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

Capítulo 3

Tecnologías, Herramientas y Metodologías

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.

[1] `https://tools.ietf.org/html/rfc6455`

[2] `https://www.w3.org/TR/2011/WD-websockets-20110929`

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)};
```

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
- **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

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

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

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.

3.1.4. Vert.x

Vert.x[5] 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:

- **Verticle**[6]: modelo de concurrencia que propone Vertx. Un Verticle es una clase que se comporta como un actor[7], 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.

[5] http://vertx.io/
[6] http://vertx.io/docs/vertx-core/java/#verticles
[7] https://en.wikipedia.org/wiki/Actor_model

- **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[8]:

- **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**[9]: 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.
    - 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.

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

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

- 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.

3.1.5. SpringBoot

Spring Boot[10] comprende un módulo de Spring[11] (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).

Además, cuenta con el Sistema de Inversión de Control de Spring[12][13], 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[14] (que a su vez es una forma de inversión de control).

La versión utilizada de Spring para este proyecto es la 1.4.3

[10] `http://projects.spring.io/spring-boot/`
[11] `https://spring.io/`
[12] `https://en.wikipedia.org/wiki/Inversion_of_control`
[13] `https://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html`
[14] `https://en.wikipedia.org/wiki/Dependency_injection`

3.1.6. RabbitMQ

RabbitMQ[15] es un software de mensajería de código abierto escrito en Erlang[16] que implementa el protocolo de cola de mensajes avanzados (AMQP[17]), además de otros protocolos que ha ido añadiendo cómo STOMP[18] y MQTT[19]. 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.
- **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.

[15] `https://www.rabbitmq.com/`
[16] `https://www.erlang.org/`
[17] `https://es.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol`
[18] `https://en.wikipedia.org/wiki/Streaming_Text_Oriented_Messaging_Protocol`
[19] `https://en.wikipedia.org/wiki/MQTT`

3.1.7. 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.

3.1.8. Angular

Angular[20] es un framework de JavaScript (aunque comúnmente se utiliza con Typescript[21], 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

[20] `https://angular.io/`
[21] `https://www.typescriptlang.org/`

- Una numerosa comunidad y una extensa documentación

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

3.2. Herramientas

3.2.1. Control de versiones: Git

Git[22] 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[23], 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[24] 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

[22] `https://git-scm.com/`
[23] `https://github.com`
[24] `https://maven.apache.org/`

describe las dependencias y permite añadir opciones de ejecución, test y desplegamiento de la aplicación.

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

3.2.2.2. SBT

SBT[25] 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[26] (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[27], 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[28].

[25] http://www.scala-sbt.org/
[26] https://aws.amazon.com
[27] https://aws.amazon.com/es/ec2
[28] https://aws.amazon.com/es/cli

3.2.4. Entornos de desarrollo

3.2.4.1. IntelliJ

IntelliJ[29]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[30] 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.

[29] https://www.jetbrains.com/idea/
[30] https://atom.io/

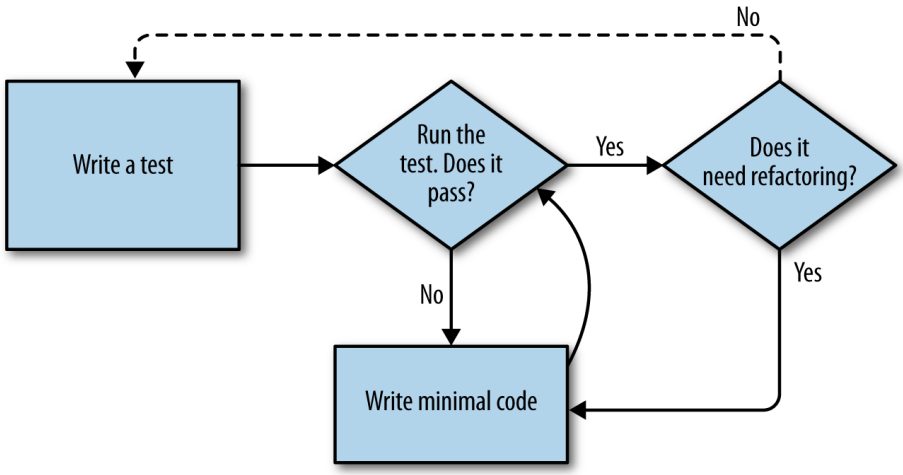
Se ha utilizado de forma conjunta con IntelliJ para desarrollar la aplicación de pruebas, ya que cuenta con paquetes que nos ayudan a desarrollar aplicaciones en Angular y soporte para Typescript.

3.3. Metodologías

El modelo de desarrollo de este proyecto se ha llevado a cabo a través de TDD[31](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.

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.

[31] https://es.wikipedia.org/wiki/Desarrollo_guiado_por_pruebas

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 siguiente organización de GitHub:

`https://github.com/TFG-DistributedWebChat`

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:

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 [1], 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 la misma que la del mensaje.

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

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.

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.

- **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

4.3. Pruebas

Capítulo 5

Estudio comparativo

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

Manual de usuario