Universidad Rey Juan Carlos

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMATICA

GRADO EN INGENIERÍA DE COMPUTADORES

Curso Académico 2015/2016

Proyecto de Fin de Carrera

DISEÑO E IMPLEMENTACIÓN DE SISTEMAS REACTIVOS ESCALABLES Y TOLERANTES A FALLOS

Autor: Francisco Javier Mateos Segano

Tutor: Micael Gallego Carrillo

AGRADECIMIENTOS

A mi familia y amigos, por apoyarme siempre que lo he necesitado y a mi tutor Mica por ayudarme y guiarme durante la realización de este proyecto.

RESUMEN

En este proyecto trataremos los problemas de la programación distribuida utilizando un modelo de programación distinto al habitual, el modelo de actores. El proyecto consistirá en una aplicación simple, un chat, que desarrollaremos con dos toolkits que utilizan este modelo, Akka y Vert.x. Durante el desarrollo nos centraremos en aspectos importantes de la programación distribuida como son la escalabilidad y la tolerancia a fallos, y una vez desarrollados, compararemos aspectos como el rendimiento y la facilidad de desarrollo para tratar de concluir cual de las dos implementaciones es mejor.

INDICE GENERAL

Índice de ilustraciones

1. INTRODUCCIÓN Y MOTIVACIÓN

Actualmente, el desarrollo de aplicaciones distribuidas es cada vez mas importante debido a la cantidad creciente de usuarios y datos que estas deben manejar sin perder rendimiento. Como ejemplo mas claro tenemos facebook, que en el segundo cuatrimestre de 2015 alcanzó la cifra de 1,490 millones de usuarios activos[1].

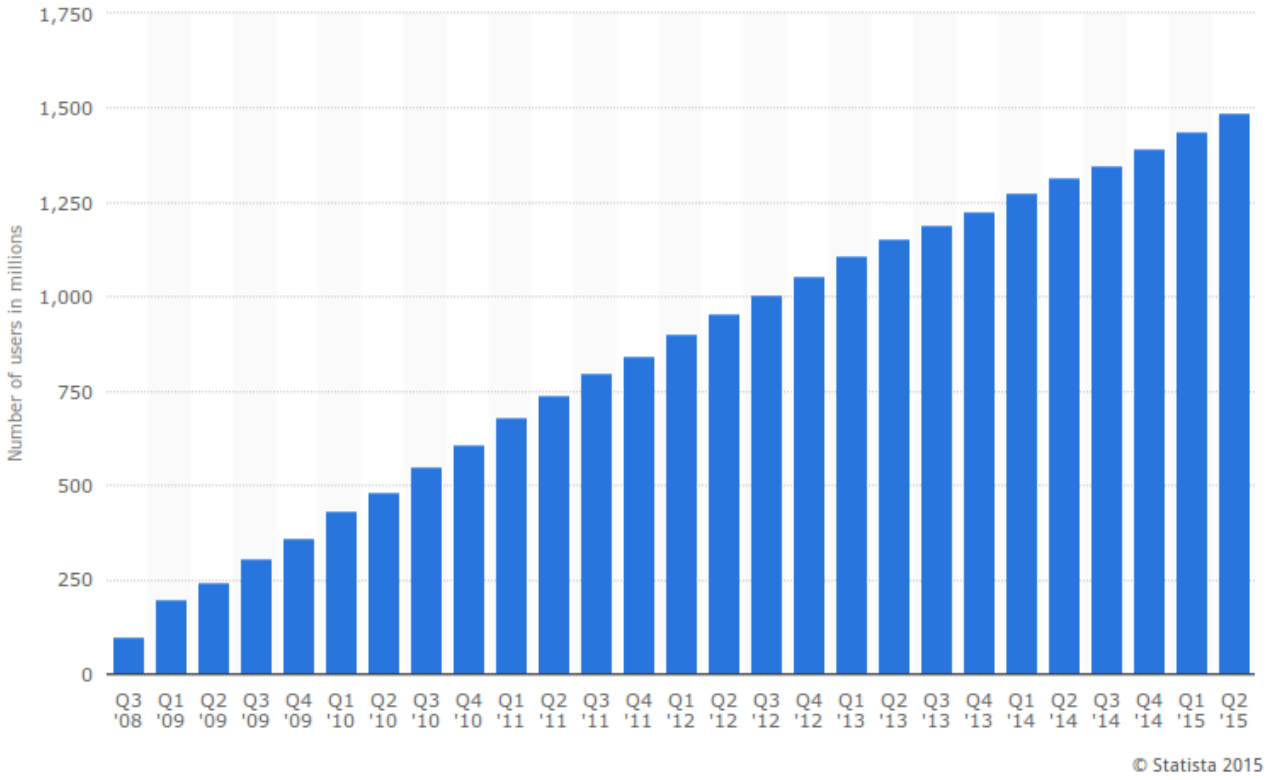


Ilustración 1: Gráfica de usuarios de Facebook

Datos como estos nos hacen ver la importancia del desarrollo de aplicaciones distribuidas y escalables, que puedan adaptarse a este crecimiento.

Según Coulouris[2], los sistemas distribuidos plantean una serie de desafíos, como pueden ser:

• La **heterogeneidad** que se da en redes, hardware de ordenadores, sistemas operativos, lenguajes de programación, implementaciones de diferentes desarrolladores. La forma de tratar esta diversidad es mediante el estándares, como por ejemplo los protocolos HTTP, FTP, SMTP...

• La **extensibilidad** determina si un sistema puede ser ampliado y/o re-implementado en distintos aspectos: Añadir nuevos recursos y servicios, ampliar la capacidad de servicio de los ya existentes o modificar los ya existentes por otros más capaces.

• **Seguridad**. Los recursos de información que un sistema distribuido maneja pueden ser de alto valor, por ello se deben cumplir las bases de la seguridad informática: confidencialidad, integridad y disponibilidad.

• Se considera que un sistema es **escalable** si conserva su efectividad cuando ocurre

un incremento significativo en el número de recursos y número de usuarios. El diseño de sistemas escalables presenta como retos:

- Control de los recursos físicos: al crecer la demanda debería ser posible, a coste razonable, extender los recursos físicos que prestan el servicio (ej Balanceo de carga / *mirroring*)

- Control de las pérdidas de prestaciones.

- Prevención del desbordamiento de recursos SW

- Prevención de los cuellos de botella

Dentro de la escalabilidad tenemos dos tipos:

Vertical: Que consiste en remplazar el hardware por uno mas potente. Es sencillo de poner en práctica pero está limitado por la tecnología disponible y el coste no suele escalar linealmente: un servidor el doble de rápido suele ser más del doble de caro.

Horizontal: Consiste en añadir hardware adicional, que complementa al actual. Es más complejo de diseñar, construir y mantener y requiere planificación: el SW y HW debe permitir hacerlo

• **Tratamiento de fallos**: Un sistema distribuido bien implementado proporciona un alto grado de disponibilidad frente a los fallos del hardware, ya que cuando falla algún componente del sistema distribuido sólo resulta afectado el trabajo relacionado con el componente defectuoso.

• **Concurrencia**: Una característica básica de los sistemas distribuidos es la compartición de recursos. Existe la posibilidad de que un mismo recurso sea accedido por varios usuarios al mismo tiempo por lo tanto es necesario disponer de mecanismos que garanticen la integridad de los recursos ante accesos concurrentes

• **Transparencia**: Es la ocultación al usuario y al programador de la separación de los

componentes del sistema distribuido, de forma que lo perciba como un todo más que como una colección de componentes individuales.

Durante el desarrollo de este trabajo, trataremos la mayoría de estos temas, centrándonos principalmente en dos de estos desafíos, la escalabilidad y la tolerancia a fallos. Para tratar otro de los temas, la concurrencia, hemos decidido utilizar *toolkits* basados en el **modelo de actores**[3][4], propuesto por Carl Hewitt en 1973. Este modelo matemático está basado en el paso de mensajes y los "actores" son las primitivas del lenguaje. Estos actores tienen un *mailbox* y un comportamiento, y en función del mensaje que reciban en su *mailbox*, pueden crear mas actores, enviar mensajes a otros actores o variar su comportamiento para los siguientes mensajes. La importancia de este modelo es que todas las comunicaciones son llevadas a cabo de manera asíncrona Una vez enviado el mensaje, el actor continua inmediatamente con su ejecución. Tampoco hay garantías de que los mensajes vayan a ser recibidos en orden por el destinatario. Otra característica importante es que todas las comunicaciones se realizan mediante paso de mensajes, por lo que no hay estado compartido entre los actores.

Además de utilizar el modelo de actores, los *toolkits* que utilizaremos son reactivos, es decir, cumplen las características descritas en el **reactive manifesto**[5], escrito por Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. La motivación[52] de este manifiesto es dar nombre a este nuevo tipo de aplicaciones (similar a NOSQL, Big Data, REST) y definir un vocabulario común para describirlas. En este manifiesto se explica que los sistemas reactivos deben ser:

• **Responsivos (Responsive)**: Los sistemas responden de una manera oportuna en la medida de lo posible. Se enfocan en proveer tiempos de respuesta rápidos y consistentes, estableciendo límites superiores, de manera que ellos entreguen una calidad de servicio. Este comportamiento consistente se traduce en una simplificación del tratamiento de errores.

• **Resistentes (Resilient)**: Los sistemas permanecen responsivos cuando aparecen fallos. Esto no solo se aplica a los sistemas críticos o de alta disponibilidad. Esta resistencia se alcanza mediante la replicación, contención, aislamiento y delegación. Los errores son contenidos dentro de cada componente, aislándolo de los demás y así asegurando que las partes puedan fallar y recuperarse por si mismas sin comprometer el sistema.

• **Elásticos (Elastic)**: El sistema se mantiene responsivo bajo variaciones en la carga de trabajo. Los sistemas reactivos pueden reaccionar a cambios en la frecuencia de las peticiones incrementando o reduciendo los recursos asignados a esas peticiones.

• **Orientados a Mensajes (Message driven)**: Los Sistemas Reactivos confían en el intercambio de mensajes asíncronos para establecer límites entre componentes, lo que asegura el bajo acoplamiento, aislamiento, transparencia de ubicación y proporciona los medios para delegar errores como mensajes.

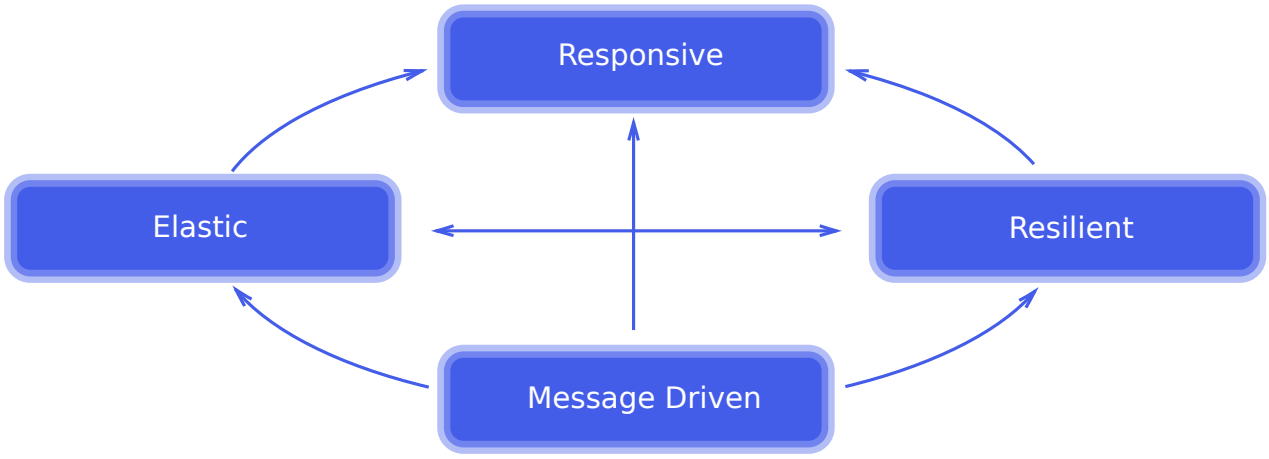


Ilustración 2: Características de los sistemas reactivos

Los *toolkits* que usaremos son Vert.x y Akka. Ambos se ejecutan sobre la máquina virtual de java (JVM) y están pensados para construir aplicaciones concurrentes y distribuidas. Vert.x está inspirado en Node.js, es manejado mediante eventos y no bloqueante. Akka está inspirado en Earlang y su implementación del modelo de actores está incluida en la librería estándar de Scala.

2. OBJETIVOS

El objetivo principal de este proyecto es construir dos aplicaciones equivalentes con dos *toolkits* distintos para posteriormente comparar su rendimiento, facilidad de desarrollo, escalabilidad y tolerancia a fallos. La aplicación consistirá en un chat en el que la comunicación entre cliente y servidor se realizará mediante websockets. El cliente consistirá en una simple web que mediante javascript enviará el nombre del usuario y el chat al que desea conectarse, creará el websocket y enviará y recibirá los mensajes. Los servidores estarán distribuidos y compararemos el rendimiento de las dos tecnologías en distintos contextos: con un único nodo, varios nodos, máquinas con 1 core, varios cores, etc. Con todo esto trataremos de determinar los inconvenientes y ventajas de cada tecnología, además de conocer en que casos sería mejor el uso de una tecnología frente a la otra. Durante el desarrollo también se tratarán de solventar los desafíos que plantean las aplicaciones distribuidas expuestos anteriormente mediante el uso de las tecnologías reactivas como son vert.x y akka.

3. TECNOLOGÍAS, HERRAMIENTAS Y METODOLOGÍA

3.1 TECNOLOGÍAS

3.1.1 VERT.X

Vertx[6][7][8] es un *toolkit* para crear aplicaciones reactivas que se ejecuta sobre la maquina virtual de java (JVM).Es similar a Node.js para JavaScript (en el que esta inspirado) o Twisted para Python. Es manejado mediante eventos y es no bloqueante. Otras de sus características son:

Ilustración 3: Logo de Vert.x

- Es políglota: Vert.x está desarrollado en Java, pero se puede utilizar con otros lenguajes como Groovy, Ruby y JavaScript.
- Tiene un modelo de concurrencia muy simple: El código puede ser escrito como el de una aplicación de un solo hilo, dejando de lado los bloques sincronizados, locks, barreras...
- Cuenta con un Event Bus: Mediante este event bus se pueden realizar las comunicaciones punto a punto o Publish-Subscribe.
- Vert.x es *open source* y está licenciado bajo la licencia Apache 2.0 y la licencia Eclipse 1.0.

Durante la realización de este proyecto utilizaremos la versión 2.1.5 de vert.x aunque actualmente este ya se encuentra en su versión 3.

3.1.1.1 CONCEPTOS BÁSICOS

En este apartado explicaremos algunos de los conceptos principales en Vert.x:

• **Verticle**: los paquetes de código que vert.x ejecuta se denominan verticles. Muchos verticles pueden ser ejecutados por una misma instancia de vert.x. Una aplicación puede estar compuesta por múltiples verticles iniciados en diferentes nodos de una red y comunicarse mediante mensajes a través del Event Bus. Estos verticles son el equivalente

los actores del modelo de actores y en vert.x su uso es opcional.

• **Event bus:** El event bus es el sistema nervioso de vert.x. Permite comunicar los verticles unos con otros independientemente del lenguaje en el que estén escritos y sin importar si se encuentran en instancias diferentes. La API del event bus es muy simple, únicamente tenemos que registrar handlers a distintos *topics* y publicar/enviar mensajes.

El event bus soporta mensajes de tres tipos:

• Publish-subscribe: El mensaje será enviado a **todos** los handlers registrados al topic especificado.

```
vertx.eventBus().publish("topic", msg);
```

• Punto a punto: El mensaje será enviado a **uno** de los handlers registrados al *topic*. Este handler será elegido mediante un algoritmo round robin no estricto.

```
vertx.eventBus().send("topic", msg);
```

• Request-reponse: Es un mensaje punto a punto al que se le pasa como parámetro un handler de respuesta. Cuando el mensaje llega al receptor, este puede opcionalmente responder al mensaje. En caso de que responda se llamará al handler pasado como parámetro en la función send.

```
vertx.eventBus().send("topic", msg, Handler<Message<String>> handler);
```

• **Instancias**: Los verticles se ejecutan dentro de una instancia. Una única instancia se ejecuta dentro de su propia JVM. Puede haber múltiples verticles ejecutándose en una misma instancia de vert.x. Se pueden ejecutar múltiples instancias en el mismo *host* al mismo tiempo.

• **Concurrencia:** Vert.x garantiza que un verticle particular nunca va a ser ejecutado por mas de un thread concurrentemente. Eso simplifica la programación con vert.x, ya que se realiza como si de una aplicación single thread se tratara. De esta manera desaparecen las condiciones de carrera y los *deadlocks* de la programación concurrente tradicional.

• **Modelo de programación asíncrona:** Vert.x provee de una serie de APIs asíncronas en su core. Esto significa que la mayoría de cosas que podemos hacer en vert.x requiere el uso de handlers. Por ejemplo, para recibir datos a través de un socket TCP, debemos crear un handler que será llamado cuando lleguen los datos. También se utilizan handlers para recibir los mensajes a través del event bus, para ser notificados de la finalización de un temporizador, etc.

• **Event loops:** Internamente, una instancia de Vert.x maneja un pequeño grupo de threads, creando el mismo numero de threads que cores tiene el host. Estos threads se llaman event loops, ya que van viendo si hay eventos y ejecutando el handler apropiado en caso de que los haya. Cuando se crea una instancia, se le asigna un event loop, que será el encargado de realizar **todo** el trabajo que genere esa instancia. Debido a que un event loop es utilizado por múltiples instancias, es muy importante no bloquear el event loop. Si necesitamos hacer llamadas bloqueantes utilizaremos un tipo especial de verticle llamado **worker verticle**.

Como nada en la aplicación es bloqueante, el event loop solo tiene que recorrer los distintos handlers en el orden en que van llegando. Debido a esto, el event loop puede manejar una enorme cantidad de eventos en un corto espacio de tiempo. Esto es conocido como el reactor pattern[40]. En la implementación estándar del **reactor pattern**, es un solo thread el que ejecuta el event loop. Sin embargo, como ya hemos comentado, en vert.x, se crean múltiples event loops, basándose en el número de cores del servidor. Esto es denominado por los creadores de vert.x como el **Multi-Reactor Pattern**.

• **APIs:** Vert.x provee de unas APIs que pueden ser llamadas directamente desde los verticles. Estas APIs están disponibles en todos los lenguajes que vert.x soporta. Las APIs de Vert.x pueden dividirse en dos:

  • Container API: es la encargada de las funciones de crear y destruir verticles, devolver la configuración de los verticles y el logging. Para crear y destruir verticles solo tendremos que llamar a las funciones deployVerticle y undeployVerticle:

```
container.deployVerticle("com/globex/app/User.java");
container.undeployVerticle(deplD.get(user));
```

Para crear el verticle solo tenemos que especificar la ruta hasta el y para eliminarlo solo hay que pasar el deploymentID, que es el identificador del verticle, como parámetro. En el apartado de descripción informática explicaremos como obtener el deploymentID de un verticle, además de otras opciones de estos métodos.

- Core API: Provee de algunas funcionalidades como:
  - Servidores y clientes de: TCP/SSL, HTTP/HTTPS y Websockets.
  - Event Bus distribuido.
  - Temporizadores y temporizadores periódicos.
  - Mapas y sets compartidos.

3.1.2 AKKA

Akka[9][10][11] es otro 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:

Ilustración 4: Logo de Akka

- 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 esta 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.
- Es open source y esta disponible bajo la licencia v.2 de Apache.

La versión utilizada de Akka durante este proyecto es la 2.4-M2, aunque actualmente ya existe la versión 2.4.0 estable.

3.1.2.1 CONCEPTOS BÁSICOS

- **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.

Para crear los actores, usaremos el siguiente método:

```
Akka.system().actorOf(Props.create(ChatManager.class), "ChatManager");
```

En este caso el actorSystem es devuelto por Akka.system(). Se invoca así debido a que es el actorSystem que crea PlayFramework (del que hablaremos posteriormente) por defecto. Pasamos como parámetros la función Props.create con el nombre de la clase y el nombre que le asignamos al actor. Props es un objeto de configuración usado para crear los actores. Es inmutable, por lo tanto es *thread-safe* y se puede compartir perfectamente.

Para eliminar un actor no necesitaremos llamar al actorSystem. Solo tendremos que enviar al actor una poisonPill y el mismo se eliminará. El propio actor también puede

enviarse a si mismo esa poisonPill.

```
self().tell(PoisonPill.getInstance(), self());
```

• **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 actor path y una actor Reference es que la actor reference tiene el mismo ciclo de vida que el actor. Si el actor se destruye su actor reference también, sin embargo un actor path puede existir perfectamente antes y después de que el actor correspondiente a ese path exista.

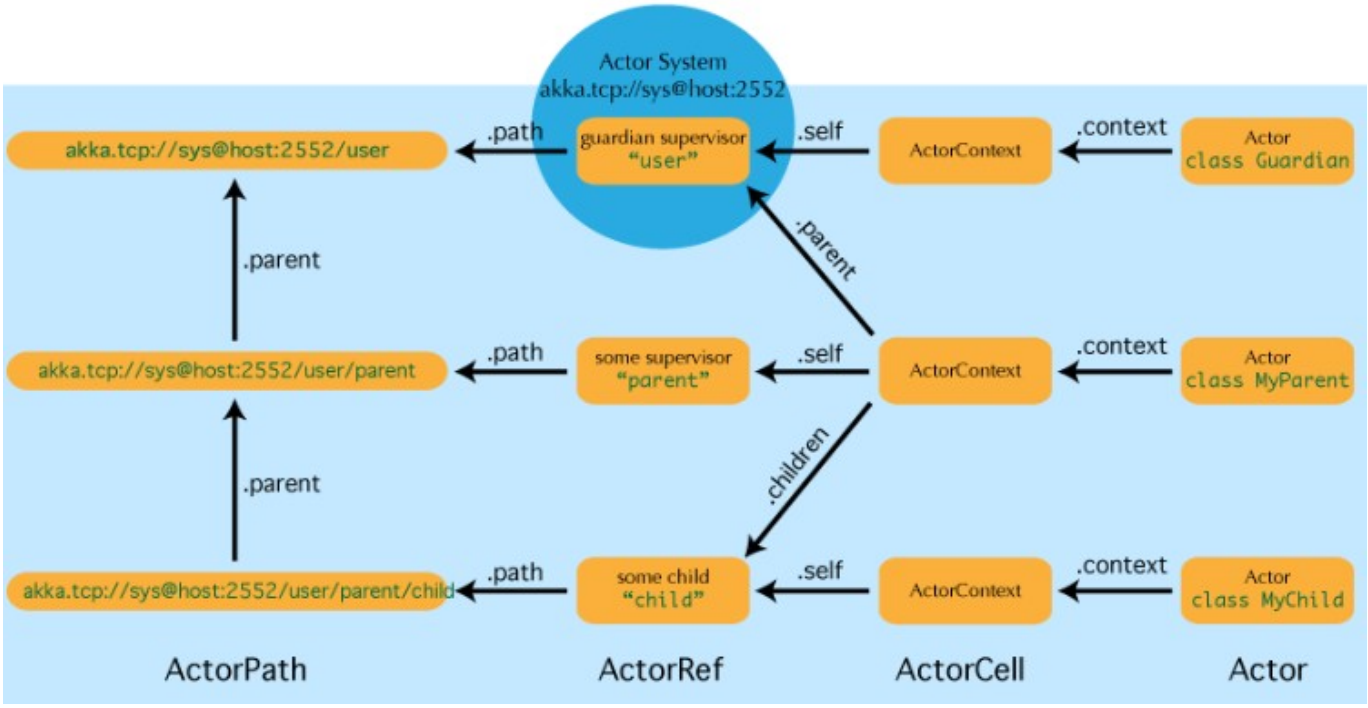


Ilustración 5: Esquema de jerarquía y paths de akka

3.1.2.2 CONCEPTOS SOBRE CLUSTERING EN AKKA

Akka cluster[30][31] provee de un servicio descentralizado, tolerante a fallos, peer-to-peer sin un único punto de fallo ni cuello de botella. Esto lo hace utilizando Vector clocks, protocolos gossip y mediante su detector automático de fallos:

• **Vector Clocks**[32]**:** Es un algoritmo que genera un orden parcial en un sistema distribuido. Estos vector clocks son utilizados para resolver las diferencias en el estado del cluster producidas por el protocolo Gossip. Un vector clock es un par contador-nodo en el que en cada actualización del estado del cluster viene acompañada con una actualización en el vector.

• **Gossip protocol**[33]: Es un protocolo inspirado en como un cotilleo (gossip) se propaga por las redes sociales, en las que un usuario propaga un rumor a sus amigos, estos a los suyos y así sucesivamente hasta que se extiende por la red. Muchos sistemas distribuidos modernos usan estos tipos de protocolos, creando un mapa de toda la red a partir de unas pocas interacciones locales. En akka, lo que se propaga es el estado actual del cluster, que es comunicado de forma aleatoria a lo largo del cluster, con preferencia por los nodos que no tienen la ultima versión.

• **Failure Detector**: Es el responsable de tratar de detectar si un nodo es inaccesible (unreachable) para el resto del cluster. Un detector de fallos preciso separa monitorización e implementación. Esto lo hace aplicable a un amplio área de escenarios y más adecuado para construir detectores de fallos genéricos. La idea es llevar un historial de los fallos, calculados mediante los hearbeats recibidos de los otros nodos, y trata de hacer conjeturas tomando múltiples factores y sus valores a lo largo del tiempo, respondiendo con un valor pi que representa la probabilidad de que el nodo esté caído.

En el cluster, cada nodo esta monitorizado por unos pocos nodos (5 como máximo por defecto) y cuando **uno** de esos nodos lo marca como inaccesible y esa información se propaga por el resto de nodos del cluster que lo marcan también como inaccesible. Sin embargo, para que el nodo sea marcado como accesible de nuevo, **todos** los nodos que lo monitorizan deben marcarlo como accesible. Una vez todos lo marquen de nuevo como accesible, el estado se propaga por el resto del cluster.

• **Líder:** El líder es el nodo encargado de cambiar el estado del resto de nodos, pero estos cambios de estado solo son ejecutados cuando se recibe el nuevo estado convergente mediante el protocolo *gossip*. Cualquier nodo puede ser el líder y este puede cambiar de una ronda de convergencia a otra. También es el encargado de pasar un nodo al estado *down* tras un determinado tiempo inaccesible.

• **Ciclo de vida:** Un nodo comienza en el estado *joining*. Cuando todos los nodos ven que el nodo se está uniendo el líder cambia su estado a *up*. Si un nodo abandona el cluster de manera esperada, cambia su estado a *leaving*. Cuando hay convergencia en todos los nodos respecto a este estado, el líder cambia su estado a *exiting*, y nuevamente, cuando todos los nodos ven ese estado, el líder lo marca como *removed*. Si un nodo es marcado como *unreachable*, no puede haber convergencia mediante el protocolo *gossip* y por lo tanto no puede haber acciones del líder, como por ejemplo permitir que otro nodo se una al cluster. Para ello el nodo *unreachable* debe volver a estar accesible de nuevo o ser marcado como *down* por el líder.

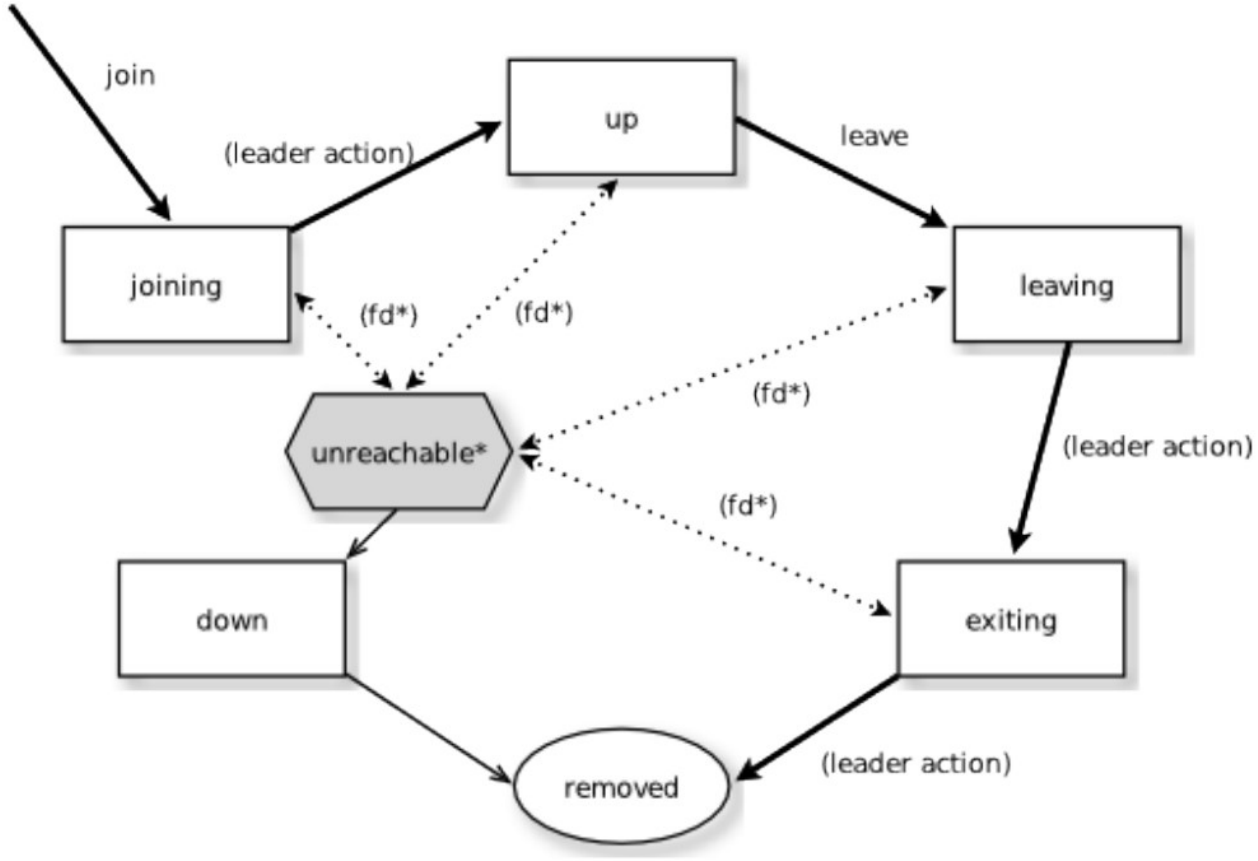


Ilustración 6: Ciclo de vida

3.1.3 PLAY FRAMEWORK

Durante el desarrollo con akka también utilizamos **Play Framework**[16], un *framework* web *open source*, escrito en java y scala y que sigue el modelo vista-controlador (mvc). Play Framework está inspirado en Ruby on Rails y Django y se encuentra dentro de la plataforma reactiva de Typesafe Inc. Compuesta por Scala, Akka, Spark y Play Framework.

Ilustración 7: Reactive platform

3.1.4 JAVA

Ilustración 8: Logo de Java

Los dos *toolkits* que vamos a utilizar y que ya hemos comentado están basados en **java**[12][13] y corren sobre la **máquina virtual de java**[13][14]. Java es un lenguaje de programación de propósito general, concurrente y orientado a objetos. Su sintaxis es similar a C y C++, pero omite muchas de las características que hacen a C y C++ complejo, confuso e inseguro. La plataforma java fue inicialmente desarrollada para solucionar los problemas del desarrollo de software para dispositivos de consumo

conectados. Fue diseñado para soportar múltiples arquitecturas y para ello se vale de la JVM, que es el componente responsable de la independencia del código java con el hardware y el sistema operativo. La JVM es una máquina virtual de proceso nativo, capaz de interpretar instrucciones expresadas en *bytecode*, el código generado por el compilador de java.

La portabilidad del lenguaje a distintas arquitecturas es una de las grandes ventajas de java, aunque también tiene sus detractores, debido a algunas características negativas del lenguaje:

• Java no es estrictamente un lenguaje orientado a objetos. Por motivos de eficiencia el paradigma de la orientación a objetos no se cumple siempre, ya que por ejemplo, no todos los valores son objetos.

• El código java puede ser redundante en comparación con otros lenguajes, debido a las frecuentes declaraciones de tipos y *castings* manuales.

Pese a todo esto, java sigue siendo uno de los lenguajes mas usados, y actualmente es el mas usado según el índice TIOBE[15]:

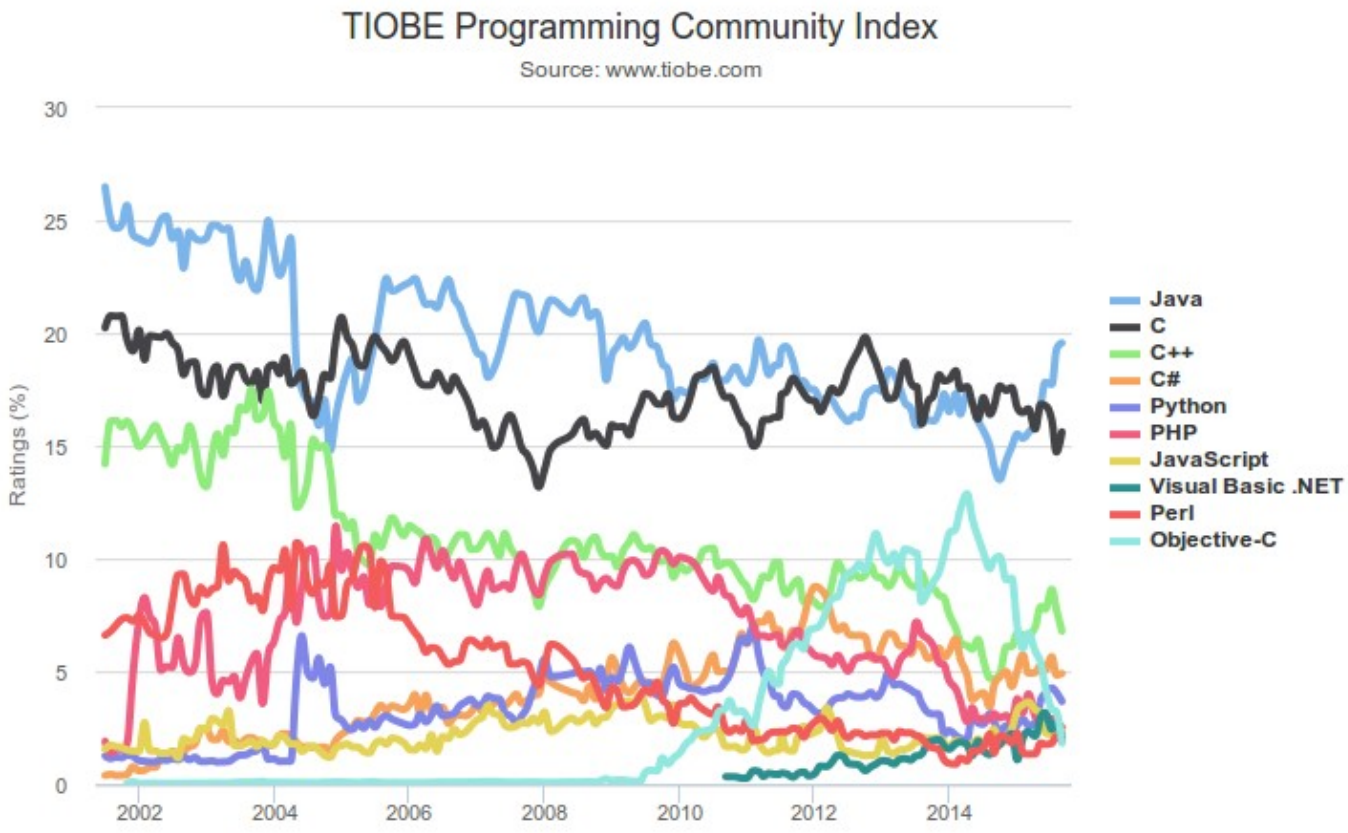


Ilustración 9: Gráfica del índice TIOBE

3.1.5. WEBSOCKETS

Definidos en el RFC 6455. los websockets[19] son una tecnología que proporciona un canal de comunicación bidireccional y *full duplex* sobre un único *socket* TCP. Ha sido diseñado para sustituir las comunicaciones bidireccionales existentes que utilizan HTTP, ya que este no fue inicialmente diseñado para comunicaciones bidireccionales.

Hay algunas diferencias[53] entre los sockets TCP convencionales y los websockets. La principal diferencia con los sockets tradicionales es que una llamada al método *send* de los websockets nunca se bloquea. Además, con websockets la recepción de los mensajes es *event-driven,* suelen registrarse handlers para la recepción de mensajes, y los datos recibidos siempre se corresponden con el mensaje completo enviado.

3.1.6. HA PROXY

Ilustración 10: Logo de HAProxy

HAProxy es un balanceador de carga de alta disponibilidad, gratuito y muy rápido. Está escrito en C y tiene la reputación de ser rápido y eficiente en términos de uso del procesador y consumo de memoria. Estas características lo han convertido en el estándar de facto dentro de los balanceadores opensource. HAProxy es utilizado por importantes sitos web como GitHub, Bitbucket, Stack Overflow, Reddit y Twitter.

3.2 HERRAMIENTAS

3.1.2 GIT

Ilustración 11: Logotipo de Git

GIT[20][21] es un software de control de versiones diseñado por Linus Torvalds, debido a que BitKeeper, el software de control de versiones usado por la comunidad para desarrollar en kernel Linux hasta 2005, dejase de

ser gratuito. GIT se diseñó basándose en BitKeeper y algunas de sus metas eran la velocidad, el soporte de un desarrollo no lineal con cientos de ramas paralelas y que fuera completamente distribuido.

En nuestro proyecto hemos usado Github, una plataforma de desarrollo colaborativo para alojar proyectos usando Git. A el hemos subido todas las versiones del proyecto.

3.1.3 MAVEN

Maven[22][23] es una herramienta software para la gestión y construcción de proyectos Java. Utiliza un *Project Object Model* (POM) para describir el proyecto a construir, sus dependencias, componentes externos y orden de construcción. Una de sus características mas importante es que puede descargar dinámicamente *plugins* de un repositorio que también provee acceso a muchas versiones de diferentes proyectos *open source*.

Ilustración 12: Logo de maven

Durante el proyecto hemos usado Maven para gestionar las dependencias de la parte de Vert.x. Aunque no eran demasiadas dependencias, el uso de Maven simplifica en gran medida tener todas las dependencias resueltas en un proyecto que se ha descargado en múltiples máquinas.

```
<dependency>
      <groupId>io.vertx</groupId>
      <artifactId>vertx-core</artifactId>
      <version>2.1.5</version>
</dependency>
<dependency>
      <groupId>io.vertx</groupId>
      <artifactId>vertx-platform</artifactId>
      <version>2.1.5</version>
</dependency>
<dependency>
      <groupId>io.vertx</groupId>
      <artifactId>vertx-hazelcast</artifactId>
      <version>2.1.5</version>
</dependency>
```

Este código se encuentra en: https://github.com/meji92/WebChatVertx/blob/master/pom.xml

3.1.4 SBT

SBT[24][25] es una herramienta para construir proyectos similar a Maven o Ant. Es el estándar de facto en la comunidad de Scala y es usado por PlayFramework. Algunas de sus características principales son:

Ilustración 13: Logo SBT

- Soporte nativo para compilar código Scala e integrarlo con muchos test frameworks.
- Las dependencias son manejadas usando Ivy, que soporta los repositorios en formato Maven.
- Soporte para proyectos mixtos de Java y Scala

Al igual que Maven, lo hemos utilizado para resolver las dependencias en la parte del proyecto con Akka.

```
libraryDependencies ++= Seq(
  javaJdbc,
  cache,
  javaWs,
  "com.typesafe.akka" %% "akka-cluster" % akkaVersion,
  "com.typesafe.akka" % "akka-cluster-tools_2.11" % akkaVersion,
  "com.typesafe.akka" % "akka-cluster-metrics_2.11" % akkaVersion
)
```

Este código se encuentra en: https://github.com/meji92/WebChatAkkaPlay/blob/master/build.sbt

3.1.5 ECLIPSE

Eclipse[26][27] es un programa informático compuesto por un conjunto de herramientas de programación de código abierto multiplataforma. Típicamente ha sido usado para desarrollar entornos de desarrollo integrados (IDE), como el IDE de Java llamado Java Development Toolkit (JDT) y el

Ilustración 14: Logo de eclipse

compilador (ECJ) que se entrega como parte de Eclipse. Eclipse es ahora desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto.

Durante el desarrollo hemos usado la versión Luna de eclipse.

3.1.6 INTELLIJ IDEA

IntelliJ IDEA[28][29] es un IDE para el desarrollo de programas informáticos. Desarrollado por JetBrains, está disponible en dos ediciones: comunity (gratuita) y ultimate (de pago). IntelliJ ofrece muchas funcionalidades para el desarrollo con *frameworks* como Java EE, Spring y Play. Además ofrece soporte para Maven, Gradle, SBT e integración con GIT, Subversion y Mercurial.

Ilustración 15: Logo de IntelliJ

Hemos utilizado dos IDEs debido a un cambio a mitad del proyecto. Se cambió eclipse por intelliJ debido a la mejor (en mi opinión) integración con git y github. Durante el desarrollo usamos la versión 14.1.5.

3.3 METODOLOGÍA

La metodología empleada para el desarrollo de este proyecto ha sido el conocido modelo de **desarrollo en espiral**[17][18] , definido por primera vez por Barry Boehm en 1986. Las actividades en este modelo se conforman en una espiral en la que cada bucle o iteración representa un conjunto de actividades. En cada vuelta o iteración tenemos que tener en cuenta:

- Los **objetivos**, es decir, la necesidad que debe cubrir el producto.
- Las **alternativas**: Las diferentes formas de conseguir los objetivos de forma exitosa.
- **Desarrollar y verificar**: Programar y probar el software.

• Si el resultado no es el adecuado o se necesitan implementar mejoras o funcionalidades se **planifican** los siguientes pasos y se comienza un nuevo ciclo de la espiral.

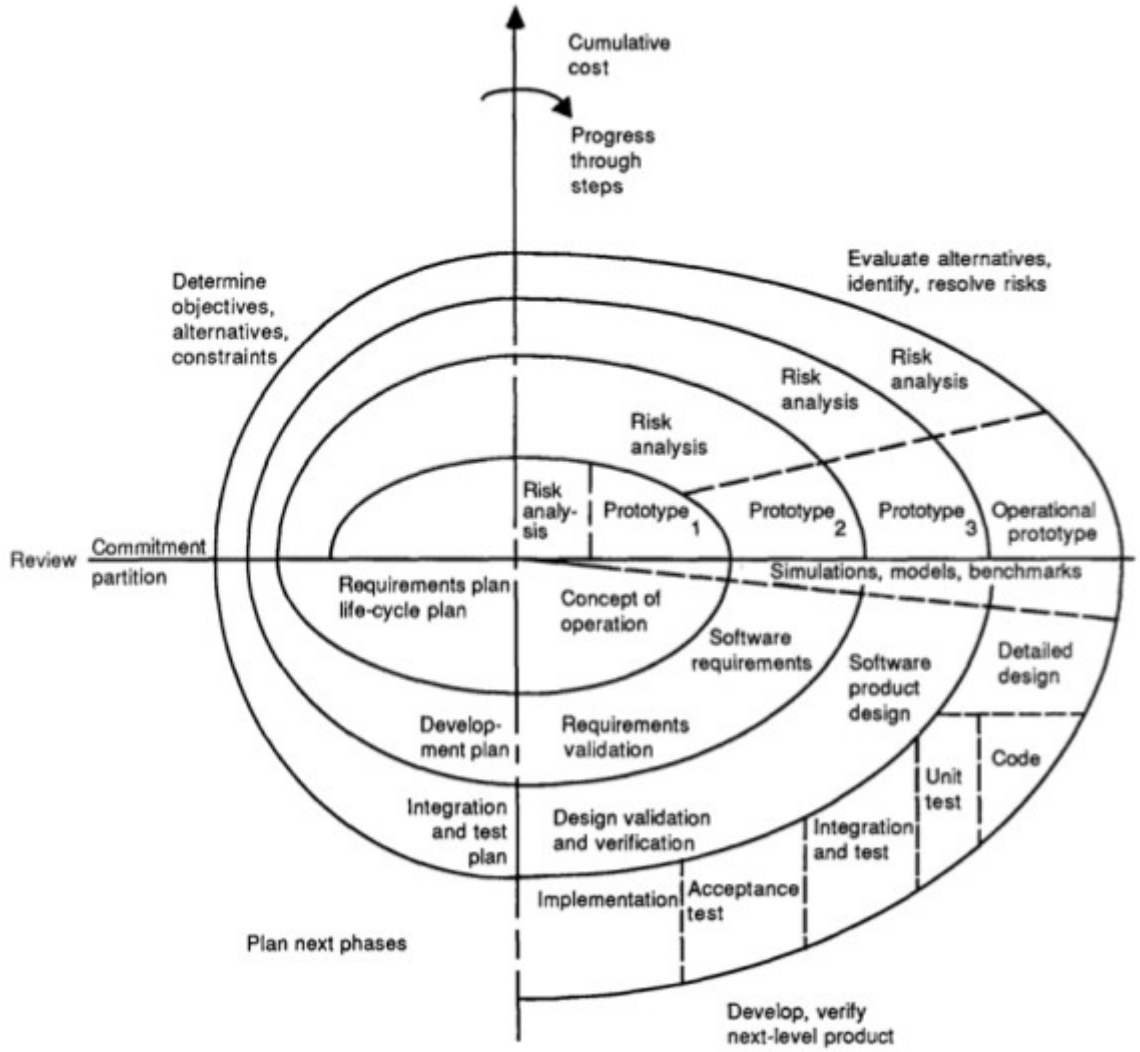


Ilustración 16: Desarrollo en espiral

La espiral tiene dos dimensiones, la angular y la radial. La angular especifica el avance del proyecto dentro de un ciclo y la radial indica el aumento del coste del proyecto, ya que cada iteración implica un mayor gasto de tiempo.

4. DESCRIPCION INFORMATICA

4.1 CHAT MULTIUSUARIO CON AKKA

Todo el código comentado en esta sección se encuentra en: https://github.com/meji92/WebChatAkkaPlay

4.1.1 CHAT MONOLÍTICO

El primer paso en el desarrollo con Akka fue crear un servidor simple que gestionase la conexión con websockets. Para ello, primero planteamos los tipos de actores que compondrán el sistema

- Un actor `User` por cada cliente conectado. Este actor *User* es el que maneja el websocket, envía y recibe los mensajes.
- Un actor `Chat` por cada chat que se cree. Este actor enviará los mensajes de los usuarios al resto de usuarios conectados a ese chat. También se encargará de rechazar a los usuarios con nombres repetidos.
- Un actor `ChatManager` por cada nodo, que se encarga de enviar las direcciones de los chats a los nuevos usuarios y en caso de que no exista ese chat, crearlo

El primer problema que surgió, es que akka no tiene una implementación para realizar conexiones con websockets, pero tampoco hubo que ir muy lejos para encontrar la solución, ya que playframework si tenía, además de otras herramientas que serán de utilidad durante el desarrollo.

La utilización de playframework para esto es muy sencilla, solo hay que modificar el archivo routes para que se llame a los distintos métodos según la dirección de la petición y crear los métodos en Application.java:

```
GET     /                           controllers.Application.index()

GET     /chat                       controllers.Application.socket()
```

Este código se encuentra en: https://github.com/meji92/WebChatAkkaPlay/blob/Monolithic/conf/routes

Index() devuelve página creada dinámicamente (posteriormente lo veremos) y socket() crea un actor para manejar el websocket:

```
public WebSocket<String> socket() {
	return WebSocket.withActor(new F.Function<ActorRef, Props>() {
		public Props apply(ActorRef out) throws Throwable {
			return
User.props(out,Akka.system().actorFor("akka://application/user/ChatManager"))
;
		}
	});
}
```

Este código se encuentra en:
https://github.com/meji92/WebChatAkkaPlay/blob/Monolithic/app/controllers/Application.java

Se crea un actor *User*, al que se le pasan las direcciones de dos actores (actorRef en akka) : out, creado por playframework y encargado de enviar los mensajes a través del websocket (Nosotros solo le enviamos los mensajes como si de cualquier otro actor se tratara) y *Chatmanager*, el actor encargado de gestionar los chats. Este segundo actor se crea al iniciar la aplicación y está siempre presente, por lo que se obtiene su actorRef a partir del path. Para que Chatmanager se cree al iniciar la aplicación, es necesario crear una clase que extienda la clase `GlobalSettings` y en su método `onStart` creamos los actores que necesitemos desde el inicio:

```
@Override
public void onStart(play.Application application) {
	super.onStart(application);
	chatManager = Akka.system().actorOf(Props.create(ChatManager.class),
"ChatManager");
}
```

Este código se encuentra en: https://github.com/meji92/WebChatAkkaPlay/blob/master/app/Global.java

Cada actor tiene un método `onRecieve(object message)`, en el que se tratan los mensajes en función de la clase que sean. Los mensajes son clases serializables con nombres que las identifican, siguiendo los ejemplos de akka, ya que así se simplifica bastante la lectura del código.

Actualmente al conectarse un cliente los actores se comunicarían de la siguiente

manera:

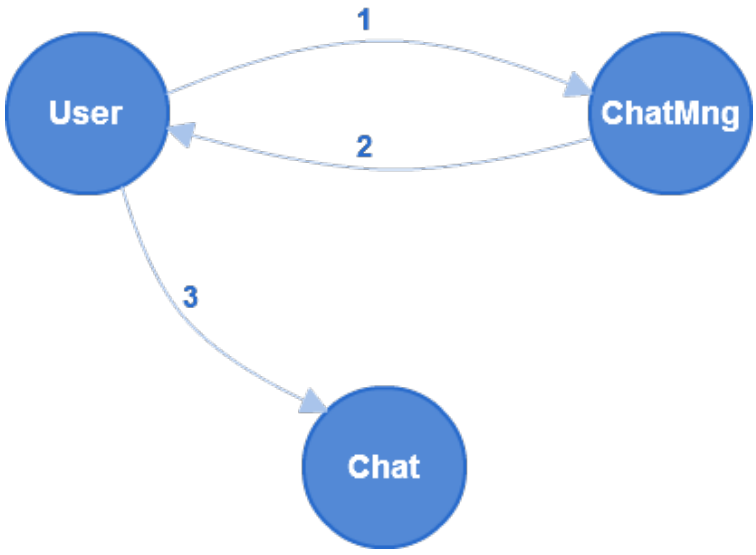


Ilustración 17: Mensajes durante la suscripción

1. Al conectarse el cliente se crea el actor User, este pide el chat con el nombre del mensaje de registro al ChatManager.

2. ChatManger devuelve su ActorRef al Usuario y si el chat no existe, ChatManager lo crea.

3. Este, al recibirlo manda un mensaje al Chat, que comprueba que no existan ya usuarios con ese nombre y lo añade a la lista de usuarios para que reciba los mensajes del chat. En caso de que ya exista el usuario, el chat manda un mensaje al User, que se lo comunica al cliente y se elimina a si mismo enviándose una PoisonPill, que es la manera que tienen los actores de eliminarse.

```
self().tell(PoisonPill.getInstance(), self());
```

Cada mensaje que recibe el user por parte del cliente, es enviado al chat, y este lo distribuye entre el resto de suscriptores.

4.1.2 CHAT DISTRIBUIDO

Hasta aquí, todo es bastante sencillo, lo que mas trabajo tiene de todo es adaptarse a los nuevos conceptos de akka y el modelo de actores. El siguiente paso es crear el chat distribuido. Para crear el chat distribuido, lo primero que hay que hacer es

añadir las siguientes dependencias al build de SBT:

```
"com.typesafe.akka" %% "akka-cluster" % akkaVersion,
"com.typesafe.akka" % "akka-cluster-tools_2.11" % akkaVersion,
"com.typesafe.akka" % "akka-cluster-metrics_2.11" % akkaVersion
```

Este código se encuentra en: https://github.com/meji92/WebChatAkkaPlay/blob/master/build.sbt

Además debemos añadir la configuración para el cluster en el fichero de configuración de play:

```
akka {
  actor {
    provider = "akka.cluster.ClusterActorRefProvider"
  }
  remote {
    log-remote-lifecycle-events = off
    netty.tcp {
      hostname = "192.168.1.204"
      port = 8000
    }
  }
  cluster {
    seed-nodes = [
      "akka.tcp://application@192.168.1.42:8000"]
    auto-down-unreachable-after = 10s
  }
}
```

Este código se encuentra en:
https://github.com/meji92/WebChatAkkaPlay/blob/master/conf/application.conf

La configuración es bastante sencilla, solo hay que aclarar lo que son los seed-nodes:

Seed Nodes: Son los puntos de contacto para los nuevos nodos que se unen al cluster. Cuando un nuevo nodo arranca, envía un mensaje a los seed nodes y después envía un mensaje para unirse al cluster al nodo que antes conteste. No es necesario que todos los seed-nodes estén funcionando, pero el nodo configurado como primer elemento en los seed-nodes deber estar funcionando al iniciar el cluster, en otro caso los otros seed-nodes no se inicializaran y no se podrán unir otros nodos al cluster. Esto es así para evitar que se formen islas separadas cuando se inicia el cluster vacío.

Después podremos suscribir un actor a los eventos del cluster si queremos que se

nos notifique de los cambios de estado en los nodos a través de mensajes:

```
cluster.subscribe(getSelf(),
ClusterEvent.initialStateAsEvents(),MemberEvent.class,
UnreachableMember.class);
```

Este código se encuentra en:
https://github.com/meji92/WebChatAkkaPlay/blob/master/app/actors/ChatManager.java

Con esto, ya tenemos los nodos conectados. Ahora queda comunicar los actores unos con otros. Los únicos mensajes que tenemos que comunicar en este ejemplo son los enviados por un cliente a otro cliente conectado en otro nodo y algún que otro mensaje para evitar los nombres de usuario repetidos. Para ello había dos maneras de plantearlo: Que los nuevos usuarios obtuviesen la dirección del chat al que se quisieran conectar, sin importar en que nodo estuviera, o que los chats estuvieran replicados en cada nodo y se comunicasen entre ellos. Puesto que en la primera opción, la caída de un nodo podía afectar a usuarios en otros nodos, optamos por la segunda opción, mejorando la tolerancia a fallos a costa de algo mas de recursos.

Para conectar los chats, optamos por utilizar el *publish-subscribe*[34] que proporciona akka, quedando la estructura del programa de la siguiente manera:

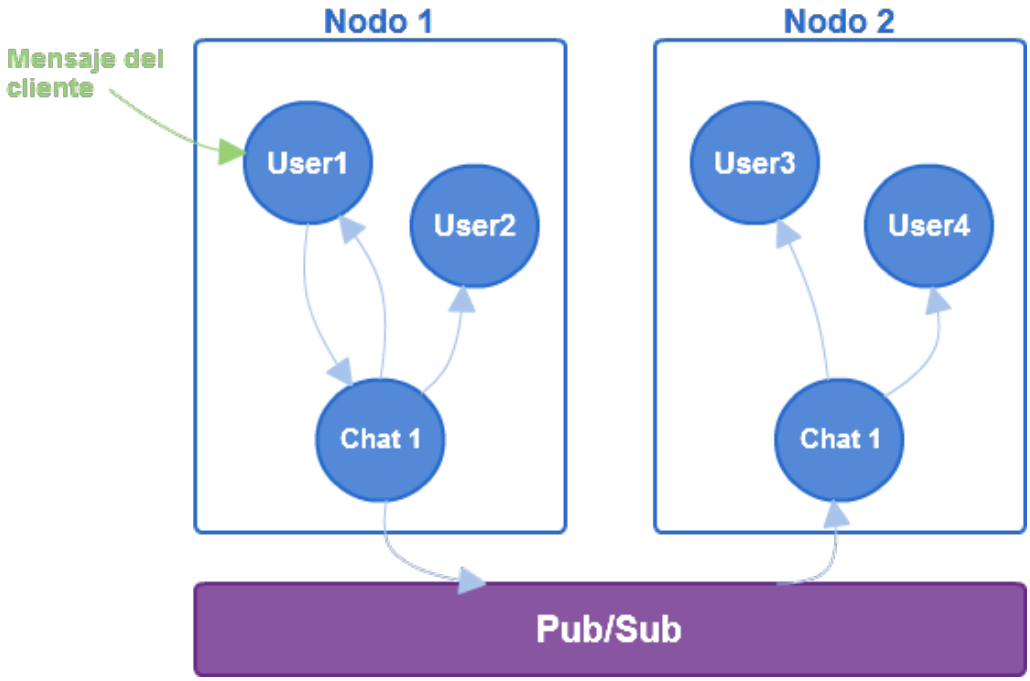


Ilustración 18: Distribución del mensaje

Al llegar un mensaje a un user, este lo envía al chat, y este a su vez lo envía a

todos sus users y al *publish-subscribe*. Del *publish-subscribe* recibirán el mensaje todos los chats con el mismo nombre y estos lo repartirán entre sus users, que lo enviarán a los clientes.

Esta utilidad se encuentra en el paquete akka-contrib o en la ultima versión de akka ("2.4-M2") y su uso es muy sencillo:

```
mediator = DistributedPubSub.get(getContext().system()).mediator();
```

Este código se encuentra en:
https://github.com/meji92/WebChatAkkaPlay/blob/master/app/actors/ChatManager.java

Solo hay que obtener el ActorRef del mediator devuelto por la llamada anterior, y luego suscribirse a lo que nos interese, en este caso el nombre del chat:

```
mediator.tell(new DistributedPubSubMediator.Subscribe(chatName, getSelf()),
getSelf());
```

Este código se encuentra en:
https://github.com/meji92/WebChatAkkaPlay/blob/master/app/actors/Chat.java

Los mensajes se envían de forma parecida:

```
mediator.tell(new DistributedPubSubMediator.Publish(chatName, message),
getSelf());
```

Este código se encuentra en:
https://github.com/meji92/WebChatAkkaPlay/blob/master/app/actors/Chat.java

Donde el primer argumento (chatName) es el tema a cuyos suscriptores se enviará el mensaje y el segundo argumento es el mensaje.

4.1.3 BALANCEO DE CARGA

En este momento, las peticiones se hacen a los nodos por un *round-robin*

mediante ha-proxy (lo veremos posteriormente) y estos devuelven una página web que crea un websocket. Para tratar de balancear la carga, la idea es que las peticiones de la web se sigan haciendo por *round-robin*, pero que la conexión permanente del websocket se haga con el nodo menos cargado. Para ello, debemos devolver una web dinámica que abra el websocket a la dirección del nodo que queremos. Para ello utilizamos el sistema de *templates* de playframework[35] . El problema viene cuando queremos pedir la dirección desde playframework a akka, ya que akka es asíncrono y la petición no se va a procesar instantáneamente. Cuando desde playframework nos comunicamos con un actor, este nos devuelve un objeto Future, que convertimos en un Promise de playframework[36]:

```
public F.Promise<Result> index() {
      ActorRef chatManager =
Akka.system().actorFor("akka://application/user/ChatManager");
      return F.Promise.wrap(ask(chatManager, "GiveMeTheChatIP",
10000)).map(
                  new F.Function<Object, Result>() {
                        public Result apply(Object response) {
                              return ok(chat.render(response+":9000");
                        }
                  }
      );
}
```

Este código se encuentra en:
https://github.com/meji92/WebChatAkkaPlay/blob/master/app/controllers/Application.java

Este es el código que devuelve la web dinámica. Devuelve el Promise generado mediante la función wrap a partir del Future devuelto por el chatManager.

Para obtener la dirección del nodo menos cargado vamos a usar los routers de akka. Un router es un actor encargado de enviar mensajes eficientemente a otros actores. Estos actores de destino se conocen como routees. Aparentemente los routers son como actores normales, pero están diseñados para ser extremadamente eficientes para recibir mensajes y enviarlos a los routees sin convertirse en cuellos de botella.

Un router puede ser de dos tipos:

- Pool: el router crea routees como actores hijos y los elimina cuando han acabado el trabajo.
- Group: Los actores routees son creados externamente y el router solo envía los

mensajes a un determinado path.

Nosotros vamos a usar el segundo tipo de router, y los actores de destino van a ser los únicos que son fijos en la aplicación: los chatManagers. Para crear un router en cluster con **balanceo de carga**[37] adaptativo necesitamos como mínimo esta configuración:

```
akka.actor.deployment {
   /ChatManager/router = {
      router = adaptive-group
      #  metrics-selector = heap
      # metrics-selector = load
      # metrics-selector = cpu
      metrics-selector = mix
      routees.paths = ["/user/ChatManager"]

      cluster {
         enabled = on
         allow-local-routees = on
      }
   }
}
```

Este código se encuentra en:
https://github.com/meji92/WebChatAkkaPlay/blob/master/conf/application.conf

Donde:

• /ChatManager/router es la dirección donde se va a crear el router

• `router = adaptative-group` es el tipo de router que se va a crear. Hay muchos tipos de router, como por ejemplo: RoundRobinGroup, RandomPool, BalancingPool... En este caso, adaptative-group selecciona el actor de destino en función de la capacidad de cada nodo.

• `metrics-selector = mix` es el tipo de MetricsSelector seleccionado. Hay varios:

  • Heap: Los pesos son calculados utilizando el heap usado y maximo de la JVM. La fórmula es (max - used)/max

  • Load: Calculado en función de la carga el pasado minuto. Este valor puede consultarse utilizando la funcion top de los sistemas Linux. Los pesos se calculan de la siguiente manera: 1 - (load/processors)

- Cpu: basado en el porcentaje de uso de la CPU. Su fórmula es 1-utilización.
- Mix: Combina heap, cpu y load.

• `routees.paths = ["user/ChatManager"]` especifica la dirección en la que se encuentran los actores destinatarios de los mensajes del router.

• `allow-local-routees=on` significa que si el nodo del router contiene la routee puede enviarse a si mismo los mensajes al igual que a otro nodo.

Una vez tenemos la configuración del router, vamos a crear el actor router, especificando que se va a crear desde el fichero de configuración y el nombre de esa configuración, en este caso "router":

```
ActorRef router = getContext().actorOf(FromConfig.getInstance().props(),
"router");
```

Este código se encuentra en:
https://github.com/meji92/WebChatAkkaPlay/blob/master/app/actors/ChatManager.java

Ahora solo tenemos que enviar los mensajes al router como si de un actor cualquiera se tratara, y este los repartirá en función de la carga:

```
router.tell(new GetIP(), getSender());
```

Ahora que tenemos el router funcionando el sistema funcionaría de la siguiente manera:

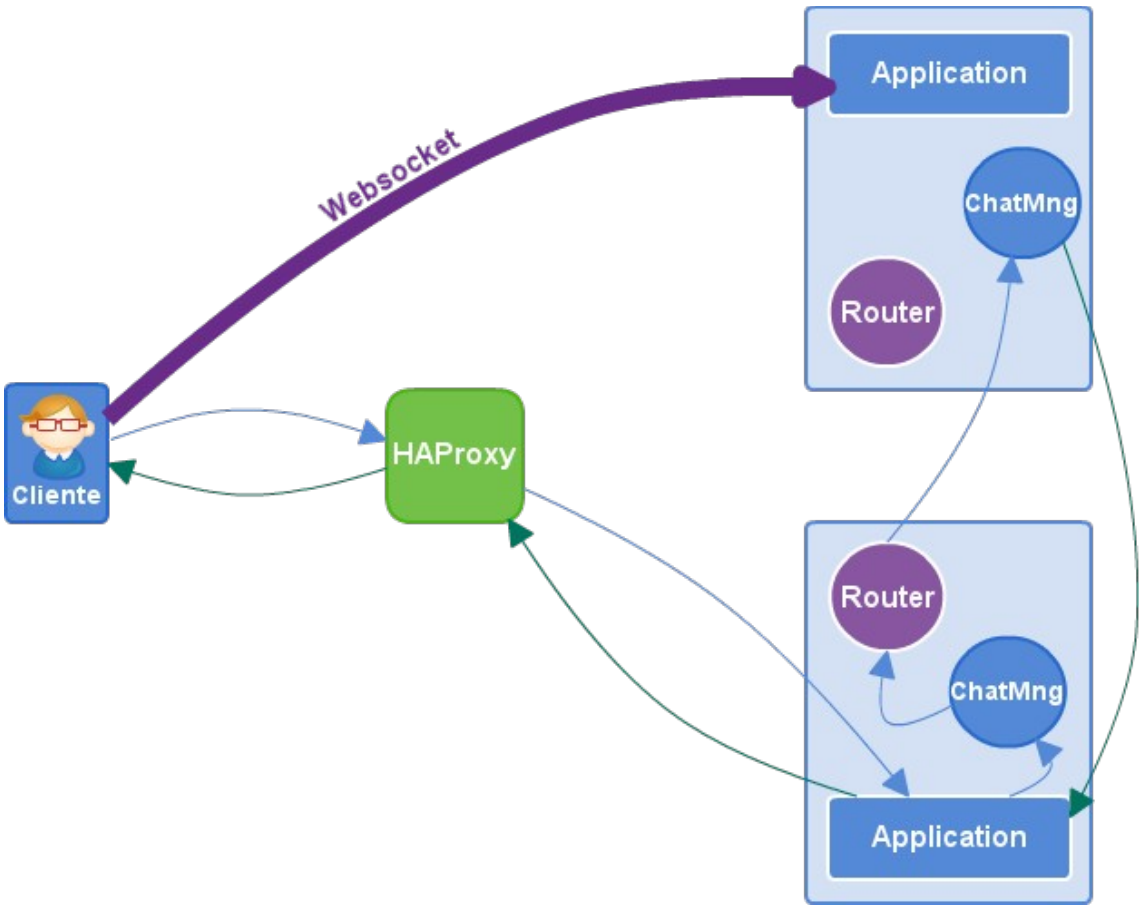


Ilustración 19: Conexión del cliente

El cliente se conecta a un nodo determinado por HAPRoxy. La aplicación de Play Framework solicita la dirección a su ChatManager y este envía al router la petición de la dirección al nodo menos cargado. El nodo menos cargado le devuelve su dirección y el primer nodo devuelve la página formada dinámicamente con la dirección del nodo menos cargado. Al recibir la página, el cliente abre el websocket con el nodo menos cargado.

4.1.4 RECONEXIÓN DEL CLIENTE

Solo nos queda un apartado por ver en el desarrollo con akka. ¿Qué es lo que ocurre con los clientes en caso de que el nodo al que están conectados se caiga? En este momento, la conexión se cortaría y el cliente debería conectarse manualmente a otro nodo recargando la página. Para evitar esto, tenemos que modificar el cliente para que en caso de cierre del websocket, se reconecte a otro servidor. Para ello, también tenemos que modificar el servidor, para que devuelva la página con el resto de direcciones de los nodos del cluster. Cuando el websocket se cierra, se hace una

conexión con el siguiente nodo de la lista:

```
websocket.onclose = function(ev) {
    websocket.close();
    if (ips.length == index){
        index = 0;
    }
    var dir= "ws://"+ips[index]+":9000/chat";
    connection(user, chat, dir, index+1);
    $('#message_box')
        .append(
                "<div class=\"system_msg\">Reconecting...</div>");
};
```

Este código se encuentra en:
https://github.com/meji92/WebChatAkkaPlay/blob/master/app/views/chat.scala.html

En el servidor, solamente tenemos que mantener actualizada una lista con los servidores que están activos. Como vimos anteriormente, podemos suscribirnos a los eventos en el cluster, y gracias a estos podemos conocer que servidores se agregan y eliminan del cluster:

```
if (message instanceof MemberUp) {
      MemberUp mUp = (MemberUp) message;
      log.info("Member is Up: {}", mUp.member());
      if (!ips.contains("\""+mUp.member().address().host().get().toString()
+"\"")) {
            ips.add("\"" + mUp.member().address().host().get().toString() + "\"");
      }
      System.out.println(ips.toString());


if (message instanceof MemberRemoved) {
      MemberRemoved mRemoved = (MemberRemoved) message;
      log.info("Member is Removed: {}", mRemoved.member());
      ips.remove("\""+mRemoved.member().address().host().get().toString()
+"\"");
```

Este código se encuentra en:
https://github.com/meji92/WebChatAkkaPlay/blob/master/app/actors/ChatManager.java

4.1.5 PROBLEMAS ENCONTRADOS

Algunos de los problemas encontrados han sido los siguientes:

• El chatmanager, encargado del clustering, no se iniciaba hasta que llegaba la primera petición, por lo que era en ese momento cuando se iniciaba la conexión con el resto de nodos, retrasando mucho la respuesta de las primeras peticiones. Para que chatmanager se iniciase al arrancar la aplicación, hemos tenido que crear un objeto global que extiende GlobalSettings de playFramework, y añadir la creación del chatmanager al método onStart:

```
@Override
public void onStart(play.Application application) {
      super.onStart(application);
      chatManager = Akka.system().actorOf(Props.create(ChatManager.class),
"ChatManager");
}
```

Este código se encuentra en: https://github.com/meji92/WebChatAkkaPlay/blob/master/app/Global.java

• Otro de los problemas que encontramos estaba relacionado con el *pub-sub* de akka. Este lo utilizamos, a parte de para compartir los mensajes entre los chats, para consultar al resto de chats si ya tienen registrado un cliente con el mismo nombre que estamos registrando nosotros. Al crear un actor chat se creaba el objeto mediator para el *publish-subscribe*, pero este tardaba un tiempo en crearse, y en ese tiempo, el un nuevo cliente podía conectarse al chat recién creado sin que este consultase al resto de chats si existía un usuario con el mismo nombre, por lo que podían repetirse nombres de usuario en un mismo chat. Para evitarlo, lo único que hubo que hacer fue crear el objeto mediator en el chatManager en vez de hacerlo individualmente en cada chat, y pasar su `ActorRef` como parámetro a cada chat:

```
Akka.system().actorOf(Chat.props(((GetChat) message).getChatname(),
mediator)
```

Este código se encuentra en:
https://github.com/meji92/WebChatAkkaPlay/blob/master/app/actors/ChatManager.java

• El ultimo problema encontrado apareció realizando las pruebas. Al conectarse el cliente de pruebas (que posteriormente explicaremos), este empezaba a enviar mensajes

en cuanto se abría el websocket, pero en ese momento el chatManager todavía no le había devuelto el Chat al User, por lo tanto daba un error ya que el objeto chat era igual a `null`. Para resolverlo, ha habido que añadir una lista a la que se añaden los mensajes mientras chat es igual a `null`, y cuando llega el mensaje de ChatManager con la ActorRef del chat, se le envían todos los mensajes almacenados en la lista.

4.2 CHAT MULTIUSUARIO CON VERT.X

Todo el código comentado en esta sección se encuentra en:

https://github.com/meji92/WebChatVertx

https://github.com/meji92/WebChatVertxMonolithic

4.2.1 CHAT MONOLÍTICO

La primera versión del chat consistirá en una clase principal, llamada ChatManager, en la que se creará el servidor web y de websockets. Por cada cliente que se conecte, se creará un verticle User, que estará suscrito en el event bus al nombre del chat en el que se encuentre.

La idea es sencilla, solo hay un problema, la manera en la que se crea el servidor de websockets:

```
vertx.createHttpServer().websocketHandler(new
Handler<ServerWebSocket>() {
      public void handle(final ServerWebSocket ws) {
         if (ws.path().equals("/chat")) {
            ws.dataHandler(new Handler<Buffer>() {
               public void handle(Buffer data) {
                  //Manejo de los mensajes recibidos a través del websocket.
               }
            });
         } else {
            ws.reject();
         }
      }
   }).requestHandler(new Handler<HttpServerRequest>() {

   public void handle(HttpServerRequest req) {
```

```
        if (req.path().equals("/"))
req.response().sendFile("com/globex/resources/index.html"); // Serve the html
      }
   }).listen(9000);
}
```

Este código se encuentra en:
https://github.com/meji92/WebChatVertxMonolithic/blob/master/src/main/java/com/globex/app/Websockets.java

Este código es el que abre el websocket. Este código se encuentra en la clase ChatManager. La idea inicial era que el User enviara directamente los mensajes, pero el objeto de la clase ServerWebSocket a través del que se tienen que enviar los mensajes no puede ser enviado al User a través del event bus. Por lo tanto habrá que crear un handler que escuche los mensajes que le llegan al user y los envíe a través del websocket. Por lo tanto el esquema de los mensajes quedaría de la siguiente manera:

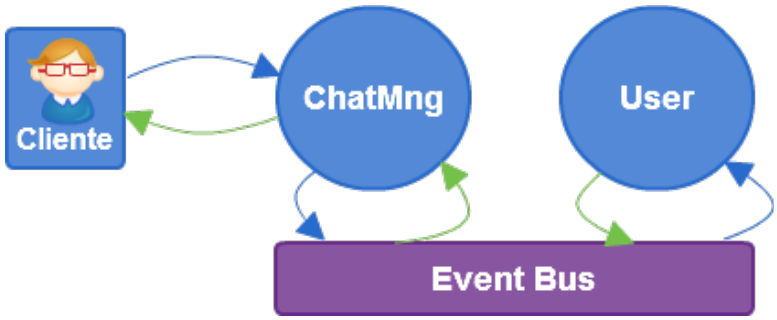


Ilustración 20: Distribución del mensaje

Al recibir un mensaje a través del websocket, el ChatManager lo publica en el event bus con el nombre del chat que corresponda. Los Users suscritos lo reciben y envían el mensaje formado de vuelta a sus respectivos ChatManagers para que lo envíen a través del websocket al cliente.

Para crear los verticles de usuario, debemos llamar a la siguiente función:

```
container.deployVerticle("com/globex/app/User.java", config,
newVerticleUser(ws,config));
```

Este código se encuentra en:
https://github.com/meji92/WebChatVertxMonolithic/blob/master/src/main/java/com/globex/app/Webs

ockets.java

En la llamada, se especifica, primero, la dirección del código del verticle que queremos crear. El segundo parámetro es un `JsonObject` mediante el cual podemos pasar parámetros al verticle recién creado. Por ultimo, y de manera opcional, podemos incluir un handler que se ejecute cuando el verticle esté creado. Esto nos será útil para eliminar el verticle. Para hacerlo, tenemos que llamar a la siguiente función:

```
container.undeployVerticle(depID);
```

Este código se encuentra en:
https://github.com/meji92/WebChatVertxMonolithic/blob/master/src/main/java/com/globex/app/Websockets.java

El deployment ID es el identificador del verticle. Este se puede obtener de la siguiente manera[41]:

Asignamos un handler en la creación del verticle, y cuando esta se complete nos devolverá el deployment ID y podremos guardarlo para el posterior borrado del verticle:

```
private AsyncResultHandler<String> newVerticleUser(final ServerWebSocket
ws, final JsonObject config){

   AsyncResultHandler<String> handler = new AsyncResultHandler<String>()
{
        public void handle(AsyncResult<String> asyncResult) {
            if (asyncResult.succeeded()) {
              //Save the deploymentID to later remove the verticle
                depID.put(config.getString("name"), asyncResult.result());

               //A new handler to send the user messages through the
websocket
            Handler<Message<String>> userHandler = newUserHandler(ws,
config.getString("name"));
            vertx.eventBus().registerHandler(config.getString("name"),
userHandler);
            AskNodesForUser(config.getString("name"),ws, userHandler);
            } else {
                asyncResult.cause().printStackTrace();
            }
        }
        };
```

```
        return handler;
    }
```

Este código se encuentra en:
https://github.com/meji92/WebChatVertxMonolithic/blob/master/src/main/java/com/globex/app/Websockets.java

Además del deploymentID, deberemos almacenar otra serie de datos. Los mensajes habituales enviados por el cliente solo traen consigo 2 datos: el usuario y el propio mensaje. Por lo tanto, necesitamos almacenar el chat al que se envían los mensajes de dicho usuario. Este chat se envía en el primer mensaje del cliente junto con su nombre de usuario para registrarse. En este momento almacenaremos su nombre de usuario y le asignaremos un color para sus mensajes que también almacenaremos. Todos estos datos se almacenarán en mapas en los que el user será siempre la clave. Estos mapas también los usaremos al registrar a un usuario para comprobar si ya existe otro usuario con el mismo nombre.

4.2.2 CHAT DISTRIBUIDO

Para hacer que el chat funcione de manera distribuida, es tan sencillo como añadir el *flag* -cluster al ejecutar el programa. De esta manera, el event bus se compartirá entre todos los nodos en los que se ejecute la aplicación, por lo tanto, todos los mensajes que enviamos serán recibidos por todos los usuarios. Solo hay un pequeño detalle que debemos corregir: Los mapas con los usuarios no están compartidos, por lo que podrían conectarse dos clientes con el mismo nombre de usuario. Para evitar esto, cada vez que añadamos un usuario, vamos a preguntar al resto de nodos si tienen en sus mapas dicho usuario, y en caso de que lo tengan, nos enviarán un mensaje para que eliminemos al usuario.

```
private Handler<Message<JsonObject>> newUserQueryHandler (){
    Handler<Message<JsonObject>> handler = new
Handler<Message<JsonObject>>(){
        public void handle(Message<JsonObject> arg0) {
            if ((users.containsKey(arg0.body().getString("user")))&&(!
vertx.currentContext().toString().equals(arg0.body().getString("context")))){
                vertx.eventBus().publish(arg0.body().getString("user")+"?",
arg0.body().getString("context"));
            }
```

```
        }
      };
      return handler;
  }


private void AskNodesForUser(final String user, final ServerWebSocket ws, final
Handler<Message<String>> handler) {

      final Handler<Message<String>> replyHandler = new
Handler<Message<String>>(){
          public void handle(Message<String> arg0) {
              if (arg0.body().equals(vertx.currentContext().toString())){
                  ws.writeTextFrame(newMessageDuplicatedUser());
                  deleteUser(user,handler,ws);
                  vertx.eventBus().unregisterHandler(user+"?", this);
              }
          }
      };

      vertx.eventBus().registerHandler(user+"?", replyHandler,new
AsyncResultHandler<Void>(){
          public void handle(AsyncResult<Void> arg0) {
              JsonObject msg = new JsonObject();
              msg.putString("user", user);
              msg.putString("context", vertx.currentContext().toString());
              vertx.eventBus().publish("user??", msg);
          }
      });

      vertx.setTimer(10000, new Handler<Long>() {
                  public void handle(Long arg0) {

vertx.eventBus().unregisterHandler(user+"?",replyHandler);
                  }
              }
      );
    }
```

Este código se encuentra en:
https://github.com/meji92/WebChatVertx/blob/master/src/main/java/com/globex/app/ChatManager.java

El primer método, `newUserQueryHandler`, devuelve un handler que será el encargado de responder a las consultas sobre los nombres de usuario en caso de que

exista en los mapas de ese nodo. Se registra nada mas arrancar el verticle:

```
vertx.eventBus().registerHandler("user??", newUserQueryHandler());
```

Este código se encuentra en:
https://github.com/meji92/WebChatVertx/blob/master/src/main/java/com/globex/app/ChatManager.java

El segundo método, `AskNodesForUser` está dividido en tres partes. En la primera, se crea el handler que manejará la respuesta en caso de que otro nodo tenga ya ese nombre de usuario. En la segunda, se registra el handler y se crea otro handler que enviará la consulta por el event bus. Este segundo handler es necesario porque puede que se envíe la consulta y llegue la respuesta sin que el handler esté aún registrado. Lo ideal sería añadir el handler en el *publish* para manejar la respuesta, al igual que se hace con los mensajes de *request-reponse*, pero al ser un *publish* no da esa opción. La ultima parte es un temporizador para eliminar el handler pasados 10 segundos. Este método es llamado después de agregar el usuario al mapa local, ya que si lo hiciéramos antes, en caso de conectarse simultáneamente dos clientes con el mismo nombre a dos nodos distintos, los dos se agregarían, ya que en el momento de la pregunta no estarían en los mapas. De esta manera, en el peor de los casos, es decir, con dos clientes conectándose simultáneamente como el caso anterior se rechazarían los dos, pero en ningún momento podríamos conectar mas de un cliente con un mismo nombre.

Todo esto es necesario debido a que el chat ha sido desarrollado con vert.x 2.1.5, en el que solo existen mapas compartidos entre instancias en una misma máquina. Actualmente, en vert.x 3, existen mapas compartidos por todos los nodos del cluster que simplificarían mas aún el desarrollo de esta aplicación[42].

4.2.3 OPCIONES DE EJECUCIÓN

Al poner en funcionamiento nuestro proyecto, vert.x nos da una serie de opciones interesantes que merece la pena comentar:

- **Server Sharing**[43]: Al ejecutar nuestra aplicación, podemos ejecutar mas de una

instancia simultáneamente:

```
vertx run io.vertx.examples.http.sharing.HttpServerVerticle
-instances 2
```

Al ser los dos verticles iguales, ambos van a escuchar al mismo puerto, lo que en principio debería lanzar una excepción, pero vert.x maneja esto por nosotros, y al ejecutar varios servidores en una misma máquina, vert.x se encarga de repartir los mensajes entre las diferentes instancias siguiendo la estrategia del round robin.

• High availability[44]: Los verticles pueden ejecutarse en alta disponibilidad de la siguiente manera:

```
vertx run my-verticle.js -ha
```

De esta manera, cuando una instancia se cae por cualquier razón, es reiniciada en otro nodo del cluster. Además, podemos inicializar instancias de vert.x vacías, es decir, que no están ejecutando ningún verticle, en algunos nodos del cluster, solo para que reinicien las instancias caídas:

```
vertx run -ha
```

La primera opción si nos será útil para escalar verticalmente nuestro servidor, sin embargo la segunda opción, en nuestro caso, no resulta de gran utilidad, aunque nos pareció interesante comentarla ya que puede tener mucha utilidad en otro tipo de desarrollos.

4.3 COMPARATIVA ENTRE AKKA Y VERT.X

En este apartado vamos a comparar el desarrollo con ambos *toolktis*. En cuanto a la facilidad de desarrollo, tenemos un claro vencedor. Vert.x es mucho mas sencillo que Akka, por lo que empezar a desarrollar con el no requiere prácticamente ningún conocimiento previo sobre la programación con actores. La principal dificultad es aprender a usar handlers (en mi caso, ya que no los había usado), una vez sepamos eso utilizar vert.x no supondrá mayor problema ya que la creación de verticles y la comunicación a través del event bus es muy sencilla. Además, comunicar varios nodos es

automático, solo añadiendo un *flag* en la ejecución vert.x se encarga de todo. Sin embargo, Akka requiere un mayor aprendizaje, los actores, actorSystems, paths, ActorRefs, etc son conceptos que hay que tener claros antes de empezar a programar, además que el clustering, sin ser demasiado complicado, si que requiere algo más de esfuerzo que en vert.x.

En cuanto a la documentación, Akka es superior a Vert.x. Su documentación es más completa y mucho mas detallada. Además existe una mayor comunidad de usuarios de Akka, por lo que encontrar información en internet es bastante fácil, aún más si no nos importa que el código esté en Scala. En vert.x la documentación es más simple, quizás mas orientada a realizar un desarrollo rápido, pero en ocasiones insuficiente. En cuanto a la comunidad, es menor a la de Akka, aunque aparecen bastantes dudas resueltas en los grupos de Google, muchas de ellas resueltas por los propios creadores de vert.x

4.4 HAPROXY

Para repartir las peticiones entre los distintos nodos vamos a utilizar **HAProxy**[45]. La instalación y configuración de HAProxy es muy sencilla. Vamos a explicar un poco la configuración de un balanceador en HAProxy[46]:

```
listen cluster 0.0.0.0:80
    mode http
    stats enable
    balance roundrobin
    option forwardfor
    server vm1 192.168.1.69:9000 check
    server vm2 192.168.1.145:9000 check
```

Este código se encuentra en:

El balanceador se especifica con listen mas el nombre que elijamos, la dirección y el puerto al que va a escuchar. Si especificamos la dirección 0.0.0.0:80 se escucharán todas las peticiones por el puerto 80. Se pueden incluir varios balanceadores en el archivo de configuración. La opción `mode http` especifica en que nivel trabaja el balanceador:

- http: El balanceador trabaja en HTTP. Las peticiones del cliente serán analizadas de una

manera profunda antes de conectar al servidor. Esto permite el filtrado, procesado y enrutado de los mensajes.

• tcp: El balanceador trabaja en TCP. Se establece una conexión full-duplex entre cliente y servidor. Este modo no nos proporciona ninguna de las ventajas de un balanceador de nivel 7 en el modelo OSI.

`Stats enable` habilita las estadísticas, que pueden ser consultadas en una dirección especificada en la configuración global.

`Balance roundrobin` especifica el algoritmo que se va a utilizar para balancear la carga. Se pueden configurar distintos algoritmos como:

• roundrobin: Cada servidor es utilizado un turno. Se pueden asignar pesos a los servidores para que se envíen mas mensajes a unos que a otros.

• static-rr: Es un round-robin sin pesos.

• leastconn: El servidor con menor numero de conexiones es el que recibe la conexión.

• uri: Toma parte de la uri de la petición, la divide y selecciona el servidor en función de unos pesos. El objetivo es que el mismo servidor maneje las peticiones iguales.

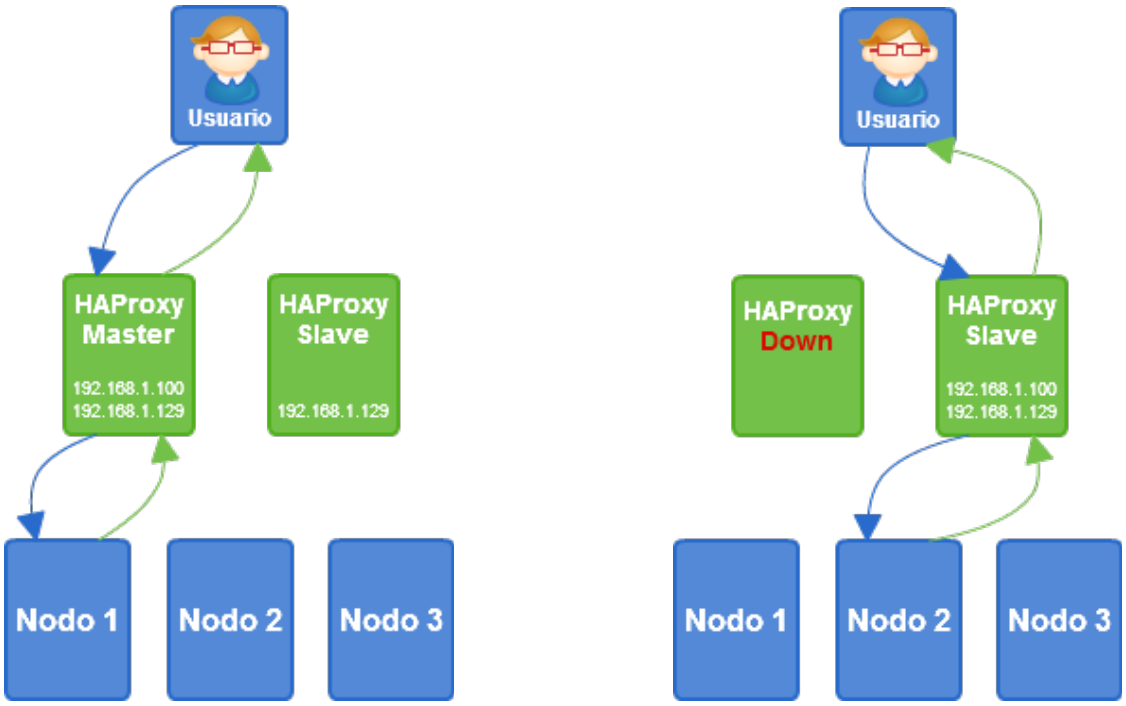
La opción `forwardfor` activa la inserción de la cabecera X-Forwarded-For a los mensajes enviados al servidor. Esta cabecera contiene la ip del cliente que hace la petición a HAProxy.

Con `server` se especifica el nombre y la dirección de cada uno de los servidores. La opción `check` permite que HAProxy compruebe si un servidor está caído. El servidor se considera disponible cuando acepta periódicamente conexiones TCP.

4.4.1 TOLERANCIA A FALLOS

Una vez configurado HAProxy podemos ir al siguiente paso. En este momento, si el nodo en el que se encuentra HAProxy se viniera abajo, todo el sistema quedaría inutilizado. Para evitarlo vamos levantar dos nodos con HAProxy. Para hacerlo vamos a utilizar **Keepalived**[47][48], un software de enrutado escrito en C. Keepalived implementa el **protocolo VRRP**[49][50] o **Virtual Router Redundancy Protocol**. Se trata de un

protocolo de redundancia no propietario definido en el RFC 3768[51]. Este protocolo nos permite crear un router virtual como única puerta de enlace y dos o más routers físicos detrás que son quien realmente los que realizan el enrutamiento. De esta manera podemos tener dos nodos con HAProxy, un maestro y un esclavo, compartiendo la misma dirección, y en caso de que el nodo maestro caiga, el esclavo seguirá manejando el tráfico.



La configuración de keepalived sería la siguiente:

```
global_defs {
        lvs_id haproxy_DH
}

vrrp_script check_haproxy {
        script "killall -0 haproxy"
        interval 2
        weight 2
}

vrrp_instance VI_01 {
        state MASTER
        interface eth0
        virtual_router_id 51
        priority 101
```

```
virtual_ipaddress {
        192.168.6.164
}
track_script {
        check_haproxy
}
}
```

Este código se encuentra en:

Primero definimos el identificador del proceso de Keepalived. Lo siguiente es definir el script que comprueba si HAProxy esta funcionando. Lo siguiente es definir el rol del nodo, la interfaz, el id del router, prioridad y la IP virtual que van a compartir los dos nodos.

En el nodo esclavo solo habría que cambiar un par de cosas:

- Es un nodo esclavo, por lo tanto:

```
state SLAVE
```

- La prioridad, que debe ser menor:

```
priority 100
```

Tras realizar esto, reiniciamos keepalived y haproxy y ya estaría funcionando.

4.5 CLOUD

Para realizar estas pruebas en cloud hemos elegido el amazon EC2. Para ejecutar nuestra aplicación en amazon hemos tenido que realizar algunos cambios. En akka, al solicitar la ip a un nodo, este devolvía su ip privada, pero para realizar la conexión en amazon necesitamos su ip pública. Por suerte en amazon nos proporcionan la manera de obtener la ip pública realizando una petición a http://instance-data/latest/meta-data/public-ipv4

```
private String getAmazonIP(){
    String dir = "ec2-";
```

```
        String ip = getEC2InstancePublicIP();
        ip = ip.replace(".","-");
        dir = dir + ip;
        dir = dir + ".eu-west-1.compute.amazonaws.com";
        return dir;
}

public String getEC2InstancePublicIP(){
        URL url = null;
        URLConnection conn = null;
        Scanner s = null;
        try {
                //url = new URL("http://169.254.169.254/latest/meta-data/instance-
id");
                url = new URL("http://instance-data/latest/meta-data/public-ipv4");
                conn = url.openConnection();
                s = new Scanner(conn.getInputStream());
        } catch (IOException e) {
                e.printStackTrace();
        }
        String aux = null;
        if (s.hasNext()) {
                aux = s.next();
        }
        return aux;
}
```

Este código se encuentra en:
https://github.com/meji92/WebChatAkkaPlay/blob/AmazonEC2/app/actors/ChatManager.java

Manipulando la ip que nos devuelven y añadiendo nuestra región obtenemos la dirección a nuestro servidor.

En vert.x tuvimos un problema, y es que amazon EC2 no soporta multicasting, y este es utilizado por la librería hazelcast incluida en vert.x para descubrir nuevos nodos. Debido a esto hubo que cambiar la configuración de hazelcast (vert.x/conf/cluster.xml) para descubrir los nodos mediante TCP/IP, especificando la dirección de uno de los nodos para que todos se unieran a él y se descubrieran unos a otros.

```
<multicast enabled="false">
        <multicast-group>224.2.2.3</multicast-group>
        <multicast-port>54327</multicast-port>
```

```
</multicast>
<tcp-ip enabled="true">
        <interface>172.31.19.187</interface>
</tcp-ip>
```

Este código se encuentra en:
https://github.com/meji92/ExtrasTFG/blob/master/Configuraciones/cluster.xml

Además de esto, debido a las limitaciones de la capa gratuita de amazon EC2, era necesario arrancar y parar las instancias habitualmente, por lo que las direcciones ip cambiaban. Para resolver esto, hubo que preparar unos scripts para modificar algunos archivos:

```
#!/bin/bash
cp /home/ubuntu/play-java-1.0-SNAPSHOT/conf/application.conf.original
/home/ubuntu/play-java-1.0-SNAPSHOT/conf/application.conf
IP=$(curl http://169.254.169.254/latest/meta-data/local-ipv4)
sed -i "s/localhost/$IP/g" /home/ubuntu/play-java-1.0-
SNAPSHOT/conf/application.conf
rm /home/ubuntu/play-java-1.0-SNAPSHOT/RUNNING_PID
#echo $IP
cd /home/ubuntu/play-java-1.0-SNAPSHOT/bin
./play-java
```

Este código se encuentra en: https://github.com/meji92/ExtrasTFG/blob/master/Scripts/startPlay

```
#!/bin/bash
cd /home/ubuntu/WebChatVertx/src/main/java/
IP=$(curl -L http://169.254.169.254/latest/meta-data/public-ipv4)
cp com/globex/resources/index.html.original com/globex/resources/index.html
IP2=${IP//./-}
#echo $IP
#echo $IP2
IPfinal="ec2-$IP2.eu-west-1.compute.amazonaws.com"
#echo $IPfinal
sed -i "s/localhost/$IPfinal/g" com/globex/resources/index.html
PATH=$PATH:/home/ubuntu/vert.x-2.1.5/bin
export VERTX_OPTS="-Xmx800m"
vertx run com/globex/app/ChatManager.java -cluster
```

Este código se encuentra en: https://github.com/meji92/ExtrasTFG/blob/master/Scripts/startVertx

5. MEDIDAS DE RENDIMIENTO

5.1 CHAT EN UNA SOLA MAQUINA - ESCALADO VERTICAL

La primera prueba a realizar fue la ejecución de la aplicación en una sola máquina virtual. A esta máquina virtual se conectaban X clientes y cada uno de ellos mandaba un total de 500 mensajes en 5 segundos. Estos clientes estaban a la escucha y cuando recibían el total de mensajes (500 x $N_{usuarios}^{2}$) se daba por concluido el test y se calculaba el tiempo medio de los mensajes. Para realizar las pruebas y crear estos clientes hemos utilizado Junit y las testTools de vert.x[54]. Este cliente envía mensajes periódicamente y los que recibe los almacena en un array para al finalizar comprobar que se han recibido todos los mensajes. Para medir el tiempo que tardan los mensajes en recibirse, el cliente que envía el mensaje adjunta en el cuerpo del mensaje el tiempo en ese momento además del número que identifica al mensaje:

```
json2.putString("message", Integer.toString(sentMessages.getAndAdd(1))
+"/"+System.currentTimeMillis());
```

Este código se encuentra en:
https://github.com/meji92/WebsocketChatTest/blob/master/src/test/java/ChatTest.java

Cuando el mensaje llega, el receptor calcula la diferencia entre el tiempo actual y el del mensaje y acumula el resultado para calcular la media.

Para comprobar el escalado vertical, primero se hizo la prueba con una máquina virtual con 1gb de RAM, 1 core y Xubuntu 15,04 corriendo sobre ella. Posteriormente se realizó la misma prueba asignando a la máquina virtual 2gb de RAM y 2 cores.

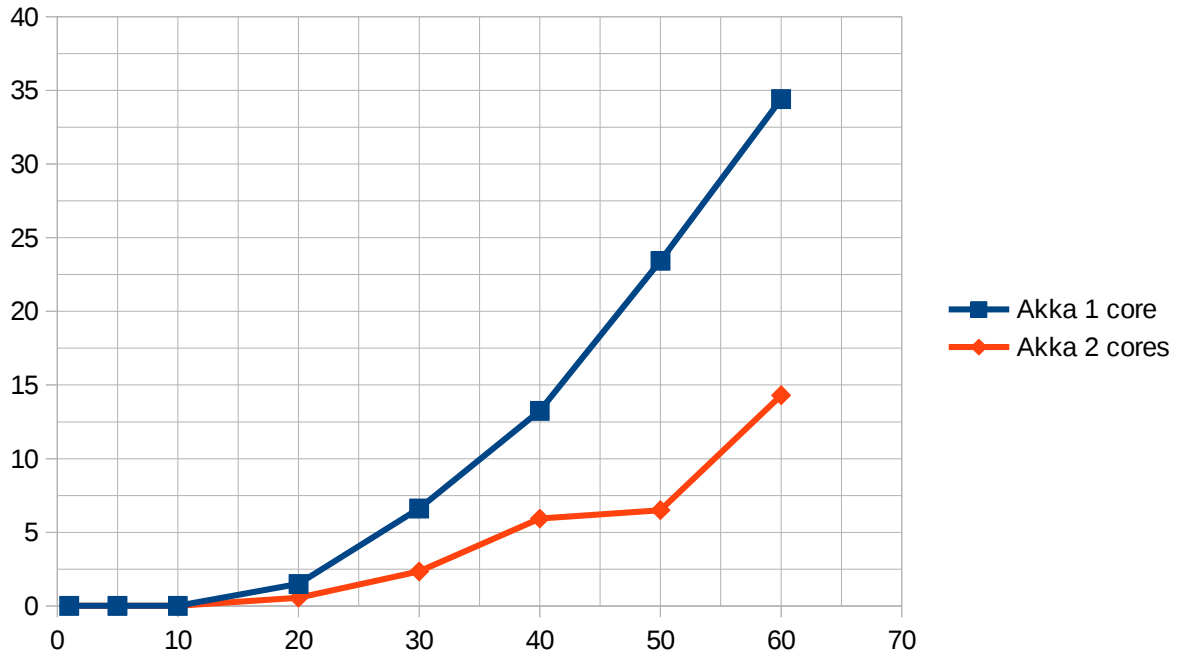


Ilustración 21: Tiempo medio de los mensajes (segundos) en función del numero de clientes en Akka

Estos son los primeros resultados. En ellos vemos cómo aumenta el tiempo medio (en segundos) que tardan en regresar los mensajes a los clientes en función del número de clientes. Como podemos ver, con un número no muy alto de clientes los tiempos no difieren demasiado, pero conforme vamos añadiendo clientes la diferencia aumenta hasta quedarse sobre 30 segundos con 60 clientes.

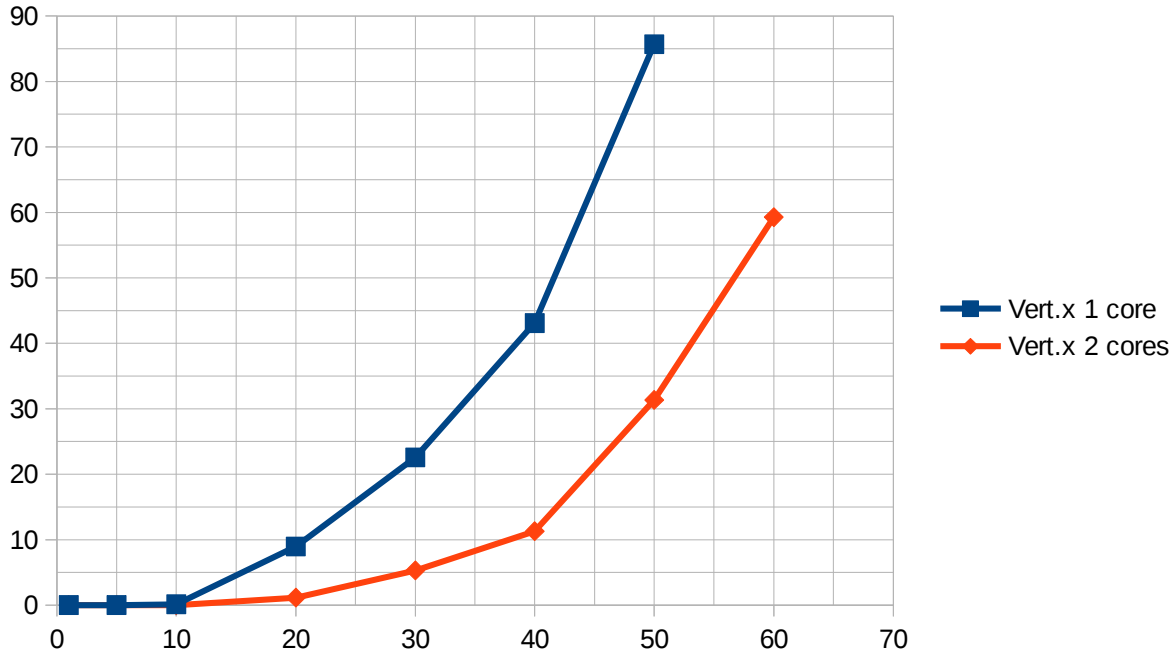


Ilustración 22: Tiempo medio de los mensajes (segundos) en función del numero de clientes en Vert.x

Aquí se muestra los resultados de la misma prueba ejecutada sobre la aplicación realizada con vert.x. Como podemos observar, las diferencias comienzan a partir de los 10 clientes (50000 mensajes en total) y en los 50 clientes la diferencia es cercana a los 30 segundos.

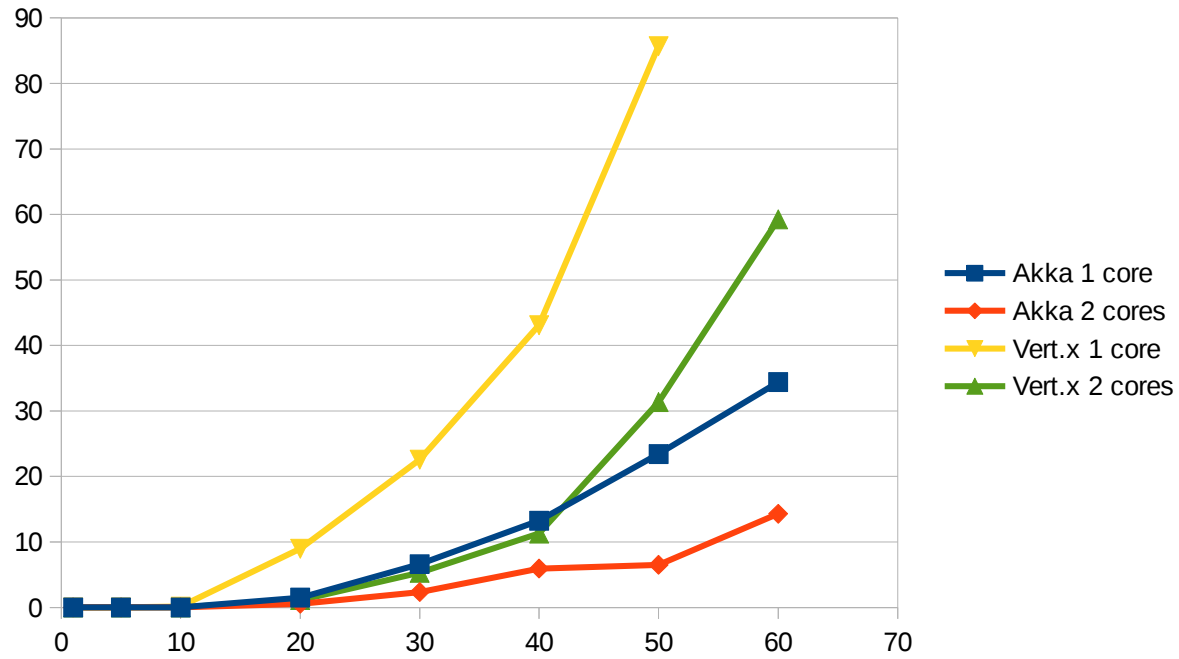


Ilustración 23: Tiempo medio de los mensajes (segundos) en función del numero de clientes

En esta gráfica podemos comparar los tiempos de akka y vert.x. Vemos que el rendimiento de Vert.x con dos cores se asemeja al de Akka y un core, hasta que a partir de 40 clientes se dispara. Durante las pruebas hemos tenido problemas con los consumos de memoria de vert.x, ya que son muy superiores a los de akka en situaciones equivalentes.

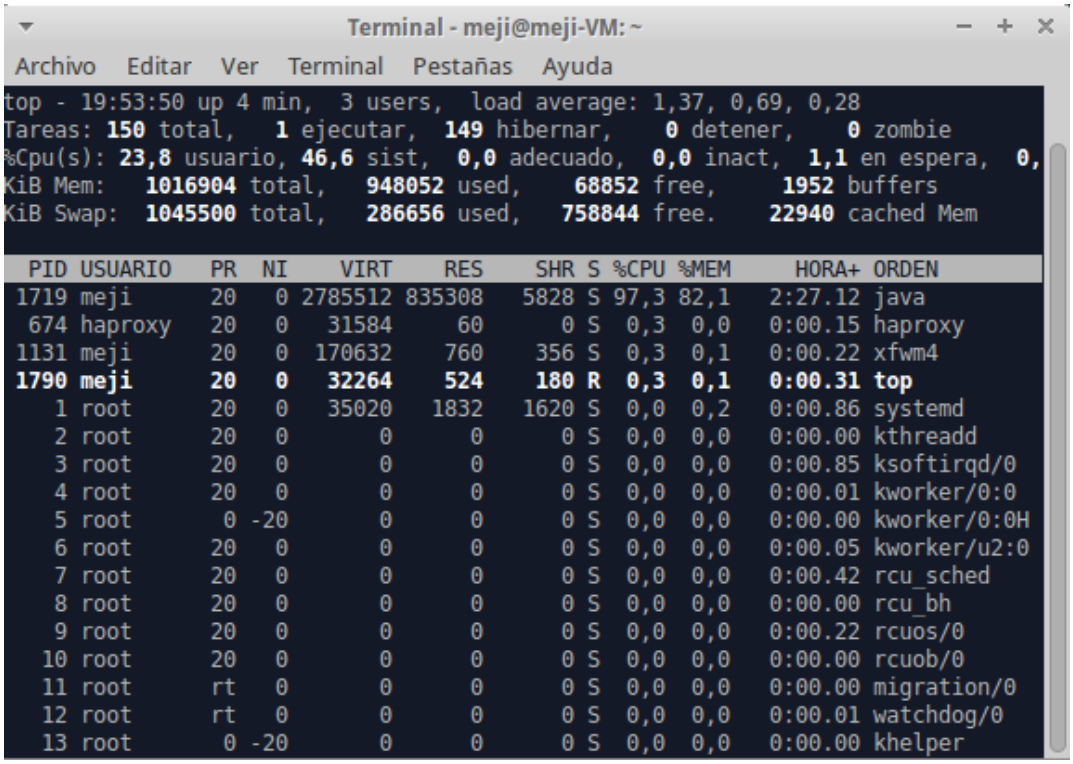


Ilustración 24: Consumo de memoria de Vert.x con 50 clientes

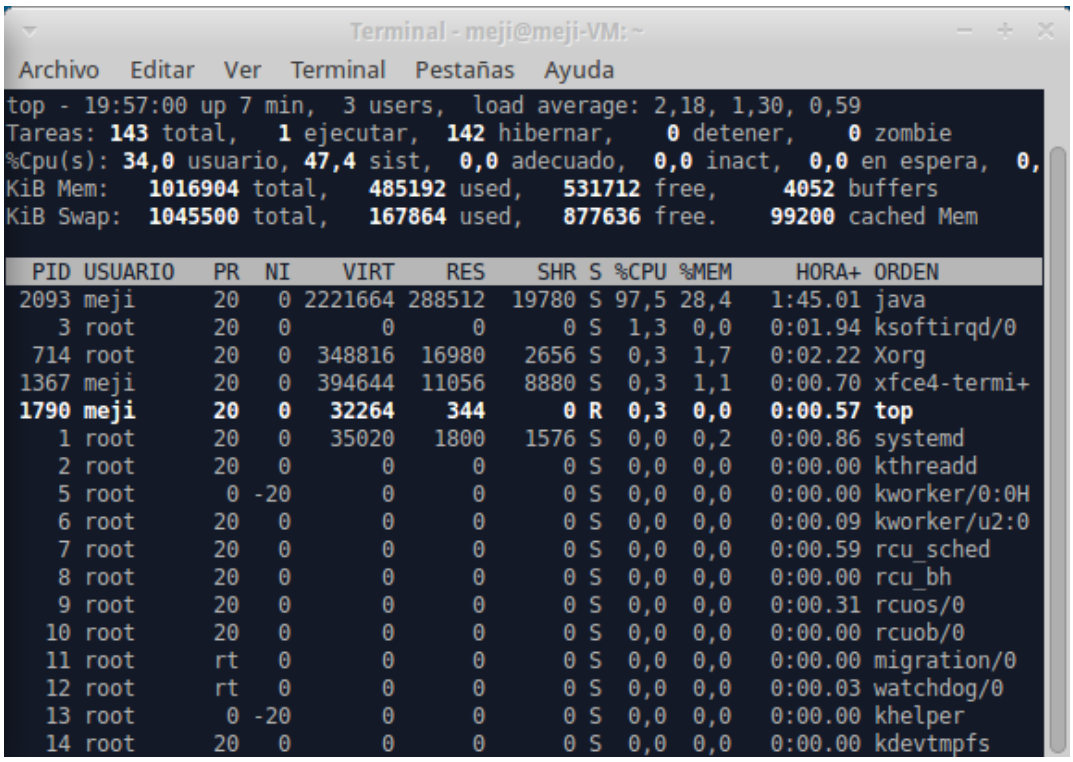


Ilustración 25: Consumo de memoria de Akka con 50 clientes

Como podemos ver en las capturas previas, en la misma situación, el consumo de memoria de vert.x ronda el 80% (el máximo que le asignamos previamente a la máquina

virtual) mientras que en akka se sitúa en el 28%. Debido a estos problemas, nos planteamos modificar el diseño de la versión con vert.x. El principal motivo por el que nos planteamos esto es que quizás los verticles no eran tan ligeros como habíamos supuesto en un principio. Para comprobarlo, realizamos una implementación en la que únicamente teníamos un verticle principal encargado de recibir y enviar los mensajes, pero tras algunas pruebas, comprobamos que el consumo de memoria seguía siendo el mismo. Esta implementación puede encontrarse en: https://github.com/meji92/WebChatVertx/blob/master/src/main/java/com/globex/app/ChatManagerSimple.java

5.2 CHAT DISTRIBUIDO - ESCALADO HORIZONTAL

Para realizar esta prueba hemos levantado en dos máquinas virtuales idénticas la aplicación y en uno de ellas además HAProxy. Las máquinas tienen las mísmas características que en la prueba anterior: 1gb de RAM y 1 core. Después de realizar la prueba con dos máquinas se ha añadido una tercera y se han vuelto a realizar las pruebas para comprobar que tal escalan horizontalmente las versiones con akka y vert.x de la aplicación.

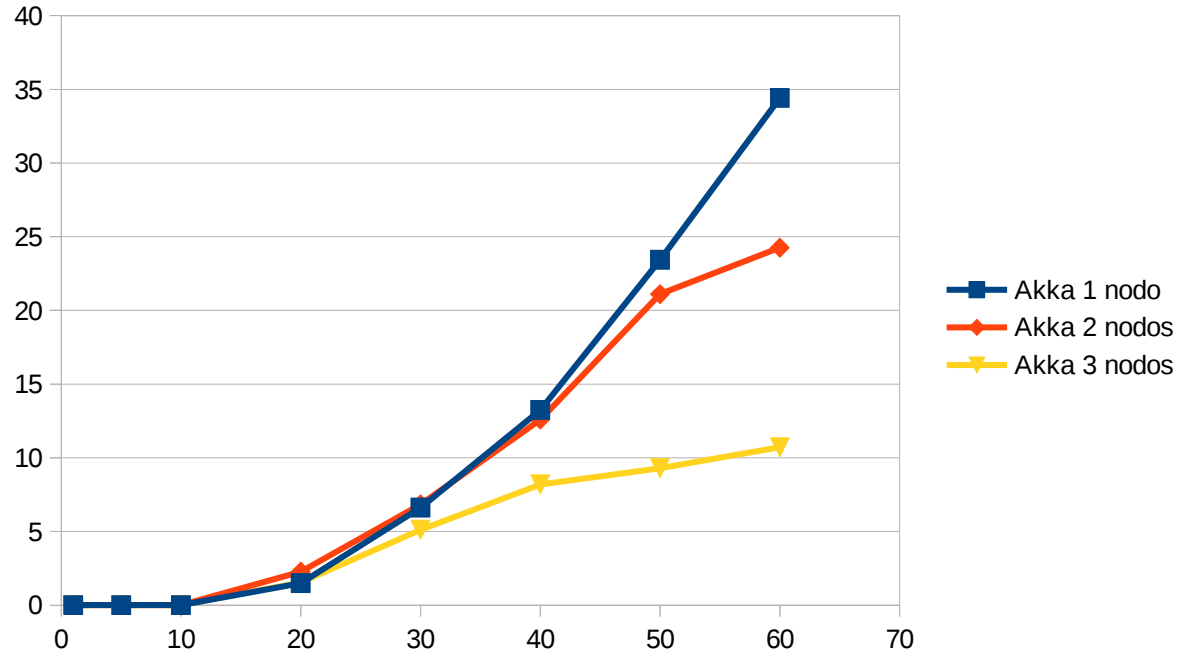


Ilustración 26: Tiempo medio de los mensajes (segundos) en función del numero de clientes con Akka cluster

En esta primera gráfica podemos observar como aumenta el rendimiento de la aplicación considerablemente solo con 60 usuarios, mientras que con 3 nodos los tiempos mejoran a partir de los 30 usuarios.

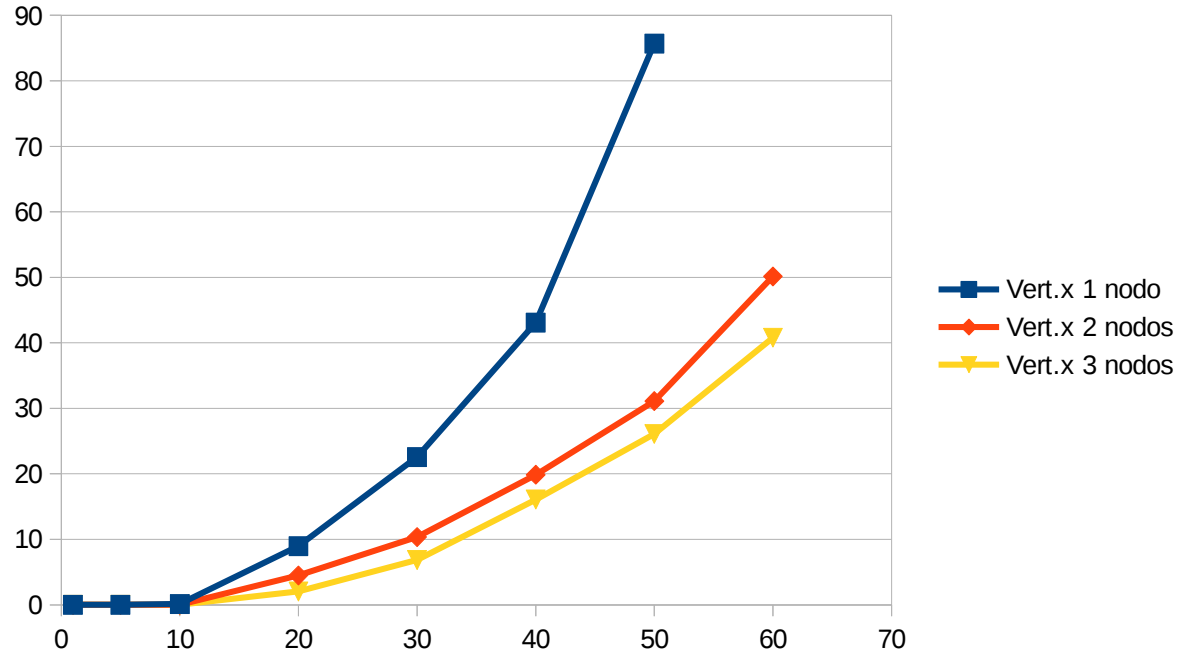


Ilustración 27: Tiempo medio de los mensajes (segundos) en función del numero de clientes con Vert.x en cluster

En vert.x, al contrario que en akka si se produce una mejora importante de 1 a dos nodos mientras que el paso a 3 nodos también mejora aunque no tan notablemente.

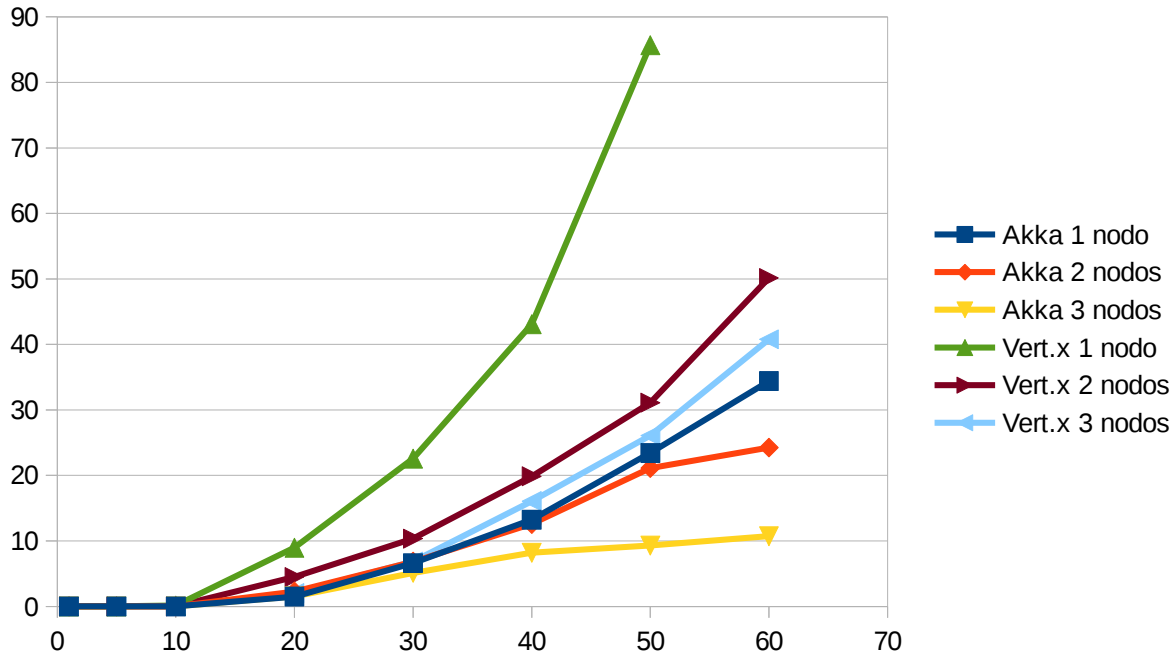


Ilustración 28: Tiempo medio de los mensajes (segundos) en función del numero de clientes en cluster

Aquí tenemos la comparación entre vert.x y akka. Como podemos observar, incluso con 3 nodos, vert.x obtiene peores resultados que akka con 1 solo nodo.

5.3 ESCALADO HORIZONTAL EN CLOUD

Para esta prueba utilizamos la capa gratuita de amazon EC2, que nos permite levantar 10 instancias t2.micro con Ubuntu server 14.04, de un núcleo y 1GB de RAM, que se ejecutan sobre un intel Xeon E5-2670 v2 a 2,5GHz. Realizamos la misma prueba que en los apartados anteriores con 1, 2, 5 y 10 instancias.

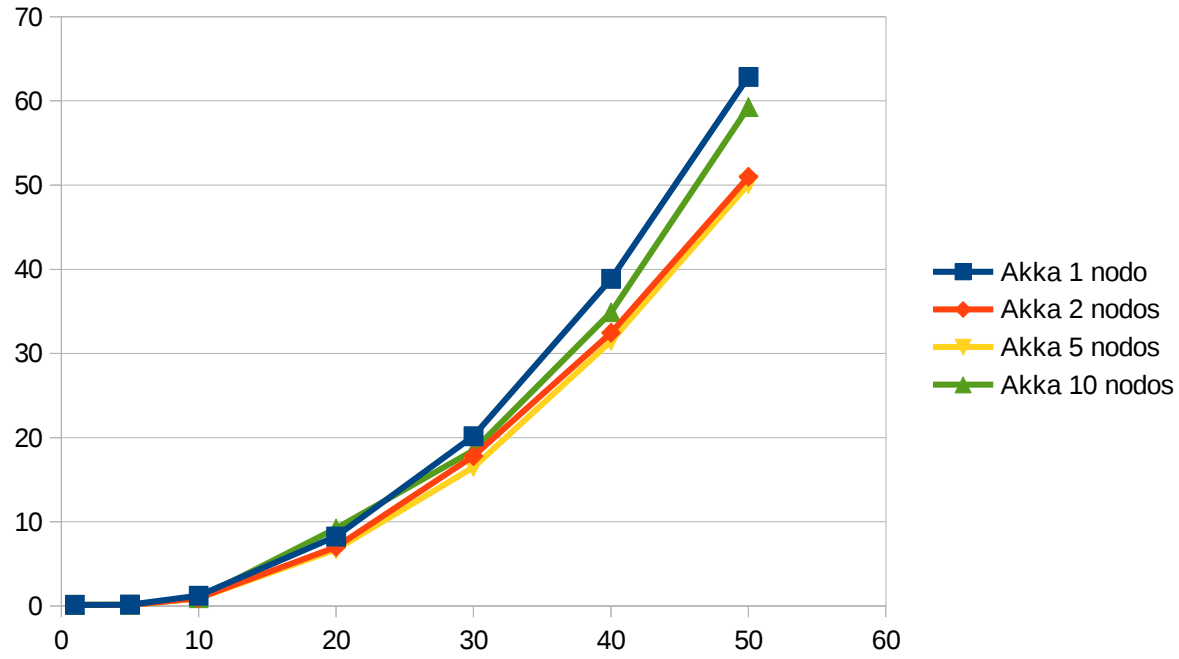


Ilustración 29: Tiempo medio de los mensajes (segundos) en función del numero de clientes con Akka en EC2

En akka vemos que de 1 a 2 nodos si tenemos una notable mejoría, sin embargo, con de 2 a 5 nodos no hay mejoría apreciable y de 5 a 10 nodos tenemos un deterioro del rendimiento. Esto puede ser debido al exceso de comunicaciones y la latencia que estas añaden. Una estrategia distinta a la que hemos usado, como tratar de agrupar los usuarios de un mismo chat en un mismo nodo podría solucionar estos problemas.

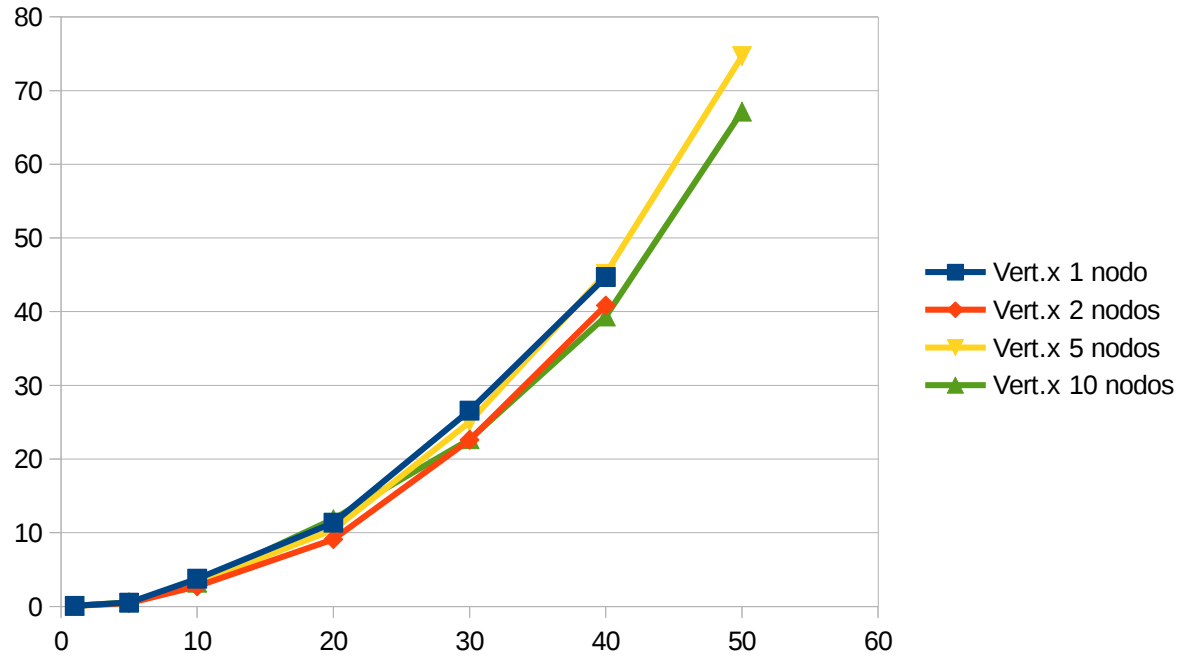


Ilustración 30: Tiempo medio de los mensajes (segundos) en función del numero de clientes con Vert.x en EC2

Aquí podemos ver como en vert.x ocurre exactamente lo mismo que en akka, el rendimiento mejora de 1 a 2 nodos, pero a partir de ahí no mejora e incluso empeora.

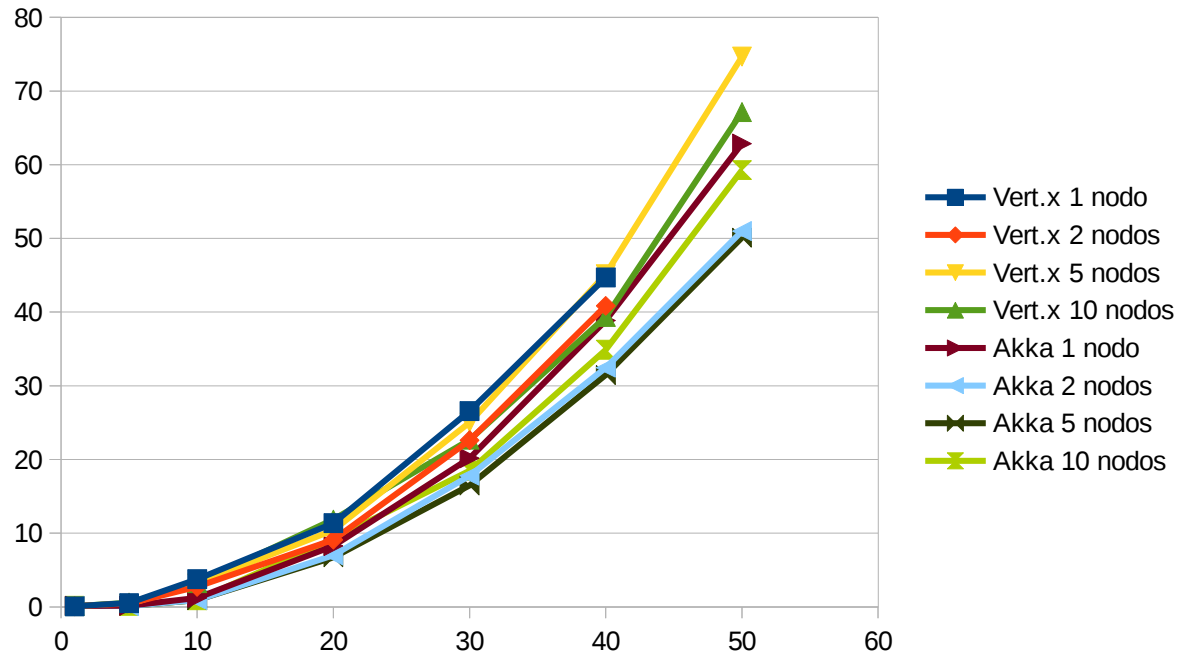


Ilustración 31: Tiempo medio de los mensajes (segundos) en función del numero de clientes en EC2

Aquí podemos ver la comparación entre ambas implementaciones. Podemos observar que la las pruebas en un único nodo con akka obtenemos mejores resultados que todas las pruebas realizadas con vert.x, al igual que en el apartado anterior.

6. CONCLUSIONES

6.1 TRABAJOS FUTUROS

Algunos de las posibles mejoras que se podían realizar son:

• Realizar la implementación con vert.x usando la versión 3 de este, lo que simplificaría enormemente la aplicación y nos permitiría comprobar el rendimiento en esta nueva versión.

• Realizar una implementación mas sencilla con Akka, utilizando exclusivamente el publish-subscribe proporcionado por akka, es decir, eliminando todos los mensajes directos y comprobar si hay cambios en el rendimiento.

• Realizar una implementación de akka en la que se trate de agrupar los usuarios del mismo chat en el mismo nodo para así reducir el número de mensajes por la red.

6.2 CONCLUSIONES PERSONALES

Este proyecto me ha permitido adquirir conocimientos sobre el desarrollo con estos dos toolktis y en general sobre el modelo de actores, la programación reactiva y el desarrollo de aplicaciones distribuidas. En cuanto a los toolkits, los dos son muy interesantes, pero si tuviera que elegir uno me quedaría con akka, ya que además de los resultados de las pruebas, en mi opinión la documentación es mucho mejor, a parte de que tiene una comunidad de desarrolladores mayor, por lo que es mucho mas fácil encontrar información en internet. Además hay cosas en la forma de programar con vert.x que no me han acabado de convencer, como por ejemplo la manera de obtener el deploymentID para destruir los verticles.

7. BIBLIOGRAFIA

[1]. http://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/

[2]. Distributed Systems: Concepts and Design George F. Coulouris, Jean Dollimore, Tim Kindberg

[3]. https://en.wikipedia.org/wiki/Actor_model

[4]. https://rocketeer.be/articles/concurrency-in-erlang-scala/

[5]. http://www.reactivemanifesto.org/

[6]. http://vertx.io/

[7]. https://en.wikipedia.org/wiki/Vert.x

[8]. http://www.cubrid.org/blog/dev-platform/inside-vertx-comparison-with-nodejs/

[9]. http://akka.io/

[10]. http://doc.akka.io/docs/akka/2.3.13/intro/what-is-akka.html

[11]. https://en.wikipedia.org/wiki/Akka_%28toolkit%29

[12]. https://es.wikipedia.org/wiki/Java_%28lenguaje_de_programaci%C3%B3n%29

[13]. https://docs.oracle.com/javase/specs/jvms/se8/html/index.html

[14]. https://es.wikipedia.org/wiki/M%C3%A1quina_virtual_Java

[15]. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[16]. https://www.playframework.com/][https://en.wikipedia.org/wiki/Play_framework

[17]. http://csse.usc.edu//TECHRPTS/1988/usccse88-500/usccse88-500.pdf

[18]. https://es.wikipedia.org/wiki/Desarrollo_en_espiral

[19]. https://tools.ietf.org/html/rfc6455][https://es.wikipedia.org/wiki/WebSocket

[20]. https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git

[21]. https://es.wikipedia.org/wiki/Git

[22]. https://maven.apache.org/

[23]. https://es.wikipedia.org/wiki/Maven

[24]. http://www.scala-sbt.org/

[25]. https://en.wikipedia.org/wiki/SBT_%28software%29]

[26]. https://es.wikipedia.org/wiki/Eclipse_%28software%29

[27]. https://eclipse.org/home/index.php

[28]. https://www.jetbrains.com/idea/

[29]. https://es.wikipedia.org/wiki/IntelliJ_IDEA

[30]. http://doc.akka.io/docs/akka/snapshot/java/cluster-usage.html]

[31]. Akka 2.3.14 documentation 6.1 Cluster Specification

[32]. https://en.wikipedia.org/wiki/Vector_clock]

[33]. https://en.wikipedia.org/wiki/Gossip_protocol

[34]. http://doc.akka.io/docs/akka/snapshot/java/distributed-pub-sub.html]

[35]. https://www.playframework.com/documentation/2.0/ScalaTemplates

[36]. https://www.playframework.com/documentation/2.3.x/JavaAkka#Converting-Akka-Future-to-Play-Promise

[37]. http://doc.akka.io/docs/akka/snapshot/java/cluster-metrics.html#Adaptive_Load_Balancing

[38]. http://www.vmware.com/products/vrealize-hyperic/

[39]. http://doc.akka.io/docs/akka/snapshot/java/cluster-metrics.html#Hyperic_Sigar_Provisioning

[40]. https://en.wikipedia.org/wiki/Reactor_pattern

[41]. http://stackoverflow.com/questions/23023653/vert-x-get-deployment-id-within-currently-running-verticle

[42]. http://vertx.io/docs/vertx-core/java/#_cluster_wide_asynchronous_maps

[43]. http://vertx.io/docs/vertx-core/java/#_server_sharing

[44]. http://vertx.io/docs/vertx-core/java/#_high_availability

[45]. http://www.haproxy.org/

[46]. http://cbonte.github.io/haproxy-dconv/configuration-1.5.html

[47]. http://www.keepalived.org/

[48]. http://dasunhegoda.com/how-to-setup-haproxy-with-keepalived/833/

[49]. http://www.keepalived.org/pdf/LVS-HA-using-VRRPv2.pdf

[50]. https://es.wikipedia.org/wiki/Virtual_Router_Redundancy_Protocol

[51]. https://tools.ietf.org/html/rfc3768

[52]. https://www.typesafe.com/blog/why_do_we_need_a_reactive_manifesto%3F

[53]. http://stackoverflow.com/questions/16945345/differences-between-tcp-sockets-and-web-sockets-one-more-time

[54]. https://github.com/vert-x/testtools