar el problema además de ser uno de los lenguajes de programación más extendidos, populares y con una amplia comunidad, como demuestra el informe TIOBE.



3.1. Tecnologías

3.1.1. Websockets

RFC 6455[1] define WebSocket como un protocolo que proporciona un canal de comunicación bidireccional y full-dúplex sobre un único socket TCP. Aunque inicialmente estaba pensado para cualquier tipo de comunicaciones entre el navegador y el servidor web, puede usarse también para aplicaciones cliente/servidor.

Por otro lado, W3C se encarga de normalizar la API[2] de WebSocket. Define una interfaz para el navegador compuesta por 4 métodos que corresponden a manejadores o gestores (*handlers*) para cada evento.

Podemos ver un ejemplo de estos manejadores en el código mostrado a continuación (Javascript en el navegador).

```
var socket = new WebSocket("ws://example.com:9000/chat");
// Send new text
socket.send("Some text");
socket.onmessage = function(event) {
   var data = JSON.parse(event.data);
   // Use data
});
socket.onopen = function(e){ console.log("WS Opened")};
socket.onclose = function(e){ console.log("WS Closed")};
socket.onerror = function(e){ console.log(e)};
```

[1] https://tools.ietf.org/html/rfc6455
[2] https://www.w3.org/TR/2011/WD-websockets-20110929

3.1.2. Java

Java es un lenguaje de programación de propósito general, concurrente y orientado a objetos. Su sintaxis deriva en gran medida de C y C++. Uno de los principales atractivos de Java es su máquina virtual (JVM) que nos permite ejecutar nuestro código Java en cualquier dispositivo, independientemente de la arquitectura. Las tecnologías basadas en Java seleccionadas para la comparativa son explicadas a continuación.

3.1.3. Akka

Akka[3] es un toolkit para crear aplicaciones concurrentes y distribuidas. También se ejecuta sobre la JVM. Se puede utilizar con Java y Scala, lenguaje con el que está escrito y del que su implementación de los actores forma parte de la librería estándar desde la versión 2.10. Otras de sus características son:

- **Tolerancia a fallos**: Akka adopta el modelo de *let it crash* que ha resultado un gran éxito en la industria de la telecomunicación.
- **Transparencia de localización**: todo en Akka está diseñado para trabajar en un entorno distribuido: todas las comunicaciones son mediante paso de mensajes y todo es asíncrono

[3] `http://akka.io/`

- **Persistencia**: Los mensajes recibidos por el actor pueden conservarse y ser reproducidos al iniciar o reiniciar el actor, por lo que se puede conservar el estado de los actores después de un fallo o al migrarlos a otro nodo.

La versión utilizada de Akka durante este proyecto es la 2.5.

La aplicación de Akka hace uso de Play Framework[4] un framework web open source, que da soporte web a la aplicación y proporciona la comunicación mediante WebSockets.

Los conceptos básicos que debemos comprender de Akka son:

- **Actores**: Los actores son objetos que poseen un estado y un comportamiento. Se comunican entre ellos exclusivamente enviando mensajes que se encolan en el mailbox del actor de destino. Los actores se organizan jerárquicamente. Un actor encargado de realizar una tarea, puede dividir esa tarea en otras sub-tareas y enviárselas a unos actores hijos a los que supervisará.

- **Actor System**: Es el encargado de ejecutar, crear y borrar actores además de otros fines como la configuración o el logging. Varios actor systems con diferentes configuraciones puede coexistir en la misma JVM sin problemas, aunque al ser una estructura pesada que puede manejar de 1..N threads, se recomienda crear una por aplicación.

- **Actor Reference**: Es un objeto que representa al actor en el exterior. Estos objetos pueden enviarse sin ninguna restricción y permiten enviar mensajes al actor con total transparencia, sin necesidad de actualizar las referencias a pesar de enviarse a otros hosts. Además evitan que desde el exterior pueda conocerse el estado del actor a no ser que este lo publique.

- **Actor Path**: Como los actores son creados en una estricta estructura jerárquica, existe una única secuencia de nombres de actores dados siguiendo recursivamente los links entre actores padres e hijos hasta el actorSystem. Esta secuencia similar a las rutas de un sistema de ficheros, por ello es conocida como actor Path.

La diferencia entre un ActorPath y una ActorReference es que el segundo tiene el mismo ciclo de vida que el actor. Si el actor se destruye su ActorReference también, sin embargo un ActorPath puede existir perfectamente a pesar de que no exista el actor.

[4] `https://www.playframework.com/`

3.1.4. Hazelcast

Hazelcast [5] es un DataGrid de Java, una herramienta escalable para la distribución de datos. Entre algunos de sus usos principales, se encuentran:

- Cacheo de datos de forma distribuida.
- Ejecución paralela.
- Mensajería distribuida.
- Escalado dinámico.
- Transacciones sobre los datos.
- Querys sobre los datos.

Para este proyecto nos interesa la mensajeria distribuida, que funciona mediante un patrón publicar-subscribir sencillo de distribuir entre diferentes nodos.

3.1.5. Vert.x

Vert.x[6] es otro toolkit de Java que permite construir aplicaciones reactivas. Se autodenomina dirigido por eventos y no bloqueante, está inspirado en Node.js. La versión utilizada en el proyecto es la 3.5.

Los conceptos básicos que debemos comprender de Vert.x son:

[5] https://hazelcast.com/
[6] http://vertx.io/

- **Verticle**[7]: modelo de concurrencia que propone Vertx. Un Verticle es una clase que se comporta como un actor[8], cuyo comportamiento está orientado a enviar/recibir mensajes. Para facilitar el desarrollo, Vertx asegura que el código de un verticle nunca va a ser ejecutado por más de un thread a la vez.

- **EventBus**: es uno de sus principales recursos que le da su carácter reactivo. Consiste en un bus transversal a la aplicación que permite la comunicación entre los verticles de distintas formas[9]:

- **Publicar-Subscribir**: Diversos verticles se subscriben a un determinado topic proporcionando un handler que opere con la respuesta. Tras esto, basta con publicar un mensaje bajo ese topic para que todos los componentes subscritos lo reciban.

- **Punto a punto**: Al igual que el anterior, envía un mensaje bajo un topic, pero en este caso, solo a uno de los subscriptores, elegido mediante un algoritmo de round-robin no estricto.

- **Petición-Respuesta**: Similar al anterior, con la única diferencia que se proporciona un handler para una posible respuesta.

- **Context**[10]: se encarga de controlar un ámbito concreto de la aplicación, además del orden en el que los callbacks/handlers son ejecutados. Vertx dispone de 3 tipos diferentes de contexts:

    - Event-loop: ejecuta los handlers de forma que un mismo handler es ejecutado únicamente en un Thread y este no debe ser bloqueante de ninguna manera (uso de herramientas de bloqueo condicional, llamadas a bases de datos, ejecuciones del sistema largas, etc?). Este modelo no es dependiente la sincronización y dota a Vertx, junto al EventBus de su reactividad, además de su carácter no bloqueante. Es el context usado por defecto.

[7] `http://vertx.io/docs/vertx-core/java/#verticles`

[8] `https://en.wikipedia.org/wiki/Actor_model`

[9] `http://vertx.io/docs/apidocs/io/vertx/core/eventbus/EventBus.html`

[10] `https://github.com/vietj/vertx-materials/blob/master/src/main/asciidoc/Demystifying_the_event_loop.adoc`

- Worker: contexto ligado a los verticles, que siguen asegurando que se ejecutan en un solo Thread, pero permiten su bloqueo.
- Multi-Thread Worker: Permite la ejecución de un verticle en diferentes threads, de forma que pueda realizar las tareas de forma concurrente, delegando en el desarrollador la responsabilidad de asegurar la concurrencia y sincronización.
- Además de los recursos mencionados, cuenta con una extensa API que abarca desde múltiples herramientas de testing hasta servidores y clientes de TCP/SSL, HTTP/HTTPS y WebSockets, cobrando estos últimos especial importancia de cara al desarrollo de la aplicación.

Vert.x utiliza Hazelcast por defecto para distribuir su bus de eventos y otras funcionalidades distribuidas, aunque puede usarse también con Ignite[11], Infinispan[12] o Zookeeper[13]. Para este proyecto usaremos la configuración por defecto con Hazelcast.

3.1.6. SpringBoot

Spring Boot[14] comprende un módulo de Spring[15] (un framework para el desarrollo de aplicaciones web) que provee de todo lo necesario para crear una aplicación con un mínimo de configuración lista para lanzar. Spring Boot proporciona:

- Una experiencia de iniciación muy rápida
- Prototipos extensibles para la mayoría de problemas que podamos tener
- Características no funcionales comunes a la mayoría de proyectos (servidores integrados, seguridad, métricas, comprobaciones de estado, configuración externalizada).

[11] `https://ignite.apache.org/`
[12] `http://infinispan.org/`
[13] `http://zookeeper.apache.org/`
[14] `http://projects.spring.io/spring-boot/`
[15] `https://spring.io/`

Además, cuenta con el Sistema de Inversión de Control de Spring[16][17], que permite la configuración de los componentes de la aplicación, mientras que la administración del ciclo de vida de los objetos se lleva a cabo a través de la inyección de dependencias[18] (que a su vez es una forma de inversión de control).

La versión utilizada de Spring para este proyecto es la 1.4.3

3.1.7. RabbitMQ

RabbitMQ[19] es un software de mensajería de código abierto escrito en Erlang[20] que implementa el protocolo de cola de mensajes avanzados (AMQP[21]), además de otros protocolos que ha ido añadiendo cómo STOMP[22] y MQTT[23]. Para este proyecto usaremos la versión 3.5.7.

Entre las caracteristicas mas relevantes que encontramos en esta tecnología, que comparte con otras tecnologías de colas de mensajes, encontramos:

- **Garantía de entrega y orden**: los mensajes se consumen en el mismo orden que se llegaron a la cola y son consumidos una única vez.
- **Redundancia**: Las colas mantienen los mensajes hasta que son procesados por completo.
- **Desacoplamiento**: al actuar cómo un middleware, siendo una capa intermedia de comunicación entre procesos, aportan la flexibilidad en la definición de arquitectura de cada uno de ellos de manera separada, siempre que se mantenga una interfaz común.

[16] https://en.wikipedia.org/wiki/Inversion_of_control
[17] https://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html
[18] https://en.wikipedia.org/wiki/Dependency_injection
[19] https://www.rabbitmq.com/
[20] https://www.erlang.org/
[21] https://es.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol
[22] https://en.wikipedia.org/wiki/Streaming_Text_Oriented_Messaging_Protocol
[23] https://en.wikipedia.org/wiki/MQTT

- **Escalabilidad**: con más unidades de procesamiento, las colas balancean su respectiva carga.

Al contrario que Vert.x o Akka, RabbitMQ es un servicio externo (pudiendo estar o no en la misma máquina dónde se ejecute nuestra aplicación). Para hacer uso de este middleware será necesario un cliente que interactúe con él.

Los conceptos principales para entender RabbitMQ son:

- **Exchange**: punto de entrada de los mensajes. Pueden ser:
  - *Direct* entrega un mensaje a una sola cola,
  - *Fanout* entrega copias del mensaje a todas las colas
  - *Topic*: entrega copias del mensaje sólo a algunas colas.
- **Queue**: punto de lectura de los mensajes. Pueden ser durable o persistentes (si almacenan los mensajes para sobrevivir a un reinicio de RabbitMQ). También pueden ser exclusivas, si sólo un consumidor puede estar conectado a la vez.
- **Bindings**: definen cómo llegar de un *Exchange* a las *Queue* asociadas.
- **Routing key o topic**: filtro asociado a un Binging que permite seleccionar sólo algunos mensajes para dicho binding.
- **Productor**: programa que escribe en un Exchange
- **Consumidor**: programa que escucha en una Queue

3.1.8. HA PROXY

HAProxy es una solución gratuita, muy rápida y confiable que ofrece alta disponibilidad, balanceo de carga y proxying para aplicaciones TCP y HTTP. Es especialmente adecuado para sitios web de mucho tráfico. Está escrito en C y tiene la reputación de ser rápido y eficiente en términos de uso del procesador y consumo de memoria. Con el paso de los años, se ha convertido en el estándar de facto del balanceador de carga opensource, ahora se incluye con la mayoría de las distribuciones de Linux, y a menudo se implementa de manera predeterminada en las plataformas en la nube.

3.1.9. Angular

Angular[24] es un framework de JavaScript (aunque comúnmente se utiliza con Typescript[25], un superconjunto de Javascript) de código abierto desarrollado por Google. Nos permite desarrollar SPAs (Single Page Applications), que siguiendo el MVC (modelo-vista-controlador), facilitan la presentación y manipulación de los datos en el lado cliente (frontend), reduciendo la carga lógica del lado servidor (backend). La versión utilizada para este proyecto es la 5.2.

Entre sus características, destacamos:

- La extensión del html mediante etiquetas y sintaxis propia.
- Inyección de dependencias
- Una numerosa comunidad y una extensa documentación

Utilizaremos Angular para ofrecer un cliente web en el que mostrar los resultados del experimento.

[24] `https://angular.io/`
[25] `https://www.typescriptlang.org/`

3.2. Herramientas

3.2.1. Control de versiones: Git

Git[26] es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente.

Para el desarrollo de este proyecto hemos usado GitHub[27], una plataforma de desarrollo colaborativa para alojar proyectos Git.

A pesar de su integración con diversos entornos de desarrollo, se ha optado por su versión de línea de comandos.

3.2.2. Gestores de dependencias

Debido a la pluralidad de tecnologías, hemos utilizado distintos gestores de dependencias:

3.2.2.1. Maven

Maven[28] es una herramienta de software para la gestión y construcción de proyectos Java creada por Jason van Zyl. Hace uso de un POM (Project Object Model), un archivo XML que describe las dependencias y permite añadir opciones de ejecución, test y despliegamiento de la aplicación.

Se ha utilizado para configurar los proyectos en Vert.x y Spring Boot.

[26] `https://git-scm.com/`
[27] `https://github.com`
[28] `https://maven.apache.org/`

3.2.2.2. SBT

SBT[29] es una herramienta de software para construcción de proyectos en Scala y estándar para contruir aplicaciones en Play Framework, similar a Maven o Ant (propios de Java). Entre sus características, permite el uso conjunto de Java y Scala en el mismo proyecto. Su archivo de configuración es un.stb, que dispone dispone de sintaxis propia.

Se ha utilizado para configurar el proyecto de Akka.

3.2.3. AWS

Amazon Web Services[30] (AWS) es una plataforma de servicios de nube que ofrece potencia de cómputo, almacenamiento de bases de datos, entrega de contenido y otras funcionalidades.

Concretamente se ha utilizado su servicio EC2[31], que nos permite lanzar instancias que contengan nuetras aplicaciones en la nube. Para este proyecto se ha hecho uso de la capa gratuita.

Para hacer uso de esta plataforma, se ha utilizado su interfaz mediante línea de comandos[32].

3.2.4. Entornos de desarrollo

3.2.4.1. IntelliJ

[29] http://www.scala-sbt.org/
[30] https://aws.amazon.com
[31] https://aws.amazon.com/es/ec2
[32] https://aws.amazon.com/es/cli

IntelliJ[33]es un IDE para Java desarollado por JetBrains ideado para mejorar la productividad del programador. Entre sus características incluye:

- Soporte para los lenguajes basados en la JVM (Java, Scala, Groovy y Kotlin)
- Soporte para diferentes frameworks basados en estos lenguajes (Spring, Play, JavaEE...)
- Control de versiones
- Asistencia al escribir código y autocompletar
- Soporte para programar en web (HTML, CSS y Javascript)

Se ha utilizado este IDE para desarrollar los distintos servidores de chat, ya que todos ellos están basados en Java.

3.2.4.2. Atom

Atom[34] es un editor de texto sencillo, ligero y extensible creado por Github. Cuenta con una gran librería de paquetes aportados por la comunidad para facilitar el desarrollo software. Por defecto, no cuenta con ningún tipo de compilador o intérprete.

Se ha utilizado de forma conjunta con IntelliJ para desarrollar la aplicación de pruebas, ya que cuenta con paquetes que nos ayudan a desarrollar aplicaciones en Angular y soporte para Typescript.

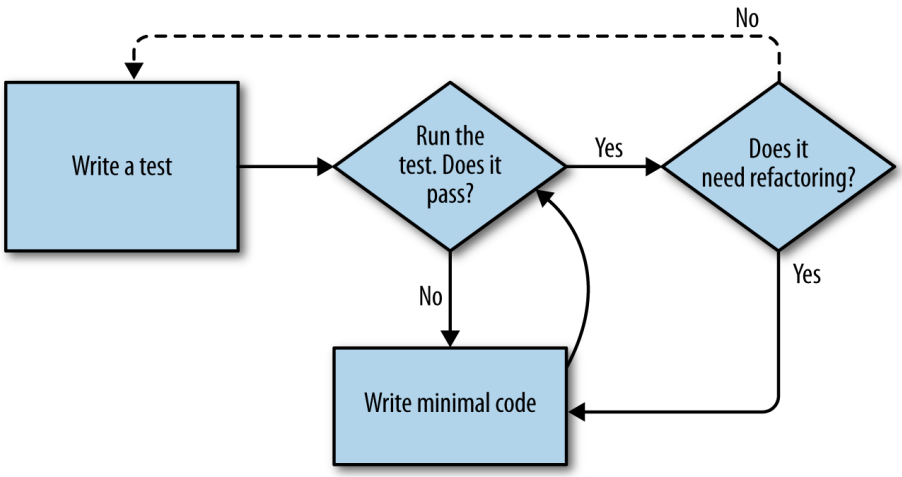
3.3. Metodologías

El modelo de desarrollo de este proyecto se ha llevado a cabo a través de TDD[35](Test-driven Development, o en español, desarrollo guiado por pruebas), una práctica de Ingeniería del Software cuya principal idea es hacer que los requistos sean traducidos a pruebas.

[33] https://www.jetbrains.com/idea/
[34] https://atom.io/
[35] https://es.wikipedia.org/wiki/Desarrollo_guiado_por_pruebas

Las razones que han llevado a utilizar un ciclo de desarrollo conducido por pruebas son:

- La naturaleza intrínseca del proyecto, distintas aplicaciones cuyo funcionamiento debe ser el mismo y por tanto comparten requisitos.
- La herencia de un proyecto, que proporcionaba dichas pruebas de integración necesarias para validar cualquier aplicación.



Por lo tanto, para cada aplicación que implementásemos, debíamos desarrollarla de acuerdo a las pruebas, de forma que una vez las pasasen, solo debíamos refactorizar la aplicación para mejorar su rendimiento y mantenibilidad.

Capítulo 4

Descripción informática

En este apartado se abordará la construcción del proyecto. Todo el proyecto (que incluye tanto las aplicaciones de chat como el cliente de pruebas, pueden encontrarse como repositorios en la organización de GitHub: *TFG-DistributedWebChat*[1].

El proyecto realizado consta de 3 aplicaciones de chat y un cliente de pruebas. Las aplicaciones construidas y que entran a formar parte de la comparativa son:

- Akka
- Vert.x
- SpringBoot + RabbitMQ

La comparativa tomará en cuenta la escalabilidad horizontal, por lo que todas las aplicaciones se desarrollaran para funcionar en 2 o más máquinas

4.1. Requisitos

Como se ha mencionado anteriormente, este proyecto es la continuación de uno anterior, del que se ha heredado un cliente de chat que funciona como prueba de integración. Los requisitos, por lo tanto, quedan condicionados al funcionamiento de dicho cliente. Cada aplicación se construirá siguiendo los mismos requisitos.

Distinguiremos entre requisitos funcionales y no funcionales:

[1] `https://github.com/TFG-DistributedWebChat`

4.1.1. Requisitos funcionales

Los requisitos funcionales fueron detallados como documentación y publicados como una página en una wiki de GitHub para que cualquier desarrollador pudiera incluir su propia aplicación. Su versión en inglés puede encontrarse en la documentación del proyecto en GitHub [2], mientras que su versión en español se detalla a continuación.

Requisitos básicos

La aplicación en cuestión debe poder soportar un chat en el que varios usuarios puedan comunicarse entre si.

Requiere lanzar la aplicación como un servidor que escuche de un puerto concreto y ofrecer una conexión WebSocket sobre la dirección */chat*.

Primera conexión

El cliente, al establecer la conexión enviará sus datos en un string, que podrá formatearse a JSON y tiene la siguiente estructura:

```
{
   "name": "MyName",
   "chat": "MyRoom"
}
```

La aplicación debe almacenar estos datos junto a la conexión WebSocket, de forma que queden registrados.

Recepción y reenvío de mensajes

Una vez se ha establecido la conexión y se ha mandado el mensaje de inicialización, el cliente enviará mensajes a la aplicación, de nuevo como un String, que se podrá formatear a un JSON con la siguiente estructura:

```
{
   "name": "MyName",
   "chat": "MyRoom",
   "message":"MyMessage"
}
```

Este mensaje debe ser reenviado por la aplicación a todos los usuarios cuya sala de chat sea

[2] https://github.com/TFG-DistributedWebChat/DistributedWebChatClient/wiki/Requeriments

la misma que la del mensaje[3].

Desconexión

La aplicación debe gestionar la desconexión de usuarios, de forma que cuando un usuario se desconecta, este debe eliminarse de la aplicación para que no se le reenvíen mensajes.

Opcionales

Aunque las pruebas que se realizan no lo requieren, para añadirle dificultad, la aplicación puede impedir que dos usuarios con el mismo nombre puedan conectarse (independientemente del chat al que pertenezcan). En caso de que ya exista el usuario debería enviar un mensaje de vuelta al cliente tal y cómo se muestra a continuación:

```
{
   "type": "system",
   "message": "A user with that name already exists"
}
```

Además, y de cara a probar rápidamente el correcto funcionamiento más básico de la aplicación, puede ofrecerse un cliente http que permita realizar la conexión desde el navegador.

4.1.2. Requisitos no funcionales

Dado el carácter comparativo que posee el proyecto, nos centraremos en los requisitos de calidad de ejecución, a fin de optimizar lo máximo posible cada aplicación. Los requisitos no funcionales más relevantes en el proyecto serán:

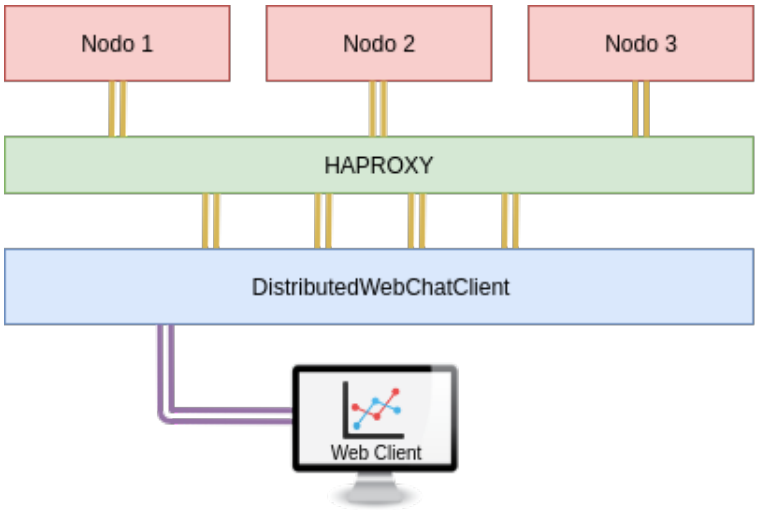
- **Latencia**: Las aplicaciones deben ofrecer un tiempo de respuesta lo más bajo posible dentro de las características de la tecnología en la que se base.
- **Consumo de recursos**: Las aplicaciones deben hacer un uso responsable de los recursos del sistema (como son la memoria o el uso del procesador).
- **Escalabilidad**: en nuestro caso, será escalabilidad vertical, que buscará que nuestras aplicaciones no vean degradada su calidad (en este caso una baja latencia y consumo de recursos) ante grandes cargas de trabajo.

[3]No debe confundirse un mensaje de chat con un mensaje de conexión, la forma de diferenciarlos es por la existencia o no de la clave *message* en el JSON.

- **Concurrencia**: Las aplicaciones tienen que estar libres de interbloqueos y esperas innecesarias. Dada la naturaleza de la mayoría de tecnologías (reactivas y no bloqueantes), este requisito es fácilmente satisfacible.

4.2. Diseño e Implementación

El proyecto seguirá la arquitectura planteada en la siguiente figura:



Al iniciar el experimento (una vez desplegados los nodos de la aplicación en clúster), el cliente abrirá varias conexiones WebSocket con el servidor, simulando varios usuarios, enviando mensajes a través de ellas y recogiendo diversas métricas. Estas métricas serán enviadas a través de WebSocket a una aplicación web en el navegador, que mostrará dichas métricas mediante el uso de gráficas en tiempo real. Las métricas podrán ser descargadas desde el navegador en formato JSON.

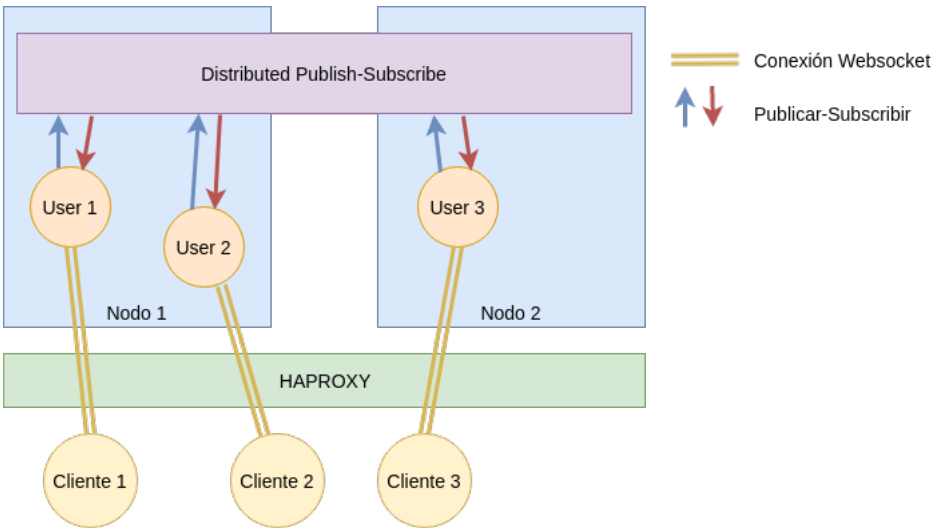
Los nodos del diagrama deberá poder ser cualquiera de las aplicaciones participantes en la comparativa (cuentan con los mismos requisitos), de forma que el cliente pueda interactuar con ellos independientemente de la implementación que posean.

A continuación, se expondrá el diseño e implementación de cada aplicación construida, así como un acceso a su código fuente. La infraestructura necesaria para desplegar cualquier aplicación esta definida en el apéndice A de este documento.

4.2.1. Akka

El código de esta aplicación se puede encontrar en un repositorio en GitHub de la organización antes mencionada bajo en nombre de *Akka-DistributedWebChat*[4]

Diseño y arquitectura



La aplicación de Akka, tal y cómo vemos en la figura hace uso del patron Publicar-Subscribir[5], permitiendo que el actor *User* sea capaz de interactura con otros de su mismo tipo incluso aunque se encuentren en otra máquina distinta bajo un topic que será en nombre de su sala de chat. Este actor es capaz de:

- Publicar nuevos mensajes que lleguen del cliente bajo un topic al resto de usuarios de la sala.
- Recibir (mediante subscripción) mensajes de un topic y enviarselos al cliente.

Funcionamiento

- **Conexión**: Cuando un cliente abre la conexión, Play ejecuta el siguiente método:

```
// /app/controllers/Application.java
public WebSocket<String> socket() {
    return WebSocket.withActor(User::props);
}
```

[4] https://github.com/TFG-DistributedWebChat/Akka-DistributedWebChat
[5] https://en.wikipedia.org/wiki/Publish-subscribe_pattern

La llamada al método estático props devuelve un nuevo actor User al que asigna la conexión WebSocket establecida que representa al cliente, tal y como se muestra en el código a continuación:

```
// /app/actors/User.java
public static Props props(ActorRef out) {
    return Props.create(User.class, out);
}
```

Al crearse el actor, también se le asigna una referencia a un mediador, que gestionará nuestras peticiones de publicar y subscribirnos a mensajes.

```
// /app/actors/User.java
this.mediator = DistributedPubSub
    .get(getContext().system())
    .mediator();
```

Una vez creado el actor, es capaz de recibir mensajes y estos serán tratados mediante las reglas que hayamos definido en su método *createReceive()*. Este método diferenciará entre mensajes que lleguen de la clase *String* (llegan del cliente) o de la clase *Message* (llegan del mediador.)

```
// /app/actors/User.java
@Override
public Receive createReceive() {
    return receiveBuilder()
    .match(String.class, message -> { /* FROM CLIENT */ })
    .match(Message.class, message -> { /* FROM OTHER USERS */ })
}
```

El primer mensaje del cliente (el de conexión) provoca la subscripción de dicho usuario a la sala de chat que ha definido:

```
// /app/actors/User.java
this.mediator.tell(
    new DistributedPubSubMediator
        .Subscribe(this.chatName, getSelf()),
    getSelf()
);
```

De forma previa a esta subscripción, se válida que el nombre del usuario no está repetido. De ser así, el actor se *suicida* (mandandose una *PoisonPill* a sí mismo) y cierra la conexión con el cliente:

```
// /app/actors/User.java
self().tell(PoisonPill.getInstance(), self());
```

- **Re-envío de mensajes**: Cómo hemos definido antes, la entidad *User* recibe dos tipos de mensajes:

  - Los mensajes del cliente (*String*) el cual llega através del Websocket. Este se publica haciendo uso del mediador para distribuirlo al resto de usuarios en la sala de chat:

    ```
    // /app/actors/User.java
    mediator.tell(
       new DistributedPubSubMediator
          .Publish(this.chatName, message),
       getSelf()
    );
    ```

  - Los mensajes de otros usuarios (*Message)* son directamente re-enviados al cliente por Websocket (usando el actor que representa esta conexión):

    ```
    // /app/actors/User.java
    out.tell(message.getJson().toString(), self());
    ```

- **Desconexión**: Cuando el usuario cierra la conexión WebSocket, se ejecuta el método postStop() del User correspondiente, que se encarga de dar de baja al usuario de su sala de chat.

```
// /app/actors/User.java
mediator.tell(
   new DistributedPubSubMediator
      .Unsubscribe(this.chatName, getSelf()),
   getSelf()
);
```

Despliegue en clúster

La entidad mediadora (*mediator*) es la que nos proporciona la capacidad de comunicarnos con usuarios de otros nodos situados en otras máquinas. Akka junto al framework Play nos abstrae de la lógica necesaria para esta comunicación, pero debemos declarar uno o varios *seed-nodes* en el archivo */conf/application.conf* para crear nuestro clúster. Estos *seed-nodes* ó nodos semilla en español identifican a una aplicación ya lanzada a la que una nueva aplicación puede unirse para formar un clúster.

```
// /conf/application.conf
cluster {
   seed-nodes = [
      "akka.tcp://application@127.0.0.1:8000"
   ]
}
```

El uso de *seed-nodes* nos garantiza que al arrancar una nueva máquina con un nuevo nodo no se formarán islas en el clúster (nodos sin ninguna conexión con el resto).

Para desplegar 2 ó más nodos en AWS basta con lanzar una máquina (sin necesidad de declarar *seed-nodes*) y obtener su dirección IP (privada). Los siguientes nodos deberán contener esta dirección entre sus *seed-nodes* o declararlo como semilla al lanzarlo.

```
./webchat-1.0/bin/webchat \
   -Dakka.cluster.seed-nodes.0=akka.tcp://application@${SEED}:8000
```

Para facilitar el desplegado del clúster se facilita un script en python que automatiza todos los pasos necesarios. Para más detalles, consultar el fichero README.md[6] del repositorio de Akka.

Problemas en el desarrollo

El mayor desafío era comprender cómo hacer evolucionar una aplicación monolítica a una distribuida. A partir de la documentación fué sencillo probar la aplicación haciendo uso de dos nodos en local, pero para desplegar estos nodos en AWS era necesario definir y abrir los puertos necesarios para que las máquinas pudieran comunicarse. Este despliegue require un gran número de pasos que dan lugar fácilmente a errores. La solución, tanto para Akka cómo

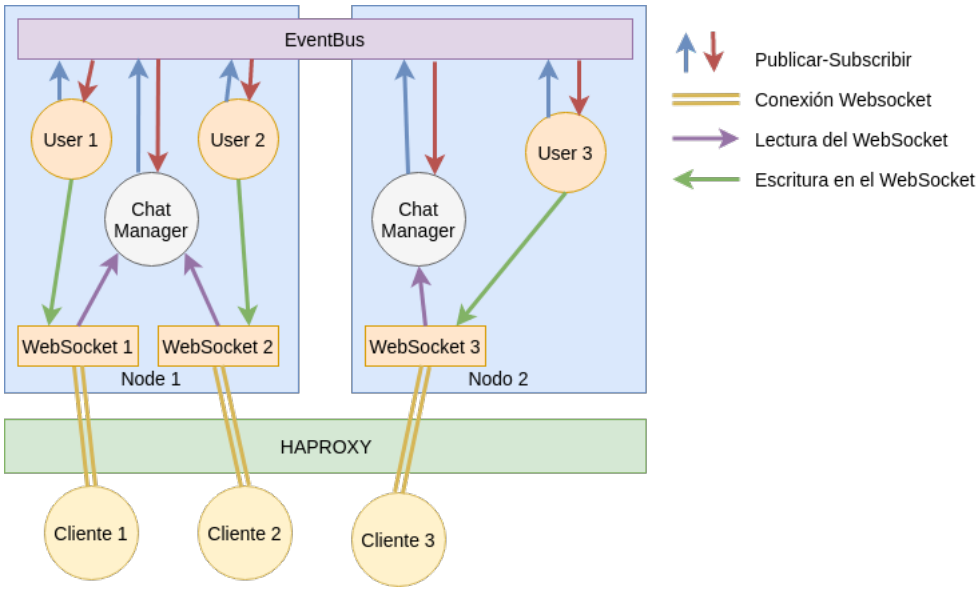
[6] `https://github.com/TFG-DistributedWebChat/Akka-DistributedWebChat/blob/master/README.md`

para el resto de aplicaciones a este problema ser resolvió automatizando este despliegue, tal y cómo se ha comentado anteriormente.

4.2.2. Vertx

El código de esta aplicación se puede encontrar en un repositorio en GitHub de la organización antes mencionada bajo en nombre de *Vertx-DistributedWebChat*[7]

Diseño y arquitectura



En el caso de esta aplicación, el funcionamiento y la arquitectura de la aplicación es muy similar a la versión monolítica ya que lo único que se distribuye es el bus de eventos.

La aplicación se compone de:

- Un *ChatManager*, un Verticle que se ocupa de la recepción de mensajes por parte de los clientes, su posterior distribución y la gestión de los usuarios (creación y eliminación).
- Varios *User*, Verticles que representan a cada usuario de la aplicación, que se encargan de almacenar la conexión WebSocket con su cliente para enviarle mensajes.

Funcionamiento

- **Conexión**: Cuando un usuario inicia la conexión WebSocket, al contrario de otras aplicaciones, no se realiza ninguna acción más que proporcionar un handler para los mensajes.

[7] `https://github.com/TFG-DistributedWebChat/Vertx-DistributedWebChat`

Cuando el cliente manda el mensaje de conexión, si el nombre no existe, se crear un nuevo User y se incluye en el contexto de la aplicación, guardando su id de Verticle en un mapa cuya clave es el nombre, tal y cómo podemos apreciar en el código a continuación:

```
// /src/main/java/com/globex/app/ChatManager.java
vertx.deployVerticle(user, res -> {
   if (res.succeeded()) {
      //Save the deploymentID to later remove the verticle
      users.put(name, res.result());
   } else {
      System.err.println("Error at deploy User");
   }
});
```

- **Re-envío de mensajes**: El chat manager es el encargado de recibir los mensajes de los clientes, publicándolos en el EventBus con la dirección igual a la sala de chat.

```
// /src/main/java/com/globex/app/ChatManager.java
vertx.eventBus().publish(message.getString("chat"), message);
```

Por otro lado, cuando un User es incluido en el contexto de la aplicación, se suscribe a su chat para recibir los mensajes dirigidos a esa sala y re-enviar a su cliente dichos mensajes. En el fragmento de código a continuación podemos apreciar esta subscripción y el callback que se ejecuta al recibir un mensaje:

```
// /src/main/java/com/globex/app/User.java
// Listen for messages from his chat
this.handler = vertx.eventBus().consumer(chat).handler(data -> {
   try{
      // Try to send the message
      this.wss.writeFinalTextFrame(data.body().toString());
   }catch(IllegalStateException e){
      // The user is offline, so I delete it.
      this.handler.unregister();
      vertx.eventBus().publish("delete.user", name);
      wss.close();
   }
});
```

- **Desconexión**: Se produce cuando User no es capaz de enviar un mensaje a su cliente. Publica su borrado en el EventBus y cierra la conexión. El evento de borrado es capturado por el ChatManager, que da de baja al User.

```
// /src/main/java/com/globex/app/ChatManager.java
vertx.undeploy(users.get(user_name));
users.remove(user_name);
```

Despliegue en clúster

La entidad de esta tecnología que nos permite distribuir el envío de mensajes es el *Event Bus* o bus de eventos. Este recurso puede utilizarse tanto en local cómo en distribuido de forma análoga. Para distribuirse, hace uso de Hazelcast[8] y es necesario declarar almenos un nodo semilla (al igual que Akka) aunque también permite definir una interfaz de red bajo la que descubrir los nodos. Para este proyecto hemos usado la primera opción para descubrir el nodo semilla mediante TCP simplemente añadiendo la dirección de un nodo (previamente lanzado en otra máquina) en el archivo de configuración *cluster.xml*:

```
<tcp-ip enabled="true">
   <interface>172.31.34.96</interface>
</tcp-ip>
```

Es posible configurar Hazelcast para que trabaje junto a un security group de AWS, pero resulta bastante más compleja.

Para desplegar el clúster se ha hecho uso de nuevo de un script en python que automatiza todos los pasos necesarios. Para más detalles, consultar el fichero README.md[9] del repositorio de Vertx.

Problemas en el desarrollo

Sin duda el mayor reto fué comprender cómo funciona Hazelcast a nivel de TCP-IP y que requiere de la IP privada y no la pública (otra tecnología que veremos más adelante, RabbitMQ, es capaz de resolver incluso a partir de DNS) lo que dió lugar a bastantes problemas ya que la documentación para clúster era bastante pobre y los ejemplos no proporcionaban la guía necesaria para hacer un uso correcto de esta funcionalidad.

[8] https://hazelcast.com/

[9] https://github.com/TFG-DistributedWebChat/Vertx-DistributedWebChat/blob/master/README.md

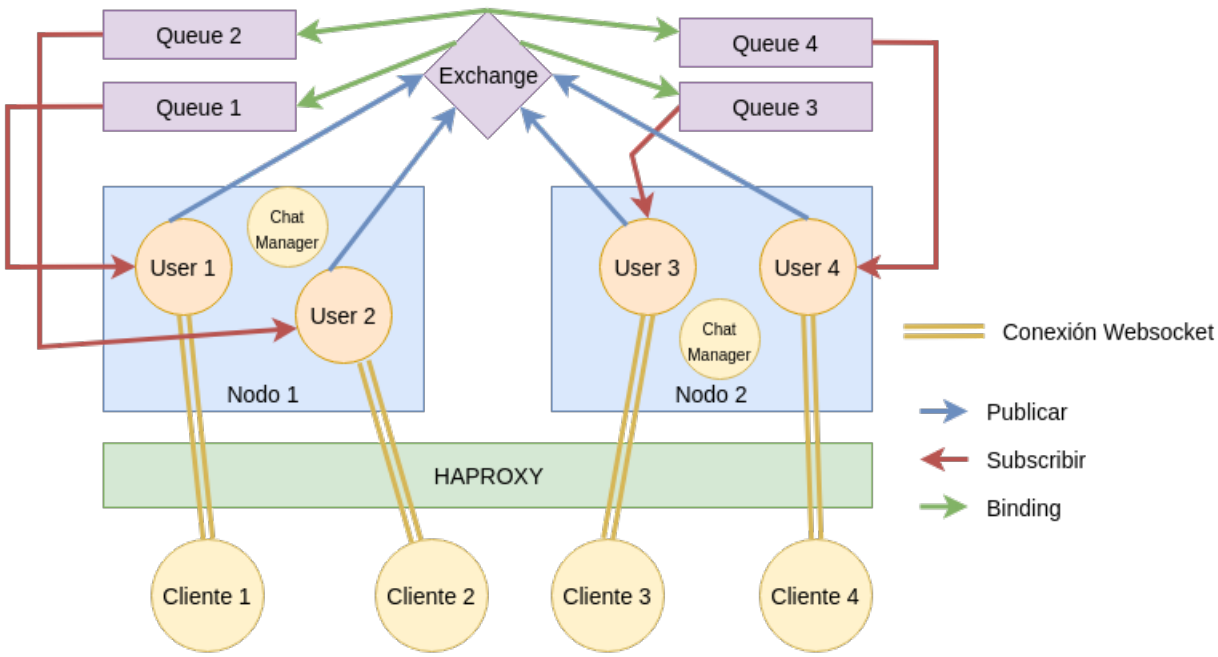
De forma adicional, Vert.x arrastraba del proyecto anterior un grave problema de memoria, siendo la tecnología que más uso hacia de este recurso. Tras trabajar con RabbitMQ (que también sufria de problemas similares y se hayó el problema) se llegó a la conclusión de que probablemente algun recurso registrado en el EventBus no estaba siendo liberado (pensando que simplemente con expulsar al actor del sistema, se borraba cualquier referencia). Finalmente se detecto que recurso era: un manejador (*handler*) que el usuario registraba para poder subscribirse a los mensajes de un chat. Para darle de baja solo era necesario guardar una referencia a este manejador y liberarlo cuando se cerrase la conexión Websocket:

```
this.handler.unregister();
```

4.2.3. Spring + RabbitMQ

El código de esta aplicación se puede encontrar en un repositorio en GitHub de la organización antes mencionada bajo en nombre de *SpringBoot-RabbitMQ-DistributedWebChat*[10]

Diseño y arquitectura



Cómo podemos apreciar, parte de la lógica del sistema se encuentra fuera de la aplicación de Java, en el servidor de RabbitMQ, será necesario por tanto una libreria/cliente AMQP para

[10] https://github.com/TFG-DistributedWebChat/SpringBoot-RabbitMQ-DistributedWebChat

poder conectarnos a dicho servidor.

Utilizaremos una cola para cada usuario y un único *Exchange* para todo el sistema. De esta forma, los usuarios comunicarán al *Exchange* sus mensajes y este se encargará de hacerlos llegar a las colas de los usuarios en su misma sala.

La aplicación de Spring hace uso de la anotación @ServerEndpoint sobre la clase User, que convierte a dicha clase en un punto de entrada para la conexión WebSocket. Permite a la clase implementar métodos bajo las anotaciones:

- **@OnOpen**: Se ejecuta cuando el usuario establece la conexión
- **@OnMessage**: Se ejecuta cada vez que el usuario manda un mensaje
- **@OnClose**: Se ejecuta cuando la conexión WebSocket se cierra
- **@OnError**: Se ejecuta cada vez que sucede un error en la conexión, capturándolo

Estas anotaciones permiten tener un control sencillo del flujo de la aplicación y de los eventos que requiere.

Cada vez que un cliente se conecta, se crea una instancia de User que se encargará de recoger los eventos de ese usuario en concreto. Esto es posible gracias a SpringBoot, que se encarga de servir esta clase como un componente reutilizable bajo la anotación *@Bean* en WebChatSpringBootApplication (que actúa como archivo de configuración):

```
// /src/main/java/com/chat/WebChatSpringBootApplication.java
@Bean
public ChatManager reverseWebSocketEndpoint() {
   return new User();
}
```

Todas las anotaciones utilizadas pertenecen a la librería WebSocket de Java[11].

La clase *User* implementa tambien la interfaz (a partir de una clase intermedia, *UserConsumer*) *Consumer*, que le permite ser subscriptor a una cola de RabbitMQ.

Por otro lado, cada aplicación/nodo cuenta con un ChatManager, una clase que implementa el patron *Singleton*[12] y a la que los User tienen acceso. Esta clase permite a un User:

[11] https://mvnrepository.com/artifact/javax.websocket/javax.websocket-ap
[12] https://es.wikipedia.org/wiki/Singleton

- Darse de alta en el registro de usuarios (comprobando previamente si existe el nombre).
- Subscribirse a un topic (nombre de la sala de chat). El Chat Manager se encarga de crear la conexión.

Funcionamiento

- **Conexión**: Al iniciarse la conexión, se crea un objeto de la clase *User* que manejara dicha conexión. Se ejecutará el método *@OnOpen*, que obtiene una referencia al *ChatManager*. El mensaje de conexión, en cambio, es capturado por el método bajo la anotación *@OnMessage*, que tras validar que el usuario asociado a esa sesión no tiene aún atributos como name o chat, comprueba que el nombre sea único y se los asigna. Se informa al *ChatManager* para que se le incluya y le subscriba a su sala de chat, asignandole al *User* su conexión con el *Exchange*.
- **Re-envío de mensajes**: Un *User* recibirá mensajes de dos formas:
  - Un nuevo mensaje del cliente llega desde el WebSocket y ejecuta el método *handleMessage()*. Este mensaje será enviado al *Exchange*, que se ocupará de distribuir dicho mensaje a todas las colas de otros usuarios, siendo el topic de este envío su sala de chat.

    ```
    this.channel.basicPublish(EXCHANGE_NAME, chat, null, message.
        getBytes());
    ```

  - Un nuevo mensaje procedente de su cola y que recoge en el metodo *sent()* que lo envía directamente por WebSocket al cliente.

    ```
    session.getBasicRemote().sendText(message);
    ```

- **Desconexión**: Cúando un cliente se desconecta (se cierra su conexión Websocket), se ejecuta el método bajo la anotación *@OnClose*, que cierra la conexión con RabbitMQ de ese usuario e informa al *ChatManager* para que le elimine del registro.

Despliegue en clúster

En este caso, la aplicación de Spring no se distribuye, si no que es el servicio externo, RabbitMQ, el que lo hará. Los pasos para formar un clúster RabbitMQ son algo más complejos que en el resto de aplicaciones:

- Arrancar el servicio RabbitMQ en ambas máquinas, deben estar en la misma versión y compartir una clave denominada *erlang-cookie* (un archivo de configuración).

- Detener RabbitMQ en una de las máquinas, ejecutar:

  ```
  rabbitmqctl join_cluster rabbit@ip-${MASTER-IP}
  ```

  siendo MASTER-IP la IP de la máquina que no hemos detenido. Arrancamos de nuevo RabbitMQ y ya se habrá formado el clúster

RabbitMQ permite usar plugins de mucha utilidad. Para esta aplicación ha sido de especial interés el plugin *rabbitmq-management* que nos permite acceder desde el navegador a diferentes métricas de nuestro clúster: número de colas, conexiones, exnchanges, además del uso de memoria.

Una vez nuestro clúster esta desplegado, basta con lanzar nuestra aplicación de Spring en cada máquina.

Problemas en el desarrollo

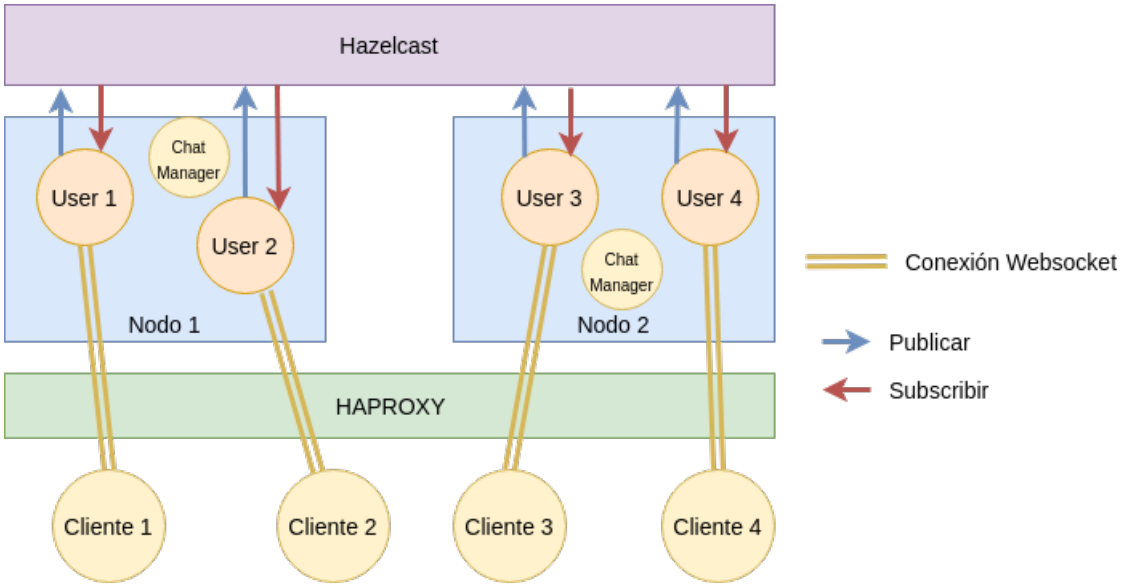
En contraste a las otras tecnologías que son toolkits completos, con RabbitMQ y Spring nos encontramos el problema de lidiar con un servicio externo. La mayor dificultad de esta aplicación fué distribuir los nodos de RabbitMQ, comprender la importancia de la *erlang-cookie* y manejar la unión de nodos, que si no se seguian los pasos en un orden concreto, era imposible de realizar. Aunque la documentación oficial es muy completa, si duda ha sido gracias a la comunidad y a la lectura de numerosos ejemplos de terceros lo que ha hecho posible la comprensión de este proceso.

Otro problema que surgió a raiz de un desconocimiento de la gestión de recursos del cliente AMQP fué el no cerrar apropiadamente las conexiones de los usuarios. Esto provocaba un aumento constante de la memoria RAM utilizada por la máquina, que para las pruebas más exigentes provocaba la parada del servicio. Gracias al plugin antes mencionado, se pudo monitorizar los nodos para detectar este problema, haciendo los cambios pertinentes con resultados satisfactorios.

4.2.4. Spring + Hazelcast

El código de esta aplicación se puede encontrar en un repositorio en GitHub de la organización antes mencionada bajo en nombre de *SpringBoot-RabbitMQ-DistributedWebChat*[13]

Diseño y arquitectura



Al contrario que la aplicación de Spring junto a RabbitMQ, esta aplicación usa una arquitectura más similar a Vert.x (ambos usan Hazelcast), pero es muy parecida en cuanto a código se refiere a la primera, por lo que se mencionaran los cambios realizados.

La aplicación hace uso de la anotación @ServerEndpoint sobre la clase User de nuevo, no varia su implementación salvo para el envío y recepción de mensajes de otros usuarios.

Cada *User* implementa la interfaz *MessageListener* que le permite subscribirse y recibir mensajes de Hazelcast. El *ChatManager*, de nuevo único para la aplicación, se limitara a gestionar las altas y bajas de usuarios, manteniendo una referencia a la instancia de Hazelcast para ello.

Funcionamiento

El funcionamiento general de la aplicación es muy similar al de su versión con RabbitMQ.

- **Conexión**: Al iniciarse la conexión, se crea un objeto de la clase *User* que manejara dicha conexión. Se ejecutará el método *@OnOpen*, que obtiene una referencia al *ChatManager*.

[13] `https://github.com/TFG-DistributedWebChat/SpringBoot-RabbitMQ-DistributedWebChat`

El mensaje de conexión, en cambio, es capturado por el método bajo la anotación *@OnMessage*, que tras validar que el usuario asociado a esa sesión no tiene aún atributos como name o chat, comprueba que el nombre sea único y se los asigna. Se informa al *ChatManager* para que le subscriba a su sala de chat, devolviendole un *topic*, una referencia a su sala de chat para poder interactuar con ella. Además, se guarda junto a su nombre un identificador único con el que poder darse de baja más adelante.

- **Re-envío de mensajes**: Un *User* recibirá mensajes de dos formas:
  - Un nuevo mensaje del cliente llega desde el WebSocket y ejecuta el método *handleMessage()*. Este mensaje será enviado a través de la referencia del topic a todos los usuarios subscritos a él.

```
@OnMessage
public void handleMessage(Session session, String message){
    // ...
    this.topic.publish(message);
    // ...
}
```

  - Un nuevo mensaje de su topic y que recoge en el metodo *onMessage()* que lo envía directamente por WebSocket al cliente.

```
@Override
public void onMessage(Message<String> message) {
    this.send(message.getMessageObject());
}
```

- **Desconexión**: Cúando un cliente se desconecta (se cierra su conexión Websocket), se ejecuta el método bajo la anotación *@OnClose*. Se informa al *ChatManager* para que le tramite la baja y le desubscriba del topic.

Despliegue en clúster

El despliegue en clúster es idéntico al de la aplicación de Vert.x. Al usar Hazelcast, no se distingue al hacer uso de el tanto en local cómo en distribuido. Sólo es necesario definir el archivo de configuración de hazelcast, al igual que en Vert.x.

Se proporciona, al igual que en otras tecnologías, los scripts necesarios para automatizar este proceso

Problemas en el desarrollo

Fué relativamente sencillo construir y lanzar esta aplicación en un clúster tras comprender cómo funciona Hazelcast en la aplicación de Vert.x. Se intentó automatizar aún más su despligue haciendo uso de los servicios de AWS, pero se llegó a la conclusión que era mucho más laborioso y no aportaba ningun valor añadido.

4.3. Pruebas

Las aplicaciones desarrolladas carecen de pruebas unitarias o de integración propias, comparten un cliente común capaz de probarlas de forma completa y distribuida.

A lo largo de este apartado hablaremos de este cliente, su implementación y de sus características.

El código de esta aplicación se puede encontrar en un repositorio en GitHub de la organización antes mencionada bajo en nombre de *DistributedWebChatClient*[14]

4.3.1. Cliente heredado

Al igual que las aplicaciones, se contaba con un cliente de pruebas heredado del proyecto anterior. Este cliente usaba JUnit junto a herramientas de testing de Vert.x para probar las distintas aplicaciones de chat. El funcionamiento de este cliente era el siguiente:

1. Se escribe un archivo de configuración dónde se definia cada aplicación a probar en local y testeaba una tras otra.odía probarse tanto cómo si ya estaba lanzada cómo si no, en este caso, el propio cliente era capaz de lanzarla.

2. Se abre un cliente web en el navegador por defecto dónde van apareciendo los resultados.

3. Generaba X clientes para Y salas de chat, de forma que cada cliente enviaba (además de su mensaje de conexión) 500 mensajes al resto de usuarios en un periodo de 5 segundos.

[14] `https://github.com/TFG-DistributedWebChat/DistributedWebChatClient`

4. El mensaje contiene el momento (en milisegundos) en el que es enviado el mensaje. Cada cliente va recibiendo los mensajes y almacenando el tiempo que tardan en llegar (momento actual - momento que trae el mensaje).

5. Cuando todos los clientes han recibido todos los mensajes (sin pérdidas) se divide el tiempo total entre el número total de mensajes (Nº usuarios x Nº usuarios x 500). Este proceso se repite 10 veces de forma identica para obtener mayor homogeneidad. Al terminar estas iteraciones, en envíaba el resultado al cliente web.

6. De forma paralela, se recogen métricas de uso de CPU y de memoria para ese proceso.

Los pasos 3, 4, 5 y 6 se repiten para cada aplicación con distinto número de clientes y salas de chat.

4.3.2. Desarrollo e implementación

El cliente descrito en la sección anterior estaba límitado a aplicaciones en local, aunque era capaz de medir la latencia en máquinas externas es necesario incluir una forma de medir el resto de métricas.

Dado que usar archivos de test dificultaba bastante seguir el flujo de la aplicación, se optó por re-escribir dicho cliente desde 0 (incluyendo el cliente web), manteniendo las funcionalidades principales.

Entre los objetivos de esta re-implementación se encuentran:

- Re-estructurar la ejecución de los casos de prueba para que funcione cómo una aplicación normal y no cómo un proceso de test.
- Permitir obtener métricas de máquinas remotas (métrica por nodo/máquina).
- Notificar al usuario de la aplicación web del progreso de las iteraciones.
- Re-escribir el cliente web de forma que permita una nueva forma de entrada de configuración (en lugar de en un archivo)y permita mostrar las diferentes métricas por nodos.

Re-estructuración

En este primer paso, re-escribiremos la aplicación de Java para que no haga uso de archivos de test, aunque seguiremos usando el toolkit Vert.x para gestionar los callbacks de las pruebas.

Para comunicar los distintos componentes de la aplicación (incluso en el cliente web) usaremos el EventBus de Vert.x. Los componentes principales de la aplicación son:

- **TestResultsServer**: un sencillo servidor de Vert.x que nos permite ofrecer un cliente web con una conexión directa con él. Es el encargado de lanzar al ClientManager dentro del contexto de Vert.x. Del cliente, el cual detallaremos más a delante, llegará la configuración concreta de las pruebas que se quieren realizar, que enviará al *ClientManager*
- **ClientManager**: un actor de Vert.x que atenderá las peticiones de nuevas pruebas y las ejecutará, comunicando los resultados directamente al cliente web mediante el bus de eventos.
- **ClientGenerator**: un actor de Vert.x que recibirá un caso de prueba a realizar por parte del *ClientManager* y lo ejecutará, obteniendo las diferentes métricas.

La configuración que recibe el ClientManager se estructura cómo un JSON. Un ejemplo de esta configuración sería la siguiente:

```
{
   name: "RabbitMQ",
   address: "127.0.0.1",
   port: 5000,
   pem: "TFG.pem",
   isDistributed: true,
   nodes: [
      "ec2-34-244-127-84.eu-west-1.compute.amazonaws.com",
      "ec2-34-243-28-134.eu-west-1.compute.amazonaws.com"
   ],
   cases: [
      { numChats: 1, numUsers: 10 },
      { numChats: 2, numUsers: 20 },
      { numChats: 3, numUsers: 25 }
   ]
}
```

dónde:

- *name* es el nombre que daremos a la aplicación
- *address* es la dirección del punto de entrada del proxy.
- *port* es el puerto en el cual escucha el proxy.
- *pem* es el nombre del archivo PEM que utilizaremos más adelante.
- *isDistributed* nos indica si la aplicación es distribuida o local.
- *nodes* son las direcciones DNS de los nodos concretos de la aplicación (en el caso de ser esta distribuida)
- *cases* son los casos para los que queremos probar nuestra aplicación. Se compone de una tupla con el número de salas de chat que habrá y el número de usuarios por cada una de estas salas.

A partir de esa configuración, el *ClientManager* ejecutará cada caso 10 veces. En cada iteración de del caso de prueba se hará uso de un *ClientGenerator*, que es el encargado de generar los clientes que interactuarán con la aplicación. Este *ClientGenerator* resultaría muy similar en cuanto a funcionamiento al heredado, siendo capaz de recoger la latencia media de cada caso.

Al acabar cada iteración, se devuelve el resultado de la misma al *ClientManager* que lo seguirá completando con las iteraciones posteriores y lanza un evento en el bus de eventos dónde informa de la iteración completada.

De esta forma ya tendriamos un cliente de pruebas sencillo y funcional.

El siguiente paso sería incluir una recogida de métricas (uso de CPU y memoria) para cada nodo definido en la configuración. Para hacerlo lo más homogéneo posible, recogeremos esta métricas para la máquina completa (cuyo uso es exclusivo para ejecutar la aplicación de chat). Las máquinas que utilizaremos estarán basadas en Linux, concretamente en Ubuntu, por lo que podremos usar los siguientes scripts para obtener las métricas por nodo/máquina:

Obtención de uso de CPU:

```
#!/bin/bash
ssh -i pems/$PEM $MACHINE top -b -n 2 | grep "Cpu(s):" | awk '{ print $2 }'
```

Obtención de uso de memoria:

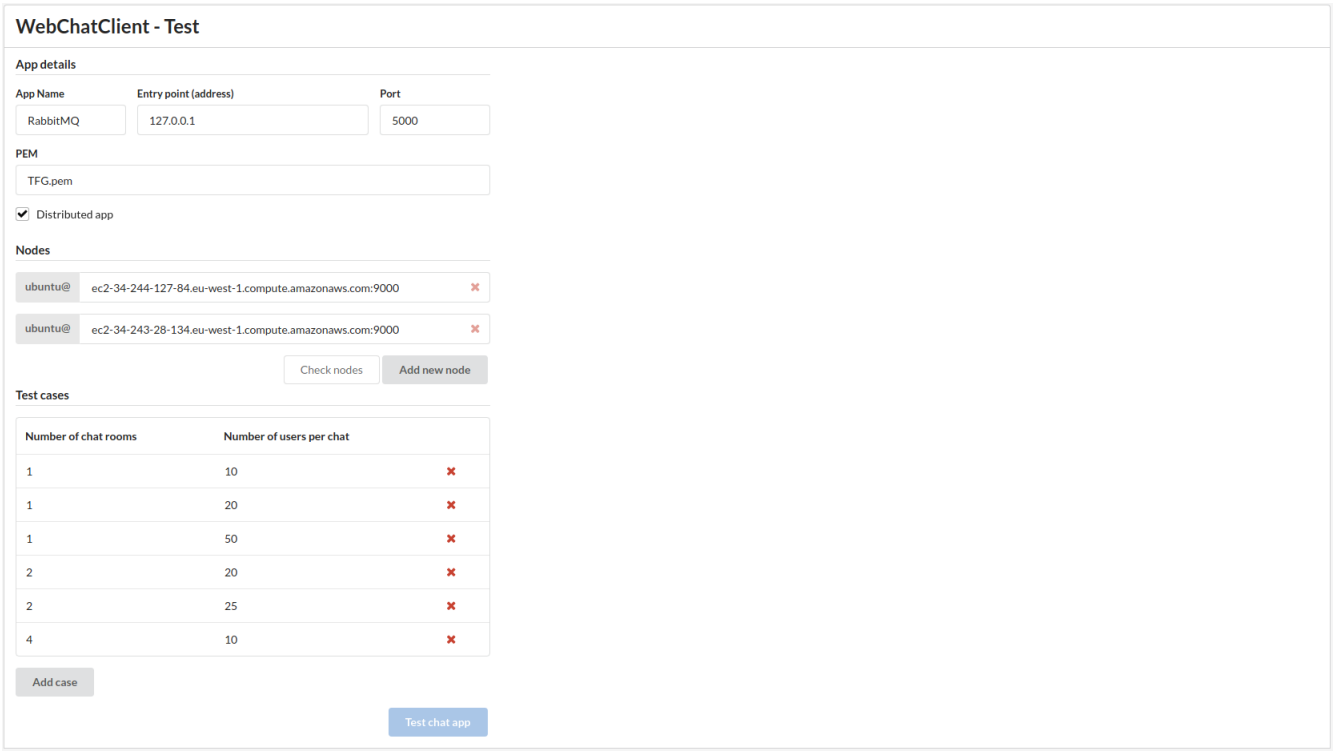
```
#!/bin/bash
ssh -i pems/$PEM $MACHINE free -m | grep "Mem:" | awk '{ print $3 }'
```

Dónde *PEM* es el nombre del archivo .pem[15] definido en la configuración y MACHINE será la dirección de la máquina también definida en la configuración. Este archivo debe estar presente en la carpeta */pems* antes de iniciar la aplicación.

En la aplicación será el *ClientGenerator* la entidad que hará uso de estos scripts desde Java. En cada iteración de cada caso, esta entidad generará un hilo de ejecución que ejecutará estos scripts de forma periodica (cada segundo). Tras finalizar la iteración, se hara la media de cada métrica para incluirlo en el resultado.

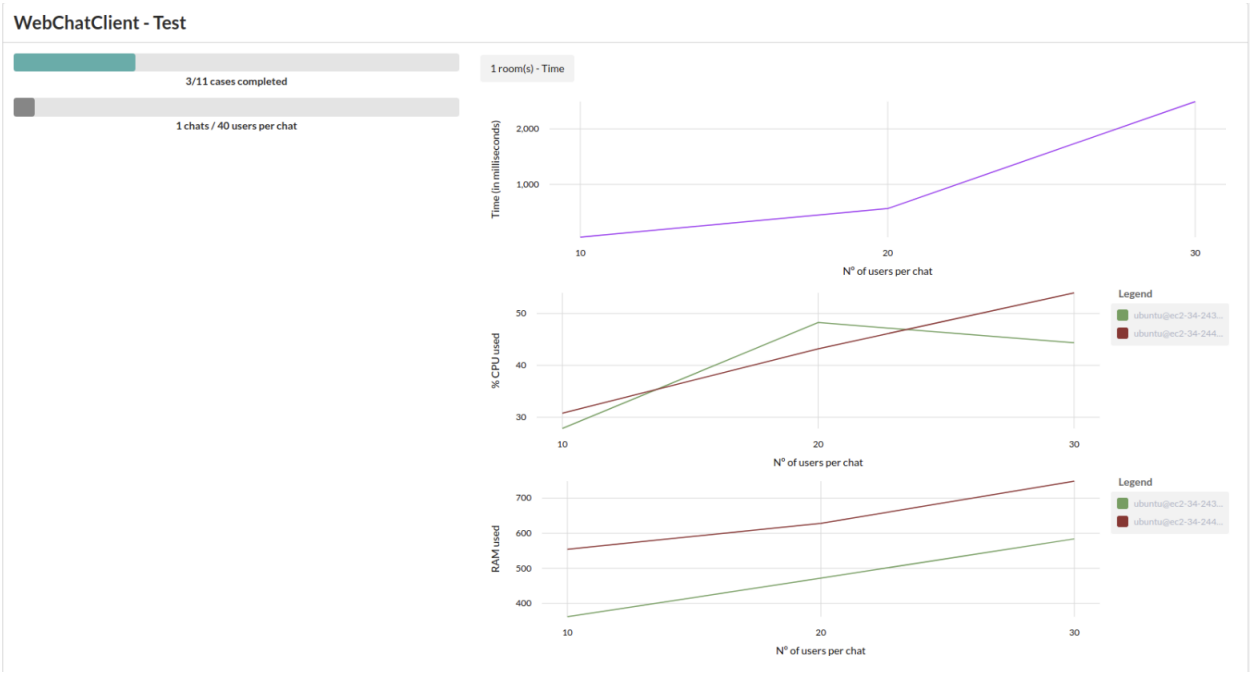
Por último, habría que re-escribir el cliente web (escrito en Angular) para soportar este nuevo servidor las opciones que ofrece.

Para ello se proporcionará una interfaz en la que poder introducir la configuración completa cómo se puede apreciar en la imagen:



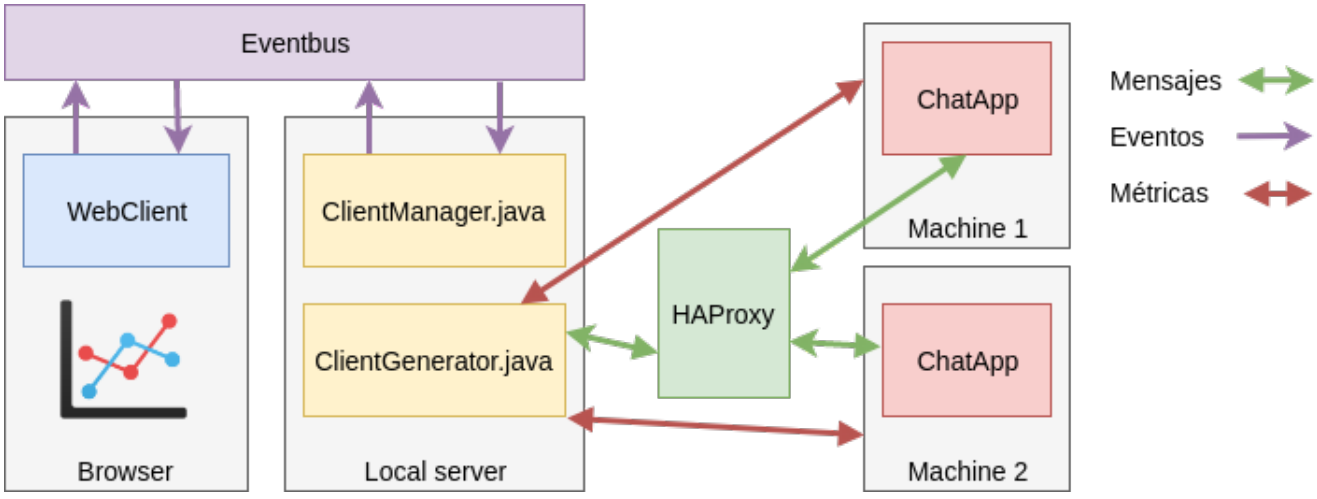
[15]Un archivo .pem nos permite realizar una conexión ssh con una máquina remota y ejecutar comandos sobre ella.

Una vez se introduzcan la configuración de las pruebas a realizar, la siguiente pantalla deberá visualizar el progreso de la misma mientras se van mostrando los resultados, incluyendo las métricas de CPU y memoria por nodo:

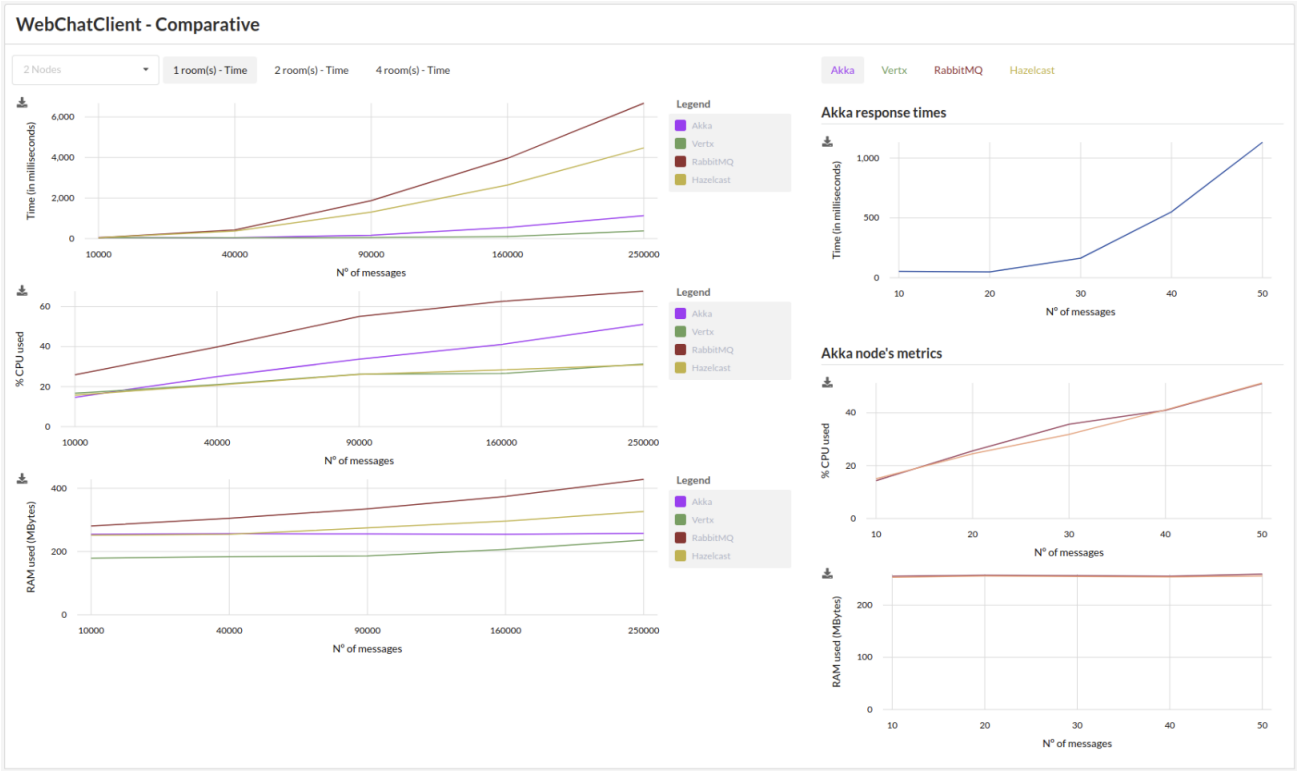


Al finalizar las pruebas, obtendremos los datos en formato JSON.

La arquitectura de nuestro sistema de pruebas resultante es la siguiente:



Esta aplicación nos permite obtener las métricas de latencia, uso de CPU y memoria de una aplicación distribuida. Una vez tengamos los resultados de varias aplicaciones, se pueden comparar en un sencillo dashboard interactivo:



4.3.3. Funcionamiento

A continuación se detalla el flujo del funcionamiento de la aplicación:

- **Paso 0:** Antes de comenzar, nos aseguramos de que las máquinas y el proxy esten operativos, situamos el archivo PEM bajo la carpeta */pems* del proyecto de la aplicación de pruebas y lanzamos la aplicación. Esto abrirá automaticamente el cliente en nuestro navegador.

- **Paso 1:** Introducimos la configuración de las pruebas que queremos realizar (configuración del proxy, los nodos y los casos) en el navegador y los enviamos al servidor.

- **Paso 2:** El *ClientManager* recibe la configuración, crea un *ClientGenerator* y lo lanza junto a un caso de prueba.

- **Paso 3:** El *ClientGenerator* genera tantos clientes Websocket cómo define la configuración para que se conecten a la aplicación a través del proxy (Estos clientes pueden pertenecer a salas diferentes). A la vez, de forma periodica, recoge las distintas métricas para cada nodo que forma la aplicación distribuida a probar.

- **Paso 4:** Tras recoger los tiempos de respuesta y las métricas de CPU y memoria, se guardan en un resultado y se devuelven en un callback al *ClientManager*

- **Paso 5:** El *ClientManager* recibe el callback y emite un evento de finalización de la iteración. Este evento es capturado por el cliente web que muestra el progreso de ese caso por pantalla.

  Los pasos 2-5 se repiten para cada iteración de cada caso.

- **Paso 6:** Cuando no hay mas casos que probar, el *ClientManager* lanza un evento de finalización, de nuevo capturado por el cliente web que provoca la descarga de los resultados en un JSON

- **Paso 7:** Una vez hemos probado varias aplicaciones, situamos dichos archivosen la carpeta *src/main/resources/webroot/results*, los declaramos en el archivo *index.json*, relanzamos la aplicación de pruebas vamos a la ruta */comparative* para visualizar la comparativa por cada métrica entre las distintas aplicaciones para distintos criterios (número de salas de chat y número de máquinas).

Capítulo 5

Estudio comparativo

Una vez disponemos de todas las aplicaciones desarrolladas y una herramienta para probarlas, llega el momento de realizar la comparativa entre tecnologías.

Las aplicaciones han usado cómo máquinas para albergar sus nodos instancias EC2 de AWS, concretamente el modelo t2.micro [1] debido a que es el único que entra dentro de la capa gratuita y no es necesario probarlas en máquinas con mayor capacidad para compararlas. Esta instancia cuenta con 1 CPU y 1 GB de memoria RAM. El sistema operativo elegido para las instancias ha sido Ubuntu 16.04 por su estabilidad, la comunidad detrás ante cualquier problema y el hecho de que las pruebas en local se realizaron en el mismo sistema operativo.

En lugar de usar una máquina remota, se ha usado HAProxy en la misma máquina en la que se lanza el cliente de pruebas, un sistema Ubuntu 16.04 con 16 GB de memoria RAM y 4 procesadores.

Los casos definidos para esta prueba han sido:

- 1 sala de chat con 10, 20, 30, 40, 50 y 60 usuarios
- 2 salas de chat con 20, 25, 30 y 35 usuarios en cada una
- 4 salas de chat con 10, 12, 15 y 17 usuarios en cada una

Se han repetido los caso para el uso por máquina de 2, 3 y 4 nodos.

Debido a que las máquinas remotas son menos potentes que la utilizada para el escalado vertical del proyecto anterior, se ha reducido los mensajes por usuario de 500 a 100.

[1] `https://aws.amazon.com/es/ec2/instance-types/`

Desglosaremos la comparativa en las distintas métricas que vamos a medir: latencia, uso de CPU y uso de memoria, además de comparar la dificultad/complejidad del desarrollo. Una vez realizado el estudio, procederemos a formular una serie de conclusiones.

Para interpretar las gráficas que usaremos para mostrar los resultados de las pruebas es importante comprender su estructura:

Cada gráfica contiene los resultados de una métrica concreta para un número de salas concreto.

En el eje Y encontraremos la métrica estudiada

En el eje X encontraremos el número de mensajes que se han enviado en total. Este número se obtiene de la siguiente fórmula:

$$N_o deMensajes = (N_0 UsuariosPorSala)^2 * N_0 DeSalas * 100$$

Junto a cada métrica, se comentarán los resultados del estudio comparativo previo.

5.1. Latencia

5.2. Uso de CPU

5.3. Uso de memoria

5.4. Desarrollo

5.5. Conclusiones generales de la comparativa

Capítulo 6

Conclusiones del proyecto y trabajos futuros

Mis conclusiones

Apéndice A

Despliegue de instancias en AWS

Para desplegar las instancias necesarias que usaremos indistintamente para cualquier tecnología necesitaremos cumplir los siguientes requisitos:

- Disponer de una cuenta en AWS (se recomienda que tenga en vigencia el año de capa gratuita para evitar costes)
- Tener instalada la interfaz de línea de comandos de AWS (aws-shell)

Una vez cumplimos los requisitos, será necesario crear un grupo de seguridad para nuestras instancias al que llamaremos *Cluster*.

A continuación se muestra cómo crear un *SecurityGroup* mediante aws-shell, dónde *security-group-id* será el id devuelto por el primer comando.

```
aws ec2 create-security-group \
   --group-name Cluster \
   --description "My security group"

aws ec2 authorize-security-group-ingress \
   --group-id security-group-id \
   --protocol -1 --cidr 0.0.0.0/0

aws ec2 authorize-security-group-ingress \
   --group-id security-group-id \
   --protocol tcp \
   --port 22 \
   --cidr 0.0.0.0/0
```

Estos comandos nos generarán un *SecurityGroup* con unas reglas de entrada poco restrictivas pero muy flexibles de forma que no tengamos problemas para ninguna tecnlología. También permite que el desarrollador pueda conectarse mediante SSH. Puede especificarse una IP concreta para aumentar la seguridad.

Una vez creado el *SecurityGroup*, lo utilizaremos para lanzar nuestras instancias. Para este experimento se han hecho uso de instancias t2.micro con la AMI correspondiente a una máquina Ubuntu Server 16.04 LTS, de forma que nos mantenemos en la capa gratuita. Para lanzar las instancias, haremos uso de nuevo del cliente por terminal ejecutando el siguiente script:

```
aws ec2 run-instances \
   --image-id ami-58d7e821 \
   --count N \
   --instance-type t2.micro \
   --key-name MyPem \
   --security-group-ids security-group-id \
   --subnet-id subnet-dd426694
```

- *N* es el número de instancias a crear
- *MyPem* es el par de claves de seguridad para poder conectarnos a la instancia mediante SSH. Si no disponemos de un par, podemos crearlas desde la consola web[1].
- *security-group-id* es el id del *SecurityGroup* creado en el paso anterior.
- La subnet *subnet-dd426694* corresponde a la subnet por defecto de la zona eu-west-1a.

Una vez las instancias esten lanzadas y disponibles, podremos desplegar cualquiera de nuestras aplicaciones.

[1] `https://docs.aws.amazon.com/es_es/AWSEC2/latest/UserGuide/ec2-key-pairs.html#having-ec2-create-your-key-pair`