UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Diseño, implantación y evaluación de mecanismos de actualización de servicios distribuidos basados en replicación pasiva.

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Antonio Manuel Larriba Flor

Tutor: Francisco Daniel Muñoz Escoí

Curso 2015-2016

Resum

L'actualització de sistemes distribuïts en temps d'execució és encara un tema espinós en l'àmbit de la informàtica. Actualment no existeix cap proposta completa i general que puga dur a terme l'actualització sense comprometre la disponibilitat del servei. L'objectiu d'aquest treball és estudiar alguna de les propostes existents per a posteriorment millorar-la, amb la fi de dissenyar una solució més completa i general que s'adapte al màxim nombre de situacions possibles. Una vegada dissenyat l'algorisme d'actualització es desenvoluparà una versió inicial en la que es provaran diversos casos senzills i s'avaluarà el seu rendiment.

Paraules clau: Sistemes distribuïts, Replicació passiva, Actualització de servicis, Nodejs

Resumen

La actualización de sistemas distribuidos en tiempo de ejecución sigue siendo un tema complicado en el ámbito de la informática. Actualmente no existe ninguna propuesta completa y general que pueda llevar a cabo la actualización sin comprometer la disponibilidad del servicio. El objetivo de este trabajo es estudiar alguna de las propuestas existentes para posteriormente mejorarla, con el fin de diseñar una solución más completa y general que se adapte al mayor número de situaciones posibles. Una vez diseñado el algoritmo de actualización se desarrollará una versión inicial sobre la que se probarán varios casos sencillos y se evaluará su rendimiento.

Palabras clave: Sistemas distribuidos, Replicación pasiva, Actualización de servicios, Nodejs

Abstract

Updating distributed systems at running time is quite a difficult problem in the computer science area yet. Nowadays it does not exist a general and complete proposal able to carry on the update without compromising the availability of the service. The target of this work is to study some of those existing proposals in order to improve them, with the aim of designing a more complete and general one able to adapt to a wider range of scenarios. Once designed the updating algorithm a demo will be developed to test some simple cases and compute its efficiency.

Key words: Distributed Systems, Passive model replication, Update services, Nodejs

Índice general

Índice de figuras

Índice de algoritmos

CAPÍTULO 1
Introducción

1.1 Motivación

El aumento del uso de los servicios distribuidos en Internet tales como el correo electrónico, las redes sociales o las compras online ha resultado en que la disponibilidad de los servicios en línea sea un aspecto crítico. Se han hecho numerosos avances en los estudios con respecto a la tolerancia a fallos y la robustez de los sistemas distribuidos. No obstante la gran mayoría de estos servicios no resuelven todas las situaciones de indisponibilidad a la hora de actualizar versiones. Se busca garantizar la transición entre versiones sin que se pierda funcionalidad en el servicio.

1.2 Objetivos

El objetivo de este trabajo es, como se ha explicado en líneas anteriores el de garantizar la disponibilidad de servicios distribuidos (basados en la replicación pasiva) durante la actualización del sistema. Después de aprender y evaluar algunas de las soluciones existentes se busca proponer una solución más completa que se implementara en una versión de prueba sobre la que poder evaluar los costes de la actualización *on the fly*. No obstante los sistemas distribuidos no son sistemas sencillos por lo que se aclara desde un principio que el objetivo de este trabajo no es el desarrollo de un sistema distribuido completamente realista, se hacen ciertas asunciones y se parte de algunas hipótesis para evitar que el alcance del trabajo sea demasiado general y/o ambicioso.

1. Se asume que existe un servicio de nombres distribuido y replicado capaz de proveer la/s dirección/es del servidor primario a los procesos que lo soliciten. Así como también de actualizar las direcciones de este en el caso de que sean modificadas.

2. Se asume que el problema de la pertenencia estará resuelto de una manera descentralizada y resistente a fallos. El problema de la *pertenencia a grupo* consiste en manterner el conjunto de procesos que pertenecen a un determinado "grupo"[6]. Suele plantearse en la gestión de servicios replicados, para controlar en todo momento qué procesos forman parte de ese grupo. Su

principal dificultad reside en detectar rápidamente las caídas (o cualquier otro tipo de fallo) de los miembros y llegar a un acuerdo entre el resto de los miembros que permanezcan vivos. Eso requiere consenso y es imposible de lograr en un entorno asíncrono en el que los procesos puedan fallar [8] [4] . Para resolverlo se propuso como mecanismo el "detector de fallos no fiable"[5].

3. Se trabaja con la hipótesis de que las diferentes versiones del servicio son compatibles a la hora de procesar mensajes. Esto significa que si durante la operación de actualización existen servidores con distintas versiones ambos deben ser capaces de tratar las peticiones del cliente o las actualizaciones del servidor primario para evitar que haya problemas de inconsistencia entre ellos.

4. Se ignora el control de credenciales y la autentificación para controlar la seguridad del sistema. Es un aspecto a tener en cuenta, pero al igual que en el caso del problema de pertenencia son problemas complejos que merecerían de su propio trabajo de investigación para ser tratados correctamente.

5. Se presupone que la pérdida de mensajes es despreciable gracias a la fiabilidad proporcionada por la capa de intercambio de mensajes.

6. La versión que se desarrollará estará limitada a gestionar un solo servicio distribuido. No se podrán gestionar dos o más servicios simultáneamente.

Una vez definidas las restricciones y los supuestos con los que se trabaja se define cuáles son los objetivos concretos que se busca alcanzar:

1. Adaptar una solución teórica para la actualización de sistemas distibuidos que no comprometa su disponibilidad. Para ello debe ser capaz de soportar fallos en las réplicas durante la actualización.

2. Implementarla en un sistema distribuido que será desarrollado usando tecnologías vistas en la carrera en base a las suposiciones y restricciones explicadas previamente.

3. Conseguir que el sistema propuesto sea capaz de soportar diferencias entre versiones que no afecten a las interfaces ni al estado pero sí a la funcionalidad de los servidores.

4. Conseguir que el sistema propuesto sea capaz de soportar diferencias entre versiones que no afecten a las interfaces pero sí que provocan un cambio de estado y/o de funcionalidad.

5. Conseguir que el sistema propuesto sea capaz de soportar diferencias entre versiones que afecten a las interfaces y/o provoquen un cambio de estado y/o de funcionalidad.

6. Alcanzar el máximo posible de eficiencia en el proceso de actualización, buscando el mínimo intercambio de mensajes que se traduzca en un menor tiempo.

1.3 Estructura de la memoria

En este primer capítulo se introduce de manera general las razones por las que el estudio de la actualización de sistemas distribuidos es un tema que puede ser tanto interesante como útil. Se definen también las restricciones y los objetivos que se buscan alcanzar. El resto de la memoria se organiza de la siguiente manera: en el capítulo dos se da un contexto general sobre sistemas distribuidos y un breve repaso al estado del arte sobre sistemas de actualización. En el capítulo tres se desarrolla y explica la solución propuesta mientras que en el capítulo cuatro se habla de la implementación y de las tecnologías usadas. Finalmente en el capítulo 5 se presentan las conclusiones y futuras líneas de investigación.

CAPÍTULO 2
Contexto general

En esta sección se da una explicación de qué son los sistemas distribuidos y los diferentes modelos de replicación. Se hace con la intención de que cualquier lector pueda entender la terminología usada en el trabajo. Si el lector ya tiene conocimientos en estos temas puede saltar el capítulo. Aunque se recomienda que lea el último apartado de esta sección para estar al tanto del estado del arte sobre la actualización de sistemas distribuidos.

2.1 ¿Qué es un sistema distribuido?

Los sistemas distribuidos son un caso concreto de los sistemas concurrentes. Los sistemas concurrentes son aquellos en los que se desarrollan actividades de manera paralela. A pesar de los inconvenientes que esto supone como la gestión de recursos compartidos y la prevención de condiciones de carrera entre otros, estos sistemas proporcionan un mejor aprovechamiento de los recursos y la consecuente mejora del rendimiento.

Lo que caracteriza a los sistemas distribuidos con respecto a otros sistemas concurrentes es que en los primeros cada uno de los elementos se ejecuta de forma autónoma y por norma general cada uno de ellos tiene su propio conjunto de recursos, principalmente memoria. El hecho de que cada proceso tenga memoria propia hace necesario que cooperen usando alguna plataforma que permita el intercambio de información con el fin de que la información en todo el sistema se mantenga consistente.

Se podría resumir que un sistema distribuido es un sistema concurrente en el que procesos independientes cooperan entre sí, intercambiando mensajes, con el fin de proporcionar un servicio como si fuera un único sistema. Además de la mayor disponibilidad y el impacto positivo en el rendimiento, los sistemas distribuidos tienen otras propiedades:

Ocultación del número de servidores, su localización, las diferencias entre las máquinas en las que se ejecutan los procesos, las comunicaciones entre ellas etc. El sistemas debe de ser **transparente** [13] al usuario. De hecho el cliente no tiene por qué saber que el sistema está distribuido en varios servidores.

El objetivo es crear a los clientes la ilusión de que el servicio ha sido implementado en un único servidor.

Disponibilidad de los servicios ofrecidos. El sistema debe estar siempre disponible, por lo que debe de estar preparado para reaccionar a los posibles fallos que puedan ocurrir durante su ejecución.

Escalabilidad del sistema. Debe de ser relativamente sencillo poder añadir nuevos procesos, reubicarlos y administrarlos.

Acceso homogéneo al sistema con independencia del punto de acceso.

Para ejemplificar las características expuestas se presenta como ejemplo de lo que podría ser un sistema distribuido la imagen 2.1:

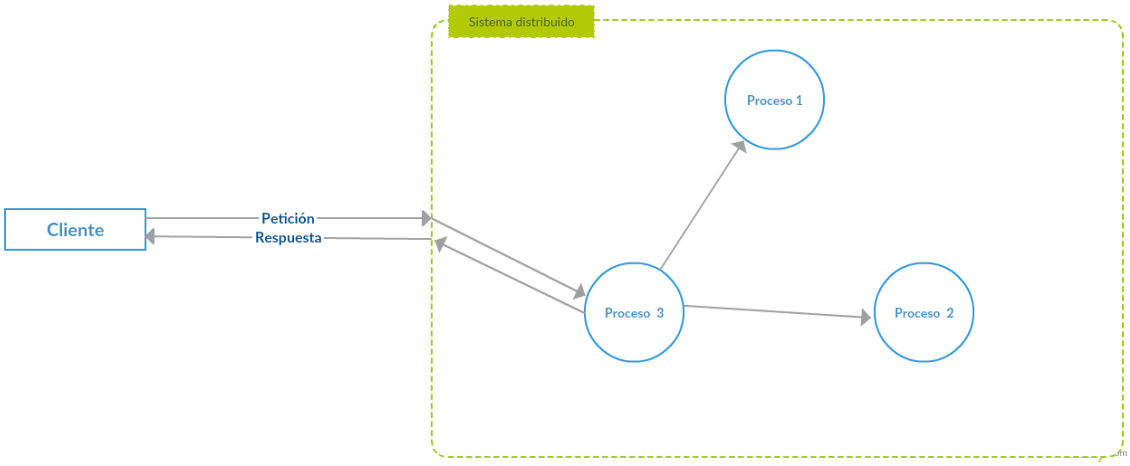

Figura 2.1: Ejemplo de sistema distribuido.

Como se puede observar, el cliente realiza una petición al sistema distribuido sin conocer los detalles de como funciona. Para él es una caja negra a la que realiza peticiones y obtiene respuestas. En el caso de que el proceso encargado de responder a las peticiones del cliente sufriera algún fallo, cualquiera de los otros dos podría reemplazarle para seguir ofreciendo servicio y que la disponibilidad no se viera afectada. Sería posible mediante ciertos protocolos añadir nuevos procesos que ayudaran a escalar el servicio. Estos nuevos procesos podrían estar en localizaciones distintas para asegurar un tiempo de acceso más uniforme sin depender desde donde realiza las peticiones el cliente.

2.2 Modelos de replicación activa y pasiva

Cada uno de los procesos que forma parte del sistema distribuido tiene sus propios recursos y no tiene conocimiento de la información que tienen el resto de procesos del sistema. Por este motivo se hace necesario que se comuniquen de alguna manera. Lo más común es que la información se transmita gracias al intercambio de mensajes entre ellos. Dependiendo de como se comuniquen estos procesos podemos clasificarlos en un modelo u otro, existen diversos modelos

peer-to-peer, cloud, grid..., pero en este trabajo nos vamos a centrar en la clásica interacción cliente/servidor.

Hay dos modelos de replicación que predominan en la interacción cliente/servidor, la replicación activa [11] y la pasiva [3]. Vamos a presentarlos con mayor detalle en lo siguientes apartados.

2.2.1. Replicación Activa

Este modelo de replicación recibe este nombre porque todos los procesos que forman parte de él juegan un papel proactivo. Cualquiera de las réplicas existentes puede hacer lo mismo que las otras. Todas se encuentran al mismo nivel, todas son idénticas y ejecutan las mismas acciones.

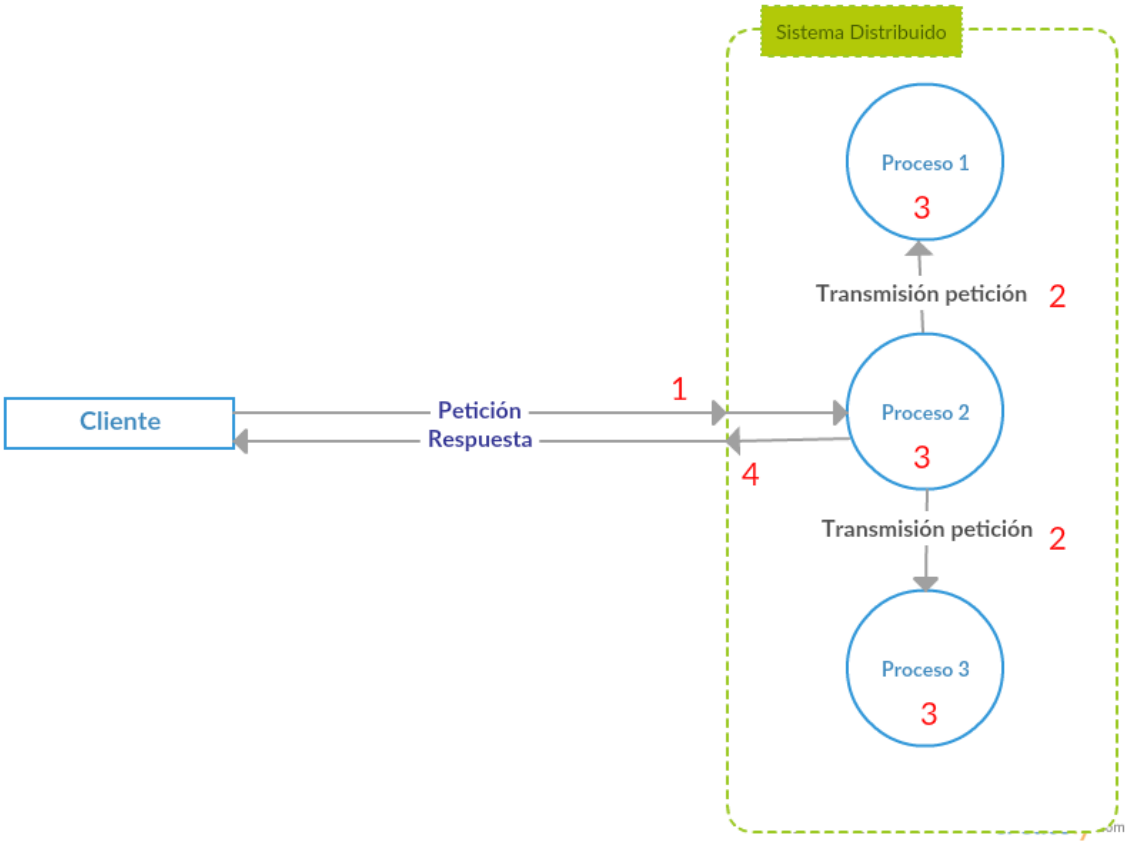

Figura 2.2: Ejemplo de replicación activa.

En la imagen 2.2 podemos observar un ejemplo de sistema distribuido basado en el modelo de replicación activa. El funcionamiento del sistema sería el siguiente:

1. El cliente manda la petición al sistema sin conocer cuantas réplicas forman parte del él. Uno de los procesos recibe la petición, podría ser cualquiera ya que todos tienen un papel activo.

2. Antes de procesar la solicitud esta es transmitida al resto de réplicas del sistema.

3. Cada réplica procesa la solicitud.

4. Al ser todas las réplicas idénticas, todas responden al cliente. Gracias a este funcionamiento se pueden gestionar modelos de fallos más severos, como el arbitrario [9] (un proceso falla de manera aleatoria). El cliente es el encargado de filtrar estas respuestas, este proceso supondrá un retardo más o menos grande en función de como de desfavorable sea el modelo de fallos que se busca soportar.

Hay partes del funcionamiento del modelo activo que se han obviado en el ejemplo para simplificar el diagrama y facilitar la comprensión de la idea general. Por ejemplo, las réplicas deben acordar previamente el orden en el que van a procesarlas para que el resultado sea el mismo para todas y la información se mantenga consistente. Para ello se necesita implantar un algoritmo de difusión fiable de mensajes en orden total [7]. Ese algoritmo requiere consenso e introduce un retardo considerable en la entrega de los mensajes de petición. Por este motivo suele haber una réplica que actúa como secuenciador, llega a un acuerdo con el resto de réplicas mediante el intercambio de mensajes y todas siguen ese orden. Se consigue de esta manera una consistencia secuencial [10]. Este modelo, como todos, presenta ventajas y desventajas. Las virtudes son las siguientes:

Mensajes pequeños , el intercambio de información no consume mucho ancho de banda ya que sólo se transmiten las peticiones. Ninguna información extra es necesaria ya que todas las réplicas van a procesar el mismo mensaje.

Reconfiguración instantánea en el caso de que alguna réplica falle, mientras queden réplicas vivas sus roles no se ven afectados.

Como aspectos negativos se destacas los siguientes puntos.

Se repiten cálculos en cada proceso. Si todas las réplicas procesan la misma solicitud significa que todas están haciendo el mismo trabajo cada vez que llega una petición.

El coste de establecer un orden para procesar los mensajes es caro. Para alcanzar un consenso en el orden de procesamiento se requiere intercambiar bastantes mensajes.

Con el modelo activo se consigue una consistencia bastante fuerte, el hecho de repetir cálculos en cada proceso puede ser más o menos importante dependiendo del servicio que se busque implementar. Por ello se recomienda usar este modelo de replicación cuando el cómputo necesario para procesar las solicitudes no sea muy costoso.

2.2.2. Replicación pasiva

Este modelo de replicación se denomina pasivo porque de entre sus réplicas se escoge una que actúa como primario y el resto adoptan un papel pasivo y se limitan a recibir mensajes del primario. El modelo pasivo es anterior al modelo activo

y fue descrito por primera vez en 1976 [1] por Alsberg con el objetivo de crear un mecanismo de compartición de recursos distribuidos resistente a fallos. Se optó por esta aproximación ya que proporcionaba menores tiempo de respuesta frente a tener varios primarios ya que no se necesitaba invertir tiempo para decidir el orden en el que las réplicas procesan las solicitudes (modelo activo). Se presenta la imagen 2.3 para explicar mejor como funciona el modelo pasivo.

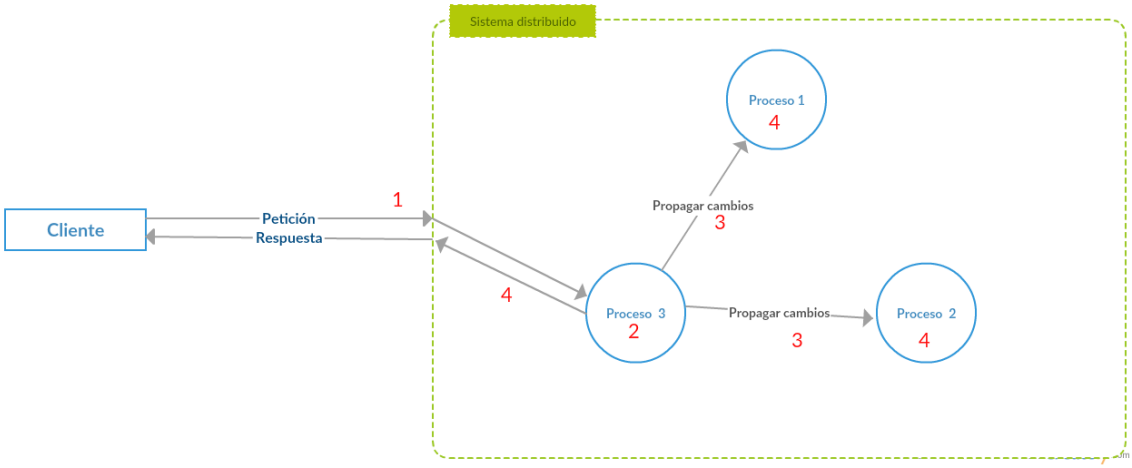

Figura 2.3: Ejemplo de replicación pasiva.

1. El cliente manda la petición al sistema sin conocer cuantas réplicas forman parte del él. El primario recibe la petición, en este caso el primario es el proceso 3.

2. El primario procesa la solicitud y calcula los cambios necesarios para modificar su estado.

3. El primario propaga esos cambios al resto de réplicas que actúan como réplicas pasivas para que puedan actualizar su estado y la información del sistema se mantenga consistente.

4. Mientras que las réplicas pasivas aplican a su estado los cambios propagados por el primario este responde al cliente sin esperar respuesta de las réplicas pasivas.[1]

En el caso de que el servidor primario sufriera algún tipo de fallo una de las réplicas pasivas pasará a ocupar su lugar para que el servicio siga funcionando. Para definir mejor lo que es el modelo de replicación pasiva se citan a continuación cuatro propiedades descritas por Budhiraja. Son cuatro propiedades que cualquier sistema distribuido basado en la replicación pasiva debería cumplir.

[1]Dependiendo de la consistencia que se busque alcanzar con el modelo el primario puede esperar o no a recibir una respuesta por parte de las réplicas cuando estas hayan procesado la actualización. Si se espera conseguir una consistencia atómica (la más fuerte) habrá que esperar a que las réplicas secundarias respondan. Si por el contrario no nos importa obtener una consistencia más débil podremos responder al cliente mientras las réplicas trabajan. De este modo respondemos más rápido al cliente.

1. En cualquier momento durante la ejecución del sistema solamente puede haber un primario.

2. Cada cliente guarda la identidad de un servidor *i* de tal manera que para realizar una petición el cliente manda un mensaje al servidor *i*.

3. Si la petición de un cliente por algún motivo llega a un servidor que no es el primario, esa petición debe ignorarse. Sólo el primario puede atender las solicitudes de los clientes.

4. Los fallos en el sistema deberán tener una frecuencia no muy alta y un tiempo finito. Se busca intentar que los fallos que puedan ocurrir en el sistema al usuario le parezca un único fallo de tiempo F. Siendo ese tiempo determinado en función de la resistencia a fallos que se quiera implementar en el sistema.

La resistencia a los fallos depende del modelo de tolerancia que se quiera implementar. Se puede hacer que el sistema sea resistente a fallos en un servidor, fallos en un servidor y en la capa de transporte, pérdida a la hora de recibir mensajes o a la hora de enviarlos por un fallo en un servidor. Obviamente que el sistema tolere estos escenarios tiene un coste tanto en el número mínimo de réplicas que debe de haber en el sistema , como en los tiempos de respuesta y los tiempos de recuperación del servicio (tiempo necesario que necesita el sistema para recuperarse de un error y responder).

Ahora que ya se han presentado las características del modelo pasivo vamos a estudiar las ventajas e inconvenientes como hicimos con el modelo activo. Sus principales puntos fuertes son:

El ahorro de cálculo ya que sólo se calcula el cambio de estado en el primario y el resto de réplicas sólo tienen que aplicar las actualizaciones pertinentes sin tener que volver a repetir cálculos.

Menor tiempo de respuesta si no se busca una consistencia fuerte. Las réplicas no tienen que decidir el orden en el que procesan las peticiones, de eso se encarga el primario.

Soporta operaciones no deterministas porque sólo el primario ejecutará la acción y propagará los efectos de esta. A pesar de usar operaciones no deterministas no se generarán inconsistencias.

Como punto más débiles se destacan los siguientes puntos:

Los mensajes pueden ser enormes dependiendo del servicio. Cuando el primario calcula el cambio de estado que provoca la petición del cliente tiene que pasar estas modificaciones a las réplicas y en algunos casos estos mensajes pueden alcanzar un tamaño considerable.

Reconfiguración costosa si falla el primario ya que hay que cambiar el rol de una réplica para que esta pase a ser el nuevo primario. También hay que informar al resto de procesos de quien es el nuevo primario.

Los modelos de fallos son más costosos de implementar con respecto al número de réplicas necesarias y algunos modelos como el arbitrario no pueden ser implementados porque no todos los fallos en el primario pueden detectarse.

2.3 Estado del arte

El trabajo más amplio que recoge el estado del arte sobre la actualización de sistemas en tiempo de ejecución es probablemente el de Seifzadeh [12]. Se trata de un *survey* que recopila los últimos sistemas de actualización dinámicos y los analiza bajo diferentes criterios que estos sistemas deberían tener en cuenta. Se habla en general de aplicaciones de escritorio, pero también sobre sistemas distribuidos. Además muchos de los aspectos importantes para los primeros también lo son para los segundos. Es un trabajo muy completo que contempla diferentes criterios, compara varios sistemas y propone estrategias para lidiar con alguno de los problemas más recurrentes, por eso se han tratado de resumir los aspectos críticos en la lista que aparece a continuación. Si el lector quiere profundizar en el estudio de la actualización dinámica se le remite a la bibliografía donde encontrará la referencia al trabajo original. Los puntos importantes para los sistemas distribuidos mencionados en el trabajo son:

- Recomienda que el estado de cada nodo se encuentre encapsulado en el mismo y que no sea accesible para el resto de nodos. Se quiere evitar que haya inconsistencia entre los datos de diferentes versiones.

- Diferencia tres casos en función de la atomicidad del proceso de actualización. Por atómico se define un paso o conjunto de pasos que para su correcta ejecución deben bloquear el sistema para que ninguna otra instrucción pueda interferir.

    1. La actualización se realiza en un único paso atómico en el que el servicio se ve bloqueado. Esto no nos interesa para nuestro objetivo ya que compromete la disponibilidad.
    2. En el segundo caso la actualización se realiza de manera paralela con la ejecución normal del servicio. Esto sería el comportamiento ideal ya que la funcionalidad del servicio no se vería comprometida. Esto no siempre es posible y se tiene que buscar un compromiso entre ambas opciones como se describe en el tercer caso.
    3. Si no es posible actualizar todo el sistema sin bloquearlo se aíslan los pasos que necesitan realizarse de manera atómica y se ejecutan por separado a lo largo del proceso. Para que esto funcione en un sistema distribuido necesitamos trabajar con una capa de transporte que nos asegure que no se pierden mensajes y que en el caso de que llegaran peticiones durante los pasos atómicos estas se guardaran en un *buffer* para poder procesarlas después.

- Respecto a los cambios que afecten al tipo de las variables y a las interfaces de las funciones se proponen cuatro estrategias:

1. No hacer nada en especial y suponer que los programadores encargados de hacer compatibles ambas versiones han actualizado todas las funciones y sus dependencias de forma correcta.
2. Proporcionar funciones auxiliares que se encarguen de realizar las transformaciones de tipos necesarias entre versiones.
3. Si es posible, reorganizar las actualizaciones de manera que por su orden de aplicación no afecten al tipo de las variables.
4. Restringir las actualizaciones para que cumplan ciertas propiedades que eviten problemas con los tipos de las variables

- Los sistemas que soporten la actualización en tiempo de ejecución deben ser capaces de detectar los cambios entre versiones antes que el proceso empiece. A este proceso Seifzadeh lo denomina "Diferenciación de código". En el ámbito de los sistemas distribuidos esto es absolutamente necesario para saber qué réplicas trabajan con la versión nueva y cuáles con la versión vieja, cuántas quedan por actualizar, cuáles son compatibles con los últimos cambios de estado etc.

Por otro lado, en un ámbito mucho más concreto Marcin Solarski propone un algoritmo para la actualización de sistemas distribuidos que implementan el modelo pasivo. Los mensajes se intercambian usando un sistema de comunicación a grupos (*Group Communication System*) [6]. Los sistemas de comunicación a grupos son un *middleware* [2] que ante cualquier cambio garantiza informar a todos los miembros del grupo en el mismo estado. Es decir, no informará a un integrante del grupo antes de actualizar y a otro después, asegura que todos los miembros tendrán la misma vista al recibir el mensaje. Partiendo de su uso para el intercambio de mensajes se busca satisfacer los siguientes requisitos:

1. El algoritmo debe iniciarse con un mensaje de actualización que se manda usando el sistema de comunicación habitual.
2. El algoritmo debe de ser capaz de recuperarse automáticamente de los fallos que puedan ocurrir durante su ejecución.
3. El algoritmo debe preservar la consistencia del sistema permitiendo transferencias de estado de las viejas réplicas a las nuevas en el caso de que sean servidores que guardan información sobre la sesión.

Para ello Solarski propone el siguiente algoritmo 2.1.

Los requisitos propuestos por Solarski se alcanzan ya que el algoritmo se dispara cuando llega la petición y el sistema de comunicación a grupos asegura que esa petición llegará a todas las réplicas. De manera similar el *middleware* de intercambio de mensajes es el encargado de preservar la consistencia al permitir las transferencias de estado entre réplicas. Por último al ser el algoritmo descentralizado, ninguna réplica controla el proceso de actualización, si alguna réplica falla el algoritmo será capaz de seguir adelante. Solarski es demasiado escueto y no acaba de explicar cómo se resuelven los fallos durante la ejecución del algoritmo.

Algoritmo 2.1: Algoritmo propuesto por Solarski

1: Lanzar una réplica con el nuevo código como réplica secundaria B_{n+1}
2: **for all** Réplicas secundarias, llevar la actualización de dicha réplica B_i. Donde $0 \leq i \leq n$ **do**
3: Lanzar la réplica $B_i^{'}$ con el código actualizado y hacer que se una al sistema como réplica pasiva.
4: Eliminar la réplica vieja B_i
5: **end for**
6: Forzar el fallo del primario para provocar la elección de uno nuevo entre las nuevas réplicas. De esta manera se elimina el último proceso con la versión antigua del código.

Delega la gestión de fallos a la capa de comunicación que se supone está preparada para estos casos. También menciona que el algoritmo no preserva el nivel de replicación y que si se buscara conseguir ese objetivo nuevos mecanismos deberían encargarse de ello.

CAPÍTULO 3
Solución propuesta: Algoritmo de actualización

Para alcanzar el objetivo de diseñar un algoritmo que no comprometa la disponibilidad del sistema vamos a partir del algoritmo de Solarski y a tratar de extenderlo para que sea capaz de mantener el nivel de replicación. También buscamos disipar las dudas de cómo podría llevarse a cabo la gestión de errores en las réplicas.

3.1 ¿Qué se busca mejorar y cómo?

A continuación se presentan los posibles casos que podían ocurrir durante la ejecución del algoritmo y como deberían de ser tratados:

- **¿Qué hacer en el caso de que falle una réplica secundaria con el código antiguo en mitad del proceso de actualización?** Este es el caso más simple, no hay que hacer nada. Siempre que quede al menos una réplica con el código antiguo que pueda funcionar como primario y encargarse de las transmisiones de estado a las nuevas réplicas no habrá problema. Tal vez a nivel de implementación haya que llevar la cuenta para saber cuántas réplicas antiguas quedan activas.
- **¿Qué hacer en el caso de que falle una réplica secundaria con el código nuevo en mitad del proceso de actualización?** En este caso hace falta llevar el control de cuántas réplicas nuevas han fallado y relanzarlas antes de que el algoritmo finalice. De este modo nos aseguramos de que el número de nodos en el servicio sea el mismo antes y después del proceso de actualización.
- **¿Qué hacer si falla el primario en mitad del proceso de actualización?** Este es el caso más complicado y hay que diferenciar dos casos:

    1. El primario falla al principio o durante la actualización, aún existen réplicas secundarias con la versión antigua del código. En este caso el primario ha fallado y debe escogerse un nuevo **de entre las réplicas con el código antiguo**. Esto es porque las nuevas versiones son las encargadas de proporcionar la compatibilidad con las versiones antiguas

y no viceversa. Si escogiéramos un primario con la versión nueva, nada nos asegura que cuando propagara las actualizaciones las réplicas antiguas pudieran entender esos mensajes. Podrían hacer referencia a nuevos datos o nuevas estructuras que aún no tiene a su disposición.

2. El primario falla cuando el algoritmo de actualización está finalizando, ya solo existen réplicas secundarias con la versión actualizada. Obviamente se elegiría un nuevo primario de entre las réplicas nuevas (las únicas existentes en ese momento) pero habría que señalar de alguna manera que el fallo del primario ya ha ocurrido. En caso contrario si provocáramos el fallo del primario al acabar de sustituir las réplicas antiguas (consultar líneas finales del algoritmo de Solarski) estaríamos eliminando una réplica nueva y volviendo a elegir el primario. El resultado sería correcto siempre y cuando hubiera réplicas suficientes pero acabaríamos con un nodo menos del que deberíamos.

3.2 Algoritmo propuesto

En base al algoritmo de Solarski expuesto al final del capítulo dos y a las mejoras presentadas en el apartado anterior se propone el siguiente algoritmo 3.1.

Algoritmo 3.1: Algoritmo de actualización para sistemas de replicación pasiva

```
1:  Procedure Actualización
2:  numReplicas ← N {Siendo N el número inicial de réplicas}
3:  numRepLanzadas ← 0
4:  failOver ← false
5:  while numrepLanzadas ≠ numReplicas do
6:     lanzarNuevaRéplica()
7:     numRepLanzadas ← numRepLanzadas + 1
8:     if falla réplica nueva {Réplica con el código nuevo} then
9:        numRepLanzadas ← numRepLanzadas − 1
10:    end if
11:    if falla el primario then
12:       elegirPrimario() {Elección en el grupo de réplicas viejas siempre que sea
          posible}
13:       if ∄ réplica vieja then
14:          failOver ← true
15:       end if
16:    end if
17: end while
18: if !failOver then
19:    forzarFalloPrimario()
20:    failOver ← true
21: end if
```

El algoritmo cumple con las mejoras propuestas, pero está condicionado a suposición de que estamos usando una capa de transporte que garantiza tanto

la entrega como el correcto orden de llegada de los mensajes. El algoritmo no contempla tranferencias o transformaciones de estado porque corren a cargo del primario y de las nuevas réplicas respectivamente.

CAPÍTULO 4
Implementación del algoritmo propuesto en un sistema distribuido

Para demostrar la efectividad de la propuesta hecha en el capítulo anterior se va a implementar en un sistema distribuido creado especialmente para ello. No se van a entrar en los detalles del código porque no es el objetivo del trabajo, tampoco se va a añadir el código en un apéndice porque tampoco es precisamente corto y desde mi punto de vista es un gasto de papel innecesario. Por ese motivo se deja a disposición del lector la siguiente dirección donde puede encontrar todo el código disponible https://github.com/Fantoni0/Proyecto.

4.1 Tecnologías empleadas

Para el desarrollo del código se han utilizado muchos de los conceptos vistos a lo largo de la carrera como concurrencia, programación orientada a eventos, algoritmia, diferentes protocolos para el envío de mensajes etc. Pero a la hora de concretar tecnologías podemos resumirlas en la siguiente lista:

Javascript/Nodejs se ha utilizado por su dinamismo, flexibilidad, portabilidad y alto nivel de abstracción. Javascript también proporciona orientación a objetos e interacciones asincrónicas mediante el uso de *callbacks* y de promesas. Concretamente se ha utilizado el entorno conocido como Node.js que está orientada a la creación de aplicaciones desde el punto de vista servidor, justamente el objetivo de este proyecto. Permite una gestión de la E/S ligera y que no se bloquea, esto se traduce en la capacidad de gestionar una gran cantidad de conexiones con una sobrecarga mínima. Lo que lo hace perfecto para la gestión de conexiones por parte de un servidor. Además Nodejs a través de la biblioteca *child-process* [1] permite lanzar nuevos procesos desde código. Para ello se emplea la función *fork()*, aunque no debe confundirse con el *fork* de Unix. No se crea una copia del proceso padre, se crea un proceso totalmente nuevo a partir de los argumentos que se le pasan.

[1] https://nodejs.org/api/child_process.html

ZeroMQ es una biblioteca que actúa como *middleware* de intercambio de mensajes. Proporciona una serie de *sockets* para enviar y recibir mensajes de manera fiable. Existen diferentes patrones dependiendo de la funcionalidad que busquemos implementar: difusión, mensajes unidireccionales etc. Facilita la tarea del programador y evita sobrecargas innecesarias. Se presentan a continuación los dos patrones usados en la implementación:

1. **Patrón REQ-REP(*request-reply*):** Es un patrón de comunicación síncrono en el que el *socket* REQ realiza una petición al *socket* REP. Todas las peticiones deben de ser respondidas para evitar que el *socket* REP se bloquee.
2. **Patrón PUB-SUB(*publish/subscribe*):** Patrón orientado a los mensajes de difusión. Diferentes *sockets* del tipo SUB se suscriben al *socket* PUB. Cuando el *socket* PUB manda un mensaje este se difunde a todos los *sockets* SUB suscritos.

4.2 Relajaciones del modelo pasivo

En los objetivos introducidos en el capítulo uno ya se menciona que el objetivo del trabajo es proponer una solución teórica para la actualización que no comprometa la disponiblidad. Al ser este el objetivo y no el de crear un sistema distribuido realista se relajan ciertas restricciones del modelo pasivo para facilitar la implantación del algoritmo de actualización:

1. El servicio de nombres encargado de proporcionar y actualizar la dirección del primario no se ha replicado. En un sistema más realista debería estarlo.

2. Los fallos en las réplicas están simulados y se tienen en cuenta sólo durante el algoritmo de actualización. Si un fallo ocurriera de manera inesperada y/o fuera del proceso de actualización no sería gestionado. Obviamente cualquier intento de sistema realista debería de gestionarlos de una manera consistente independientemente de cómo y cuándo ocurran.

3. El problema de pertenencia descrito en la introducción se ha resuelto usando una entidad centralizada que se encarga de gestionar los grupos de réplicas. En una implementación con objetivos más ambiciosos la gestión debería implementarse en cada uno de los procesos mediante un algoritmo distribuido y ayudarse del mecanismo de detección de fallos para mantener los grupos de pertenencia actualizados.

4.3 Estructura del código

El proyecto de prueba que se ha decidido implementar es el de una pequeña tienda online. Los servidores mantienen el estado de la tienda, en este caso el catálogo, y los clientes se conectan al servidor primario para pedir referencias, comprar o devolver productos. En la tienda intervienen los siguientes archivos:

- **auxFunctions.js:** Es una librería que algunas clases importan. Contiene funciones para crear enteros, cadenas y números de coma flotante *aleatorios*, además hay una función que devuelve el primer estado de la tienda en caso de que no se haya hecho ninguna modificación. Se hace con el fin de que las réplicas iniciales empiecen con una versión básica de la tienda.

- **domainName.js:** Es el servicio de nombres. Dispone de un *socket* REP para responder a las peticiones. Permite dos tipos de solicitudes: solicitar la dirección del primario y modificarla.

- **updateManager.js:** Es la capa encargada de simular la caída de réplicas y gestionar sus fallos. También se encarga de resolver el problema de pertenencia y de iniciar el proceso de actualización. Dispone de un *socket* REQ para comunicarse con el servicio de nombres y actualizar la nueva dirección del primario, de un *socket* PUB para comunicarse con todas las réplicas del servicio e informarles del cambio de primario entre o del fallo de un nodo concreto. Por último también tiene un *socket* REP para responder a los mensajes iniciales de los servidores que se registran al inicio de su ejecución.

- **serverX.js:** Con independencia de la versión de la clase server de la que hablemos todas se comunican de la misma manera, más adelante se hablará de las diferencias entre versiones. En la clase servidor hay que diferenciar entre los sockets que sólo usará el primario, los sockets que únicamente usará una réplica secundaria y los sockets que empleará con independencia de si actúa como réplica primaria o secundaria. El primario usará exclusivamente los siguientes *sockets*:

  1. Un *socket* PUB con el que comunica a las réplicas secundarias las actualizaciones que deben de realizar en su catálogo para mantenerse actualizadas. Si la operación no provoca cambios el primario no envía nada (una operación del tipo *get* por ejemplo).
  2. Un *socket* REP que se usa para responder a las peticiones de los clientes. También se utiliza para responder a las peticiones de estado de las réplicas secundarias.

  Las réplicas pasivas tienen en uso exclusivo:

  1. Un *socket* SUB que utilizan para suscribirse a las actualizaciones del primario.
  2. Un *socket* REQ que utilizan para solicitar al primario por el estado de la tienda.

  El resto de *sockets* se usan indistintamente del rol:

  1. Un *socket* SUB para suscribirse a las actualizaciones de updateManager.js, estas actualizaciones pueden informar de la caída del primario y la identidad del nuevo, o mandar un mensaje de fallo en la réplica con identificador X.
  2. Un *socket* REQ para hacer peticiones a updateManager.js . La única petición que se hace a través de este *socket* se realiza al principio. El servidor manda un mensaje con su servicio y su configuración para ser

incluido en el grupo correspondiente. La capa de actualización le responde asignándole una identidad, informándole de si es o no el primario y la configuración de este último.

Con respecto a las diferentes versiones de los servidores, las diferencias son mínimas:

1. La clase server.js se desarrolló primero y el resto están basadas en ella. Se introdujo un error a propósito para justificar la actualización a la siguiene versión. Cuando un cliente nos manda un número de referencia para pedir la información de un producto concreto esta versión ignora el identificador y siempre devuelve el primer objeto de la tienda.
2. La clase server2.js es exactamente igual que la anterior pero resuelve el problema introducido. Ahora devuelve el objeto asociado al número de referencia proporcionado por el cliente. Es decir, introduce un cambio de funcionalidad con respecto a la versión inicial.
3. La clase server3.js es igual a la clase anterior con la diferencia de que por políticas de la tienda ahora los objetos del cátálogo deben añadir un par de características más como el color y el país de procedencia. Lo primero que hace al recibir el estado del primario es calcular la transformación de estado correspondiente. Se encarga de proporcionar compatibilidad hacia atrás, es decir es capaz de procesar las actualizaciones del primario de la versión anterior. Introduce un cambio de funcionalidad y de estado con respecto a la versión inicial.

- **client.js:** Es un cliente con dos *sockets* REQ. Uno de ellos lo usa para preguntar al servicio de nombres por la dirección del primario y el segundo lo utiliza para realizar propiamente las solicitudes al primario. Existen 4 tipos de operaciones permitidas:

  1. *Buy*: Compra N unidades del producto con el número de referencia indicado.
  2. *Return*: Devuelve N unidades del producto con el número de referencia indicado.
  3. *Get*: Devuelve el objeto asociado al número de referencia indicado.
  4. *getAll*: Devuelve el catálogo completo de la tienda. Es la primera petición que hacen los clientes para saber qué productos hay disponibles.

- **administrator.js:** Es un cliente al igual que Client.js, tiene el mismo tipo y número de *sockets* que se utilizan para lo mismo. La principal diferencia es que al actuar como administrador tiene acceso a operaciones que modifican el estado de la tienda. Del mismo modo que el cliente simple lo primero que hace es solicitar el catálogo completo. Permite dos operaciones:

  1. *Create*: Se crea un objeto aleatorio que se añade al catálogo de la tienda.
  2. *Delete*: Dado un número de referencia cualquiera se borra el objeto asociado a él en la tienda.

- **failover.js:** Proceso especial que se usa para provocar el fallo de una réplica. Si no se le pasa ningún identificador por defecto provocará que el primario falle. En el caso de que se le pase como argumento un identificador se tratará de provocar el fallo de esa réplica concreta. Dispone de un *socket* REQ para solicitar a updateManager.js que provoque y gestione el fallo deseado.

- **initiateUpdate.js:** Proceso especial que se usa para iniciar el proceso de actualización. Acepta como parámetro el nombre del nuevo proceso a lanzar como servidor y se lo envía a la capa de gestión de la actualización usando su *socket* REQ. UpdateManager.js es capaz de detectar automáticamente cambios de versión en función del nombre de fichero que se le envía y comienza el proceso de actualización.

4.4 Ejemplo completo

Después de introducir en la sección anterior cuales son los archivos desarrollados y cual es su función vamos a proponer un ejemplo completo para que quede todo más claro y se muestre perfectamente el proceso de actualización. Para empezar mostramos el proceso de despliegue en la imagen 4.1.

1. Desplegamos el servicio de nombres.

2. Lanzamos la capa de control de la actualización y de gestión de pertenencia (de ahora en adelante capa de gestión).

3. Se despliegan tres servidores.

4. Lo primero que hacen los servidores es registrarse con su servicio en la capa de gestión a través del *socket* REQ habilitado para ello. La capa de gestión registra los servidores y les responde informando de quien es el primario, las configuraciones necesarias para conectarse con él y también de la dirección del *socket* PUB de la capa de gestión . Los servidores toman consciencia de su identindad y de su rol y preparan los *sockets* en función del papel que desempeñan. [2]

5. La capa de gestión registra al primario en el servicio de nombres.

6. Las réplicas secundarias se conectan al primario para solicitar el estado de la tienda. El primario responde (recordemos que obtiene la instanciación inicial de la biblioteca de funciones auxiliares) y las réplicas secundarias actualizan su tienda interna.

7. Se lanzan un cliente y un administrador que lo primero que hacen es preguntar al servicio de nombres por el primario. En cuanto este les responde reconfiguran su segundo *socket* y se preparán para comunicarse con el primario.

[2] Cuando se haga una petición desde un *socket* REQ a uno REP se mostrará sólo el mensaje de ida, aunque el mensaje de respuesta también esté implícito. Se hace con el fin de no sobrecargar el diagrama.

8. Los clientes hacen sus solicitudes al primario.
9. El primario procesa las solicitudes, calcula los cambios y se los comunica a las réplicas secundarias usando su *socket* PUB.
10. El primario responde a los clientes y las réplicas secundarias aplican las actualizaciones mandadas por el primario.
11. Se lanza el simulador de fallos con la intención de eliminar a la réplica secundaria con identificador 3. Se le manda el mensaje a la capa de gestión.
12. La capa de gestión elimina a la réplica del grupo y manda un mensaje de difusión a todas las réplicas informando el cese de esa réplica.
13. El nodo cuyo identificador coincide con el del mensaje de fallo termina su ejecución.

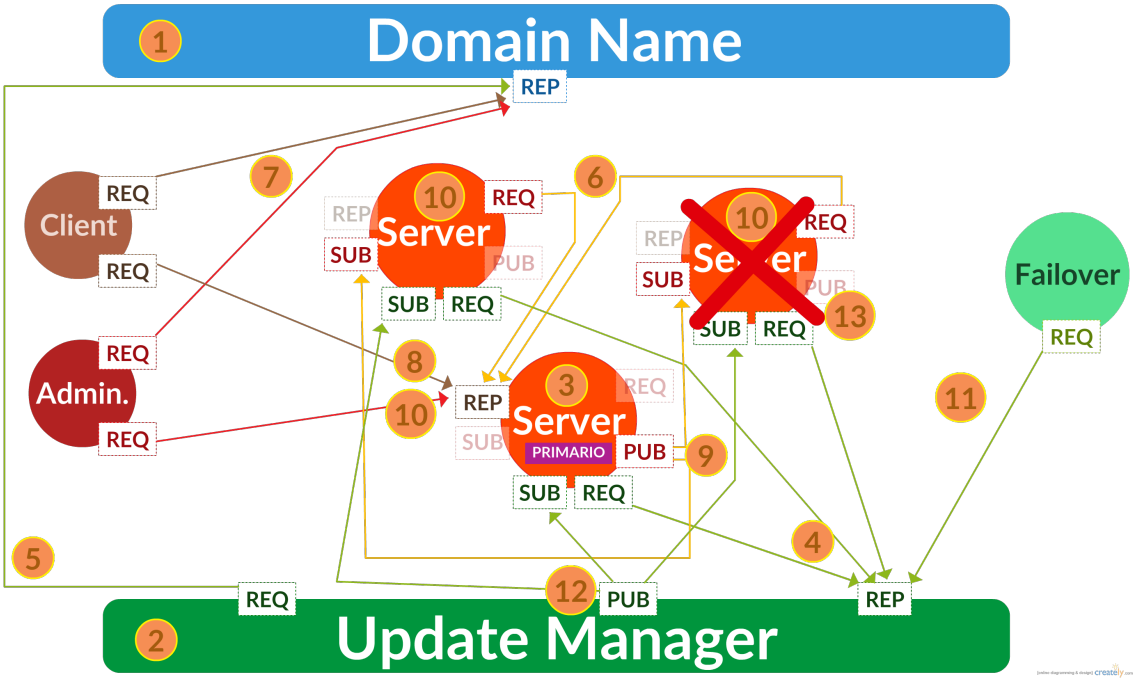

Figura 4.1: Ejemplo de funcionamiento.

Ahora que ya ha sido presentado el proceso de despliegue y como funcionaría el servicio de manera normal vamos a mostrar como sería el proceso de actualización una vez el servicio ya estuviera funcionando como se ha detallado. Se va a mostrar cómo sería el proceso sin fallos, porque ya se explicó como sería su gestión en el capítulo anterior. Además la gestión la hace el archivo updateManager.js a nivel interno y no es algo que se pueda mostar fácilmente en un diagrama.

1. Se lanza el proceso initiateUpdate.js para iniciar el proceso de actualización. Se le manda un mensaje a la capa de gestión con el nombre del nuevo proceso que va a sustituir a la versión actual.
2. La capa de gestión inicia el algoritmo. De manera iterativa va a ir lanzando una nueva réplica con la versión nueva del código y a eliminar una antigua réplica secundaria.

3. Se elimina la primera réplica secundaria a través del *socket* PUB. Se lanza una nueva como réplica pasiva. [3]

4. Se elimina la segunda réplica secundaria a través del *socket* PUB. Se lanza una nueva como réplica pasiva.

5. Se lanza una nueva réplica como pasiva, se fuerza el fallo del primario (aún sin actualizar) y se escoge un primario de entre las réplicas nuevas. El nuevo primario reconfigura sus *sockets* para adaptarse a su nuevo rol.

6. La capa de gestión actualiza la dirección del primario en el servicio de nombres.

7. Los clientes detectan que el primario no les está respondiendo, consultan al servicio de nombres y actualizan sus configuraciones para porder seguir interactuando con el servicio.

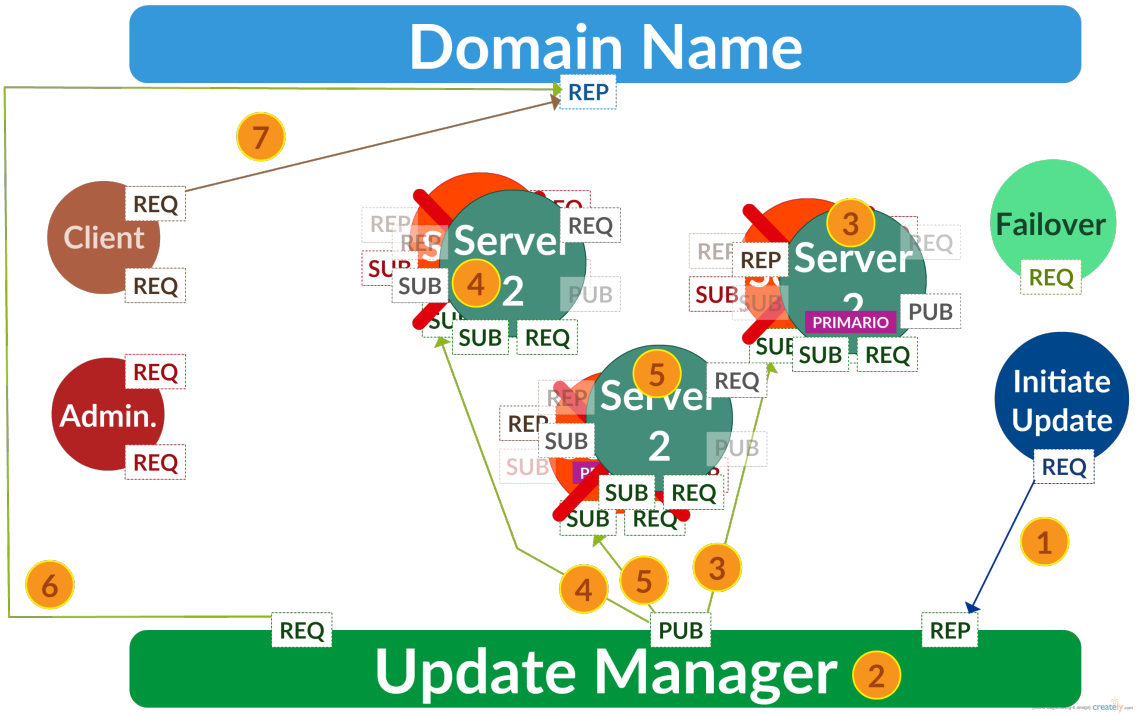

Figura 4.2: Ejemplo del proceso de actualización.

[3]Las nuevas réplicas se registrarían en la capa de gestión y preguntarían al primario por el estado actual como se mostró en 4.1. No se han incluido en este diagrama para no sobrecargarlo

CAPÍTULO 5
Conclusiones

5.1 Evaluación de objetivos

Ahora que ya se ha presentado el sistema distribuido desarrollado, cómo funciona y el comportamiento del proceso de actualización es momento de estudiar si realmente se han cumplido los objetivos presentados en la introducción.

- **¿Se ha conseguido adaptar una solución teórica de actualización que no comprometa la actualización?** Sí, en base al algoritmo de Solarski 2.1 se ha conseguido proponer una versión ampliada 3.1 que asegura el nivel de replicación y que de una manera más clara explica cómo se gestionarían los fallos durante el proceso de ejecución.

- **¿Se ha conseguido implementar la propuesta teórica utilizando tecnologías y conocimientos aprendidos a lo largo de la carrera ?** Sí, se ha conseguido realizar una implementación en un sistema distribuido desarrollado utilizando tecnologías como Javascript y Nodejs. Es cierto que se han relajado ciertos aspectos del modelo pasivo como la detección de fallos en base a las suposiciones que se explicaron en la introducción.

- **¿El sistema desarrollado es capaz de soportar actualizaciones que impliquen un cambio en la funcionalidad?** Sí, diferentes versiones que varían tan sólo en la lógica interna pueden co-existir durante el proceso de actualización ya que no provocan inconsistencias y ambas son capaces de entender las mismas operaciones.

- **¿El sistema desarrollado es capaz de soportar actualizaciones que impliquen un cambio en la funcionalidad y/o el estado interno de las réplicas?** Sí, se delega las transformaciones de estado a las nuevas réplicas que son las encargadas de proporcionar la compatibilidad con las versiones anteriores. Aunque existan réplicas con distintos estados durante la actualización ambas versiones son capaces de procesar las solicitudes/actualizaciones evitando así la inconsistencia en el sistema. Al final de la actualización la información será coherente en todo el sistema.

- **¿El sistema desarrollado es capaz de soportar actualizaciones que impliquen un cambio de interfaces y/o en la funcionalidad y/o en el estado?**

No, por falta de tiempo no se ha podido implementar esta parte. A nivel teórico para poder permitir que dos versiones en las que hay diferencia de interfaces cohabiten durante el proceso de actualización habría que actualizar en dos pasos [14]. Primero se actualizaría a una versión intermedia que fuera capaz de procesar tanto solicitudes antiguas como nuevas, mientras existieran clientes que sólo conocieran la versión antigua del servicio, debería mantener esta versión intermedia. Cuando la actualización finalizara y todas las réplicas fueran capaces de entender las nuevas peticiones se comenzaría a procesar las solicitudes con el nuevo código que ha provocado el cambio de interfaces. A continuación comenzaría la segunda actualización en la que se introducirían las réplicas que sólo trabajarían con la nueva versión y sustituirían a las que son capaces de funcionar con las dos versiones. El proceso de actualización transcurriría sin incidentes ya que ambas versiones pueden entenderse.

- **¿Se han alcanzado los objetivos buscando la máxima eficiencia posible?** Sí. Para medir si el proceso de actualización es eficiente o no primero debemos de estudiar el tiempo medio de respuesta del servicio sin llevar a cabo ninguna actualización. Para ello se ha realizado el siguiente experimento: durante un minuto se mandan una media de 50 peticiones por segundo al servicio, se calcula la media para cada segundo y se obtienen los datos representados en 5.1. Aclarar que en todas las gráficas se han eliminado los datos del primer segundo porque sus valores medios eran exhorbitados comparado con el resto de valores (10.37 ms). Esto se debe a que las primeras peticiones que se realizan se ven afectadas por el periodo de estabilización que necesita ZeroMQ para iniciar los sockets, asignar memoria, calcular el tamaño de los *buffers* etc. Esto provoca que los tiempos se disparen y *contaminen* las gráficas, por dicho motivo se ha decidido eliminar los tiempos de respuesta del primer segundo.

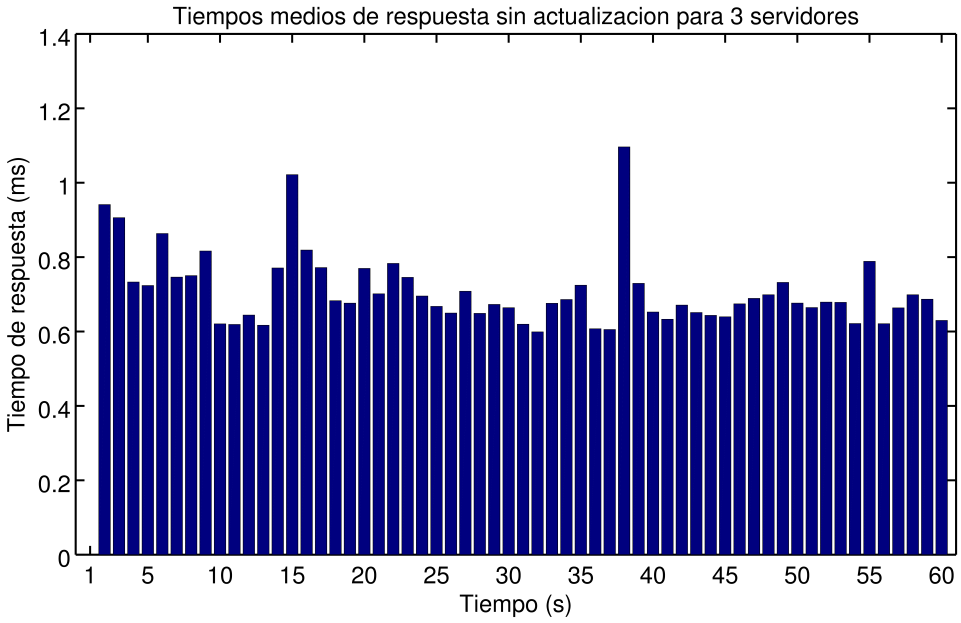

Figura 5.1: Media por segundo del tiempo de respuesta.

Con estos resultados ya podemos llevar a cabo el siguiente paso lógico. Realizar el mismo experimento provocando una actualización entre medias para obervar si afecta a los tiempos medios. En este caso se inicia la actualización del servicio después de 10 segundos. Si comparamos los resultados con los obtenidos en el experimento anterior obtenemos la siguiente gráfica 5.2.

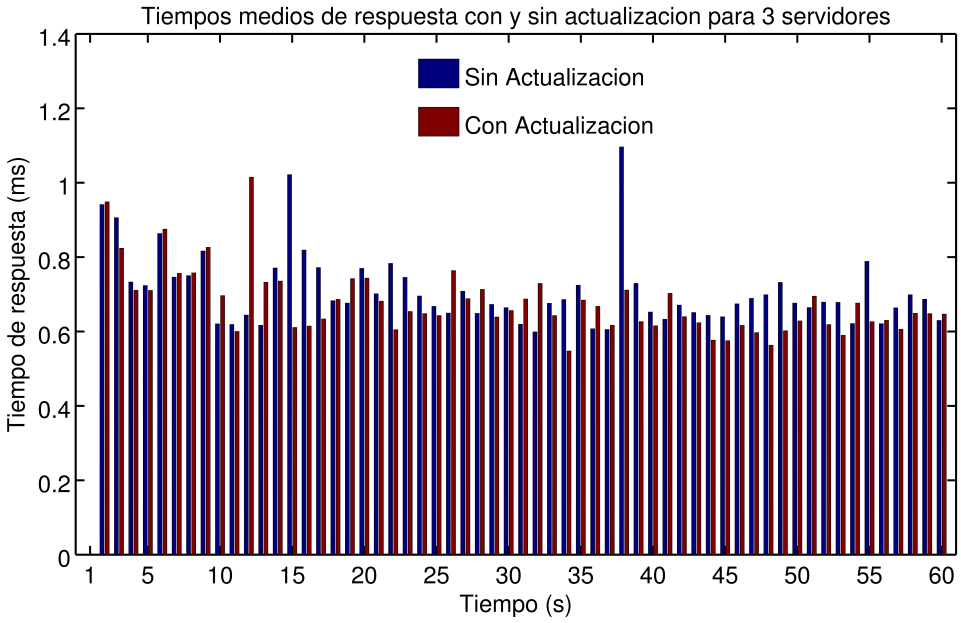

Figura 5.2: Comparación de tiempos de respuesta con y sin actualización.

Como se puede observar hay pequeñas variaciones que se pueden deber a la aleatoriedad del tipo de peticiones y a las condiciones exactas en las que se realizó cada experimento. No obstante no hay una variación sistemática y significativa en los tiempos como para pensar que el proceso de actualización afecta negativamente a los tiempos de respuesta. **Por este motivo se puede afirmar que se han logrado los objetivos de una manera eficiente**. A pesar de ello hay que tener en cuenta que cada vez que se realiza una actualización del sistema se elige un nuevo primario con la reconfiguración del cliente que eso conlleva. El cliente debe consultar de nuevo al servicio de nombres y preguntar por el catálogo al nuevo primario, por eso aunque las diferencias sean mínimas hay que tener en cuenta que la actualización supone un retraso medio de 2.44 ms. Es un tiempo más que aceptable si tenemos en cuenta que nos permite resolver la indisponibilidad asociada al proceso de actualización, pero hay que tenerlo en cuenta para no caer en el error de pensar que el algoritmo implementado está libre de coste.
Para demostrar que la actualización es eficiente con independencia del número de réplicas se ha realizado el mismo experimento con más réplicas. Como se puede apreciar en 5.3 más allá de un ligero incremento en el tiempo de respuesta con respecto a las pruebas anteriores no se producen cambios y los tiempos de los experimentos con y sin actualización se mantienen similares.

Por último se presenta la gráfica 5.4 mostrando que, como se esperaba, el tiempo necesario para llevar a cabo el algoritmo de actualización depende

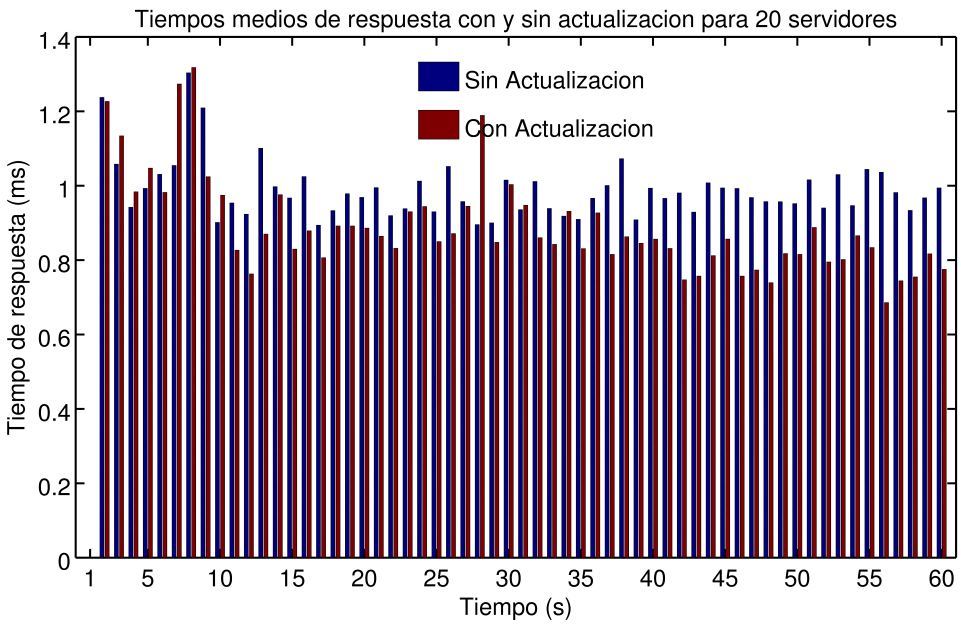

Figura 5.3: Comparación de tiempos de respuesta con y sin actualización para 20 servidores.

de manera lineal del nivel de replicación (número total de réplicas) ofrecido por el servicio. Si bien es cierto que estos tiempos están subordinados a los intervalos usados en el código para dar tiempo a las nuevas réplicas a poder registrarse en la capa de pertenencia.

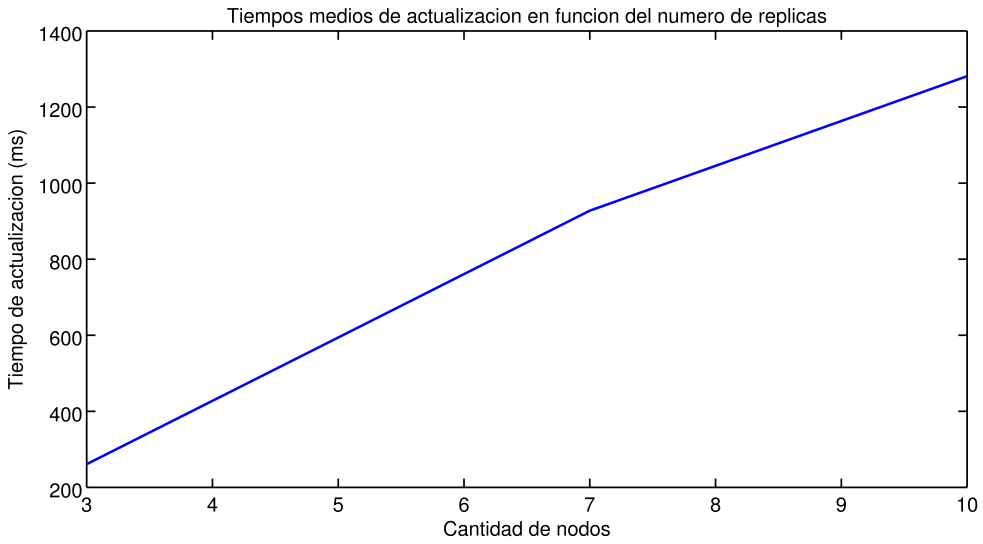

Figura 5.4: Comparación de tiempos de actualización en función del nº de réplicas.

5.2 Resultados de aprendizaje

Gracias a la realización del trabajo y a los problemas que he resuelto durante el mismo creo que he aprendido y/o reforzado las siguientes competencias:

1. Un mejor y más completo conocimiento del funcionamiento y las caracterizaciones del modelo pasivo. Reforzado gracias a la implementación que he tenido que realizar para implantar el protocolo de actualización.

2. Un repaso de la herramienta de comunicación asincrónica que es ZeroMQ. Aunque los patrones usados no han sido más complicados que los vistos a lo largo de la carrera sí que he tenido que realizar una valoración de qué componentes eran los más adecuados para implementar el modelo de replicación pasiva.

3. Un refuerzo de la programación asíncrona con Javascript/Nodejs ya vista en TSR. Cosas que sabía han mejorado, cosas que creía saber han sido explicadas y nuevos conocimientos han sido adquiridos.

4. La capacidad de buscar y consultar fuentes bibliográficas sobre nuevos temas. Diría que esta es la competencia que más valoro, gracias a los teoremas y paradigmas explicados durante estos cuatro años y a la ayuda de mi tutor he podido aprender sobre temas que desconocía leyendo artículos científicos relacionados. Representa la capacidad de ser proactivo y usar lo que ya se conoce para descubrir lo nuevo.

Los problemas que he experimentado han sido debidos a la naturaleza asíncrona del proyecto y a que muchas veces he dado por supuesto cosas que no eran como yo creía. Los problemas más graves han sido:

- Diferencias significativas con el comportamiento de la aplicación si la ejecutaba en una máquina virtual o en un sistema anfitrión. La máquina virtual en ocasiones me ha creado problemas donde no los había.

- La gestión de fallos y de la transferencia de estado en las réplicas me fue difícil de implementar y tuve que cambiar de estrategia alguna vez debido a que mis estimaciones iniciales no habían sido del todo realistas.

CAPÍTULO 6
Agradecimientos

Quisiera dar las gracias a todos los profesores que me han dado clase a lo largo de la carrera y se han preocupado no sólo de que aprobara sino también de que aprendiera.
Gracias a mis familiares y amigos por soportarme cuando me faltaban horas de sueño y el código no hacía lo que yo quería que hiciese.
Y gracias especialmente a mi tutor Francisco D. Muñoz Escoí por ayudarme, guiarme y resolver todas mis dudas a lo largo de este proyecto. Y por responder a todos mis correos, incluidos los enviados a horas intempestivas.
Gracias.

Bibliografía

[1] Peter Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering, San Francisco, California, USA, October 13-15*, pages 562–570, 1976.

[2] Philip A. Bernstein. Middleware: A model for distributed system services. *Commun. ACM*, 39(2):86–98, 1996.

[3] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Optimal primary-backup protocols. In *6^{th} International Workshop on Distributed Algorithms and Graphs (WDAG), Haifa, Israel*, pages 362–378, November 1992.

[4] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. In *15^{th} Annual ACM Symposium on Principles of Distributed Computing (PODC), Philadelphia, Pennsylvania, USA*, pages 322–330, May 1996.

[5] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[6] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[7] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[8] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[9] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. ACM Press, 2nd edition, 1993. ISBN 0-201-62427-3.

[10] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[11] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[12] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. A survey of dynamic software updating. *Journal of Software: Evolution and Process*, 25(5):535–568, 2013.

[13] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, 1^a edition, 2002. ISBN 978-0130888938.

[14] L. A. Tewksbury, Louise E. Moser, and P. M. Melliar-Smith. Live upgrade techniques for CORBA applications. In *3rd International Working Conference on Distributed Applications and Interoperable Systems (DAIS)*, pages 257–271, Kraków, Poland, September 2001.