Tolerancia a fallos para la alta disponibilidad de software de red de alto rendimiento con estados

Pablo Neira Ayuso

<pneira@lsi.us.es>
Departamento de Lenguajes y Sistemas Informáticos
Avda. Reina Mercedes, s/n - 41012 SEVILLA - Andalucía - España

Supervisado por Rafael M. Gasca <gasca@lsi.us.es>

Índice general

Capítulo 1

Introducción

Los servicios de red se despliegan normalmente sobre plataformas *off-the-shelf* que suelen carecen de mecanismos de detección de fallos. Es por ello que son susceptibles de sufrir periodos de indisponibilidad debido a fallos en el sistema. Las soluciones clásicas propuestas en las últimas décadas proponen el uso de mecanismos de redundancia física y monitorización para implementar alta disponibilidad. Concretamente, la idea consiste en desplegar el mismo servicio sobre distintas réplicas y un protocolo de consenso garantiza que, en cualquier instante de tiempo, al menos una de ellas está procesando las peticiones proveniente de los clientes. Si por cualquier razón la réplica activa sufre un fallo, una nueva réplica que estaba actuando como respaldo se selecciona para continuar proporcionando el servicio.

Desafortunadamente, la redundancia física no es suficiente para garantizar la disponibilidad de un servicio implementado por un software con estados puesto que la réplica desconoce los estados de las peticiones que estaba manejando la réplica que ha sufrido un fallo. Es por ello que las réplicas de respaldo no pueden recuperar el servicio apropiadamente puesto que desconocen el estado de las peticiones circulantes. Por esta razón, se debe garantizar que la probabilidad de que los estados sobrevivan a fallos sea alta con el fin de ofrecer una alta disponibilidad.

Durante décadas, las primitivas atómicas de broadcast y los esquemas de replicación estudiados en bases de datos aseguran que un conjunto de réplicas se mantendrán consistentes siempre que apliquen las mismas transacciones en el mismo orden [5]. Sin embargo, tales esquemas basados en transacciones síncronas introducen un retardo considerable en las respuestas a los clientes, un precio a pagar para garantizar que todas las réplicas son copias equivalentes. Esto hace que dichos esquemas no sean aplicables a otros escenarios en los que la principal preocupación sea ofrecer un alto rendimiento.

En este trabajo proponemos una arquitectura y una serie de mejoras en entornos en los que el rendimiento y la disponibilidad sean la principal preocupación: Se deben garantizar respuestas rápidas a los clientes a la vez que agilidad en la recuperación de fallos. La naturaleza de estos escenarios hace particularmente difícil la definición e implementación de una solución completa.

Algunos ejemplos de software de red con estados presentes en nuestra vida diaria en los que el alto rendimiento sea la principal preocupación son:

- Nueva generación de firewalls y routers, los cuales definen una política de filtrado basada en el estado de la conexión frente a las aproximaciones estáticas de la primera generación.

- Redes privadas virtuales, en inglés *Virtual Private Network* (VPN), que ofrecen túneles seguros a través de internet para comunicar dos redes distantes que pertenecen a la misma organización.

- Telefonía por internet, también conocido por VoIP, que permite a dos extremos realizar llamadas telefónicas a través de internet.

- Redes activas, que permiten a sus usuarios inyectar programas adaptados que despliegan algún tipo de servicio sobre el tráfico que circula por la red [14].

Aunque este trabajo se centra particularmente en los firewalls con estados como caso concreto de estudio (Véase Sección 2.2), hemos observado que la semántica de los servicios anteriormente enumerados es similar, por tanto, las contribuciones descritas en este trabajo son igualmente aplicables a éstos.

1.1. Objetivos

El objetivo general de este trabajo es el de proponer mejoras para aumentar la disponibilidad del software con estados en entornos en los que el alto rendimiento es un requisito a satisfacer. Ofrecer una solución completa de alta disponibilidad en estos entornos se convierte en una tarea particularmente difícil puesto que es necesario evaluar las técnicas propuestas con el fin de asegurar que el servicio no reduce el rendimiento ofrecido. Las metas de este trabajo se pueden resumir en los siguientes puntos:

a) Formalizar el problema planteado.

b) Emplear plataformas de bajo coste, también conocidas por sistemas *off-the-shelf* (OTS).

c) Reutilizar las soluciones existentes de tolerancia a fallos y alta disponibilidad que han sido planteadas en las últimas décadas.

d) Definir de una arquitectura sencilla, modular y extensible, que permita configuraciones avanzadas de compartición y balanceo de carga.

e) Desarrollar esquemas de replicación que se adapten a los requisitos impuestos en este escenario.

f) Proponer optimizaciones a la alta disponibilidad basadas en técnicas de detección y diagnosis de errores, y aprendizaje supervisado.

g) Ofrecer respuestas rápidas a los clientes.

h) Garantizar una recuperación rápida de fallos.

1.2. Estructura del documento

Este documento está estructurado de la siguiente forma:

- En la Sección 1 se ha realizado una breve introducción a la problemática a abordar en este documento, así como una serie de ejemplos prácticos de software de red de alto rendimiento con estados.

- A continuación, la sección 2 formaliza el modelo de software con estados empleado como referencia a lo largo de este trabajo, así como el entorno de alta disponibilidad compuesto de un conjunto de réplicas del servicio. Un caso práctico de software de red de alto rendimiento con estados es descrito en la sección 2.2.

- El documento sigue con la sección 3, que ofrece una panorámica sobre otros trabajos relacionados con el problema abordado en este documento.

- La sección 4 describe una propuesta de una arquitectura flexible y modular sobre la cual se fundamenta la solución de alta disponibilidad propuesta en este trabajo.

- La sección 5 propone una serie de protocolos de replicación confiables y no confiables que tienen como objetivo aumentar la probabilidad de que los estados de las peticiones sobrevivan a posibles fallos.

- En la sección 6 se proponen dos optimizaciones a la replicación con el objetivo de reducir los recursos dedicados a dicha tarea sin que la disponibilidad del servicio desplegado por el software se vea afectada.

- El trabajo concluye con las secciones 7 y 8 en las que se elabora una evaluación preliminar de las soluciones propuestas, conclusiones y trabajos futuros.

Capítulo 2

Modelado del sistema

2.1. Descripción del modelo formalizado

Sea un sistema sobre el que se ejecuta un software con estados. Dicho software atiende a un conjunto de peticiones $\phi = \{R_1, R_2, ..., R_n\}$ donde cada petición R_i está compuesta por una secuencia de subpeticiones $R_i = \{r_0, r_1, ..., r_m\}$. Se establece que el estado S_j de una cierta petición R_i viene determinado por la secuencia de subpeticiones $\gamma = \{r_0, r_1, ..., r_s\}$ que han sido consumadas satisfactoriamente, es decir, una secuencia de subpeticiones tal que para $r_m, r_s : s \leq m$.

Un estado puede ser representado por un conjunto de variables $S_j = \{v_1, v_2, ..., v_t\}$ con un cierto valor conocido, el número de estados posibles vienen representados por un conjunto finito π formado por $k \geq 0$ estados $\pi = \{S_0, S_1, S_2, ...S_k\}$, la evolución de un software con estados es determinista y la secuencia de estados evoluciona de forma predecible (Fig. 2.1).

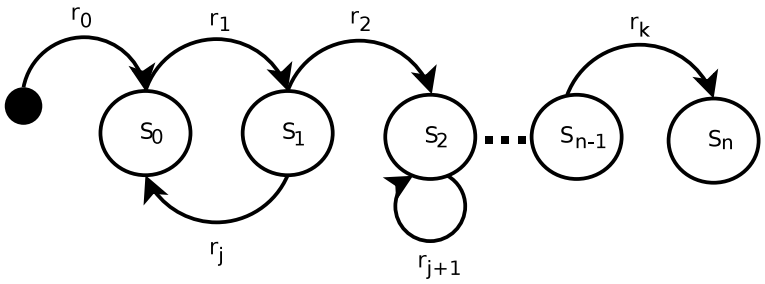

Figura 2.1: Grafo de secuencia de estados

Todo estado S_j dispone de un tiempo máximo de vida T_j, se asume que si el tiempo máximo de vida de un cierto estado es alcanzado, los recursos asociados a la petición R_i que está en dicho estado S_j son liberados, siendo abortada la petición. Dicho tiempo máximo de vida es un parámetro configurable que viene determinado por el propio servicio o el administrador del sistema. La razón de ser de dicho tiempo de vida es la de garantizar que ciertas peticiones bloqueadas no desperdicien recursos de forma indefinida, evitando así posibles ataques de denegación de servicio.

En nuestro modelo, una cierta entrada recibida por el cliente es manejada de una forma u otra dependiendo del estado de la petición asociada a dicha entrada, es decir, el estado actual de la petición es un parámetro que determina de qué forma se despliega el servicio (Fig. 2.2). La ausencia de dicho estado tiene un impacto impredecible en el servicio desplegado.

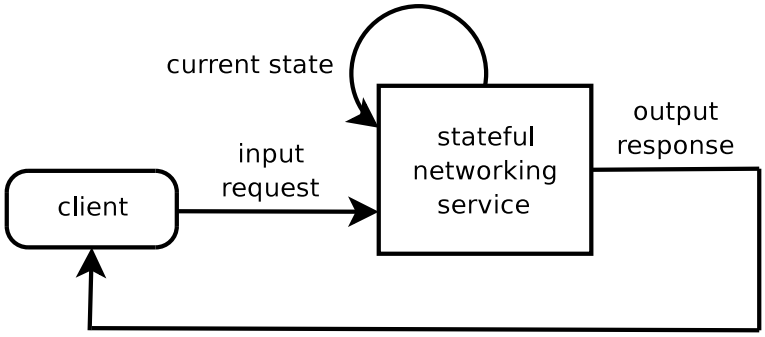

Figura 2.2: Diagrama de servicio de red con estados

De esta forma, podemos concluir que toda entrada puede potencialmente producir un cambio de estado. Esta afirmación se expresa mediante la siguiente fórmula:

$$siguiente\_estado : Estado \times Entrada = Estado$$

Dicha fórmula viene definida en la implementación del software con estados, y puede representarse de la siguiente manera:

$$siguiente\_estado : R_j(Entrada, S_k) = S_m$$

En caso de que $S_k = S_m$, se considera que la entrada no produjo ningún cambio de estado. Obsérvese que los estados previos a S_k no son necesarios para calcular la transición a un estado. Por tanto, únicamente se requiere el estado actual para garantizar que un cierto servicio se despliega correctamente, de esta forma, concluimos que los estados pasados no son necesarios.

Con el objetivo de garantizar la alta disponibilidad del software con estados se emplean técnicas de replicación. Se asume que el sistema está compuesto por, al menos, dos réplicas preparadas para desplegar el servicio. Se dice que una réplica que atiende peticiones provenientes de los clientes es una réplica *activa*, por otro lado, aquella réplica que actúa como posible candidata a reemplazar la réplica activa en caso de fallo recibe el nombre de réplica de *respaldo*. Dichas réplicas se encuentran en el dominio de una red de área local, y la comunicación se realiza a través de un enlace dedicado. Además, se asume la existencia de un software de monitorización no confiable [13] que se encarga de comprobar el estado del proceso. Dicho software decide cuándo migrar de una réplica en fallo a una réplica segura.

Puesto que los cambios de estado alteran la manera en la que el servicio es desplegado, se debe garantizar que los estados sobrevivan a posibles fallos. El concepto de *durabilidad de estados* se define como la probabilidad de que un estado pueda sobrevivir a fallos. Por tanto, una alta durabilidad asegura que el servicio es desplegado apropiadamente a pesar de posibles

fallos en el sistema. En otras palabras, una baja durabilidad supone que no se puede garantizar que las peticiones que están siendo atendidas sean retomadas satisfactoriamente tras un fallo.

Por razones de rendimiento, se asume que los estados se almacenan en un dispositivo de almacenamiento no persistente, ya que los dispositivos persistentes son generalmente lentos, lo cuál tendría un impacto considerable en el rendimiento del sistema. Por tanto, un corte de energía o el apagado inapropiado del sistema produce la desaparición del estado actual de las peticiones que están siendo manejadas.

En nuestro modelo se asumen únicamente fallos de tipo *crash-only*, por tanto, el sistema se comporta de forma predecible hasta que se produce el fallo completo del sistema o *crash*. Los clientes emplean una primitiva unicast confiable mediante la que envían las entradas al software con estados que despliega el servicio. De esta forma, se asume que las peticiones no se pierden nunca, concretamente la primitiva garantiza que si una petición no es atendida en un lapso de tiempo, ésta es vuelta a ser enviada.

2.2. Un caso práctico: Stateful Firewalls

Un firewall es una solución de seguridad perimetral que permite a consultores y expertos en seguridad informática definir políticas de filtrado de tráfico de la red. Los firewall se sitúan en un punto de la red estrátegico, controlando tráfico permitido entre dos redes. La primera generación de firewalls permitía definir dichas políticas mediante un conjunto de reglas definidas a través de una serie de parámetros, también conocidos como descriptores. Dichos descriptores emplean la información disponible en la cabecera de los paquetes que circulan por la red y la contrastan con las reglas definidas, si existe una regla que explícitamente permita la circulación de tal paquete, se le deja paso, de lo contrario, se rechaza.

De esta, la primera generación de firewalls aplicaba un filtrado que se considera sin estados. Esta aproximación estática abre la puerta a posibles ataques de denegación de servicio, tales como ataques de *reset* que permiten a una tercera parte interrumpir una comunicación establecida sin el consentimiento de las dos partes implicadas en la comunicación, o incluso inyectar basura con el fin de corromper los datos transferidos.

La segunda generación de firewalls viene a solucionar los problemas de la anterior generación mediante una aproximación dinámica basada en el mantenimiento del estado del tráfico [35]. Esta nueva generación permite el uso del estado de un flujo como descriptor, pudiendo ser asociado a una regla de filtrado. De esta forma, el firewall puede definir una política de filtrado basada en flujos, y no únicamente en paquetes.

La implementación de firewall con estados altamente disponibles requiere que las réplicas de respaldo mantengan el estado de los flujos manejados por las réplicas activas, de forma que en caso de fallo, la réplica de respaldo seleccionada para reemplazar a la réplica que está fallando pueda continuar desplegando la política de filtrado satisfactoriamente. De lo contrario, es probable que muchos de los flujos establecidos sufran bloqueos y tengan que ser restablecidos [29]. Un ejemplo de una conexión TCP filtrada por un firewall con estados viene representada en la figura 2.3.

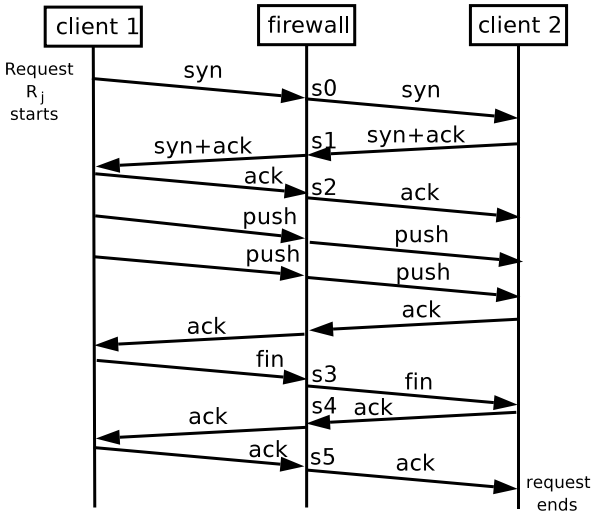

Figura 2.3: Conexión TCP a través de un firewall

Un grafo simplificado de la evolución de los estados por los que pasa una conexión TCP (Fig. 2.4) tiene los siguientes estados:

- Syn sent (S_0): Estado inicial de sincronización, el firewall observa que un cliente pretende establecer una comunicación con otro extremo que probablemente actúe de servidor. El firewall ha observado un paquete con el flag SYN activo.
- Syn recv (S_1): Respuesta satisfactoria al intento de sincronización. El extremo que actúa de servidor responde al cliente que está dispuesta a comenzar una conexión. El firewall observa un paquete con los flags SYN y ACK activos, este estado sólo es alcanzable desde S_0.
- Establish (S_2): Conexión establecida con éxito, ambos extremos están listos para intercambiar información. El firewall ha observado un paquete con el flag ACK activo. A partir de ahora, todos aquellos paquetes que contengan datos, es decir, que tengan el flag PUSH activo, no producen un cambio de estado en la conexión.
- Fin (S_3): Uno de los extremos pretende finalizar la comunicación establecida. El firewall ha observado un paquete con el flag FIN activo, este estado sólo es alcanzable desde el estado S_2.
- Fin Wait (S_4): El extremo que ha recibido el paquete con el flag FIN activo responde con otro con los flags FIN y ACK activos, confirmando que se finaliza la transferencia de datos entre ambos puntos y se procede a liberar todos los recursos asociados a la conexión.
- Closed (S_5): El extremo que comenzó la finalización de la comunicación establecida envía un paquete con el flag ACK activo para confirmar que la conexión ha finalizado satisfactoriamente.

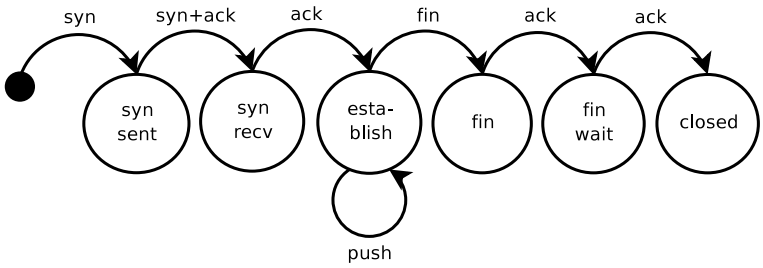

Figura 2.4: Grafo simplificado de estados de una conexión TCP

De esta forma, si una cierta conexión TCP no evoluciona siguiendo la secuencia de estados anteriormente descrita, el firewall, dependiendo de la política de filtrado definida, puede rechazar el paquete al considerarlo un comportamiento al margen de la especificación.

Capítulo 3

Trabajos previos

La idea de implementar soluciones software de replicación para garantizar la alta disponibilidad de un servicio ha sido discutida durante décadas [6] [18] [22] [33] [39]. Básicamente, dichas soluciones justifican la implementación de técnicas de replicación basadas en software en el hecho de que los sistemas *off-the-shelf* cuestan mucho menos que un sistema especializado tolerante a fallos [6] [15] [18] [33]. Dichas técnicas de replicación no ofrecen nuevas funcionalidades en el sistema sino que aumentan la calidad del producto [36]. Y por supuesto, suponen un coste extra en recursos que debe ser evaluado.

La base de toda solución destinada a garantizar una alta disponibilidad de un software se fundamenta en dos conceptos:

- Redundancia: se establecen una o múltiples réplicas que están capacidas para desplegar un mismo servicio, algunas de ellas actúan como réplicas *activas*, otras actúan como réplicas *respaldo* de las anteriormente nombradas réplicas activas. En caso de fallo, un algoritmo de selección de réplica segura, *agreement protocol* en inglés, selecciona un reemplazo que garantice que el servicio continuará siendo desplegado de manera apropiada.
- Monitorización: Se define una serie de mecanismos de monitorización del estado de las réplicas y del servicio para garantizar el correcto funcionamiento de la solución.

3.1. Paradigmas

En la bibliografía se proponen dos metodologías a la hora de abordar el problema de garantizar la tolerancia a fallos de los servicios [6], concretamente son dos arquitecturas:

- *De abajo a arriba*, en inglés *bottom-up* (BU), que implican el diseño de un sistema que dispone de una infraestructura global de detección y corrección de fallos en el hardware.
- *De arriba a abajo*, en inglés *Top-Down* (TD), que implementa un sistema tolerante a fallos sobre una plataforma OTS que carece de mecanismos de tolerancia a fallos.

En la vida real, el paradigma TD se impone al BU, aunque cada vez es más frecuente que las plataformas OTS de bajo coste comiencen a disponer de mecanismos que permitan

identificar y corregir ciertos tipos de fallos sin que se vea afectado el nivel de seguridad del sistema. Además, resulta también frecuente que la tolerancia a fallos en muchos sistemas sea un requisito que surja en etapas de post-implementación, resultando inviable la posibilidad de rediseñar el sistema para cumplir el nuevo requisito. En este trabajo se ofrecen alternativas para ambos paradigmas, tanto el TD como el BU, incluso planteando la posibilidad de definir una arquitectura híbrida aprovechando que el paradigma de las plataformas OTS está cambiando.

3.2. Soluciones ofrecidas por la industria

La industria, en el ámbito de las redes, ofrece múltiples protocolos a partir de los cuales se pueden implementar los conceptos de redundancia y los mecanismos de monitorización necesarios. Todas ellas están basadas en el paradigma TD. De entre las muchas soluciones existentes encontramos dos familias:

- Propietarias y estándares de iure: `VRRP` (Virtual Redundancy Router Protocol[19]), un protocolo que fue inicialmente desarrollado por Cisco y posteriormente estandarizado por IETF. Diseñado para routers, está siendo usando en otros ámbitos, tales como servidores HTTP, FTP, ... Existen implementaciones libres tales como *keepalived* [2]. Aunque el protocolo es un estándar de IEFT, Cisco reclama una serie de patentes sobre ciertos aspectos del protocolo. El protocolo es sencillo, es la base para la mayor parte de las soluciones empleadas en la industria. Ha sido criticado por disponer de unas especificaciones imprecisas [30]. Por otro lado está `CISCO HSRP` (Hot Stand-by Redundancy Protocol [20]), un protocolo propietario implementado en la nueva generación de routers Cisco.

- Opensource, Libres y estándares de facto: `CARP` (Common Address Redundancy Protocol [3] es un protocolo basado en VRRP, no está documentado, su única referencia es la implementación. Surge como respuesta a las reclamaciones de Cisco de las supuestas patentes. El grupo de trabajo de VRRP [4] en el IETF considera este trabajo un proyecto paralelo VRRP. Por otro lado se encuentra `Linux Heartbeat[21]`, una solución de código abierto disponible para cualquier UNIX. De nuevo, la solución carece de una documentación sobre los detalles internos, su código fuente resulta complejo al estudio.

Dichas soluciones ofrecen los mecanismos apropiados para implementar redundancia y monitorización. Los escenarios clásicos donde estas soluciones son desplegadas están compuestos por más de dos réplicas, de las cuales al menos una está activa frente a las otras que actúan como respaldo. Todos los protocolos nombrados anteriormente emplean el concepto de recurso virtual, dicho recurso virtual es asignado a aquellas réplicas activas. En caso de fallo, la solución software selecciona a una nueva réplica a la que se le asignará dicho recurso virtual.

HSRP es similar a VRRP. El grupo de trabajo de VRRP en la IEFT señala que las principales diferencias entre VRRP y HSRP son:

- HSRP dispone de un mensaje denominado *Coup* que permite a una réplica de respaldo pasar a estar activa. Esta nueva funcionalidad permite al administrador seleccionar la réplica que está activa en cualquier momento.

- Los mensajes ICMP Redirect no funcionan en HSRP.

- VRRP es más ligero y sencillo que HSRP.

Para concluir, las soluciones ofrecidas por la industria ignoran si un software carece o no de estados, por tanto, no son suficientes para garantizar la alta disponibilidad de un software con estados. Es por ello que deben ser complementadas por otras soluciones que abarquen los aspectos propios de dicho software.

3.3. Software con estados: Técnicas de Replicación

El software con estados hace necesaria la introducción de algoritmos de replicación. Dichos algoritmos mantienen la consistencia entre las réplicas mediante el intercambio de mensajes de sincronización, de forma que en caso de fallo, la réplica seleccionada para retomar el servicio dispone del estado de las peticiones. Garantizando así que las peticiones pendientes durante el fallo no quedan interrumpidas.

A lo largo de las últimas décadas, han sido dos soluciones las que han tenido un impacto particularmente importante en el campo de la tolerancia a fallos: La aproximación Primario-Respaldo y la replicación basada en Máquinas de Estados.

3.3.1. Primario-Respaldo

En la replicación Primario-Respaldo [7], cada cliente envía sus peticiones a la réplica primaria, que es la que acepta y procesa todas las peticiones recibidas, y propaga los cambios de estado a las réplicas de respaldo, que son aquellas que son candidatas a reemplazar a las réplicas primarias en caso de fallo (Fig. 3.1). La replicación se divide en cuatro etapas: En primer lugar, el cliente realiza una petición a la réplica primaria (Etapa 1), la réplica primaria propaga el cambio de estado a las réplicas mediante una primitiva multicast (Etapa 2), las réplicas de respaldo confirman a la primaria la correcta recepción de dichos cambios de estados (Etapa 3), es entonces cuando la primaria envía la respuesta a la petición realizada por el cliente (Etapa 4). Esta solución es sencilla y elegante.

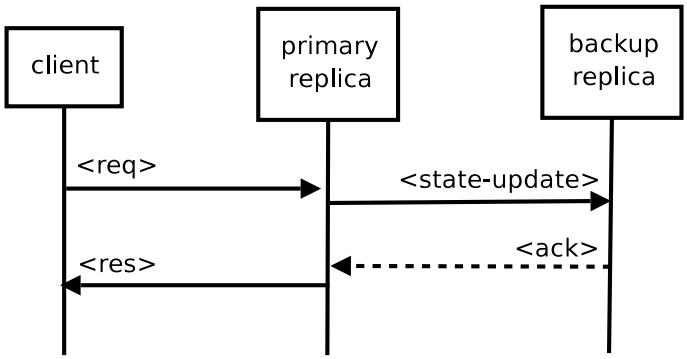

Figura 3.1: Replicación Primario-Respaldo

3.3.2. Replicación basada en Máquina de Estados

En la replicación basada en Máquina de Estados [32] (Fig. 3.2), cada cliente realiza una petición a una de las réplicas de forma aleatoria, tal réplica reenvía la petición a una cierta réplica llamada *secuenciador* que asigna un número de secuencia a la petición recibida. Entonces, se reenvía la petición junto al número de secuencia asignado a todas las réplicas, que confirman el correcto procesamiento de la petición recibida mediante un mensaje, a lo que el secuenciador responde con un mensaje denominado por *estado estable*, es entonces cuando se procede a enviar el resultado del procesamiento al cliente. Esta solución introduce un retardo en la respuesta al cliente debido a que el mensaje de estado estable no es enviado hasta que todas las réplicas no hayan procesado la petición correctamente. Esta arquitectura aumenta el tiempo de respuesta al cliente considerable, por tanto es descartada puesto que se aleja de los objetivos cubiertos por este trabajo.

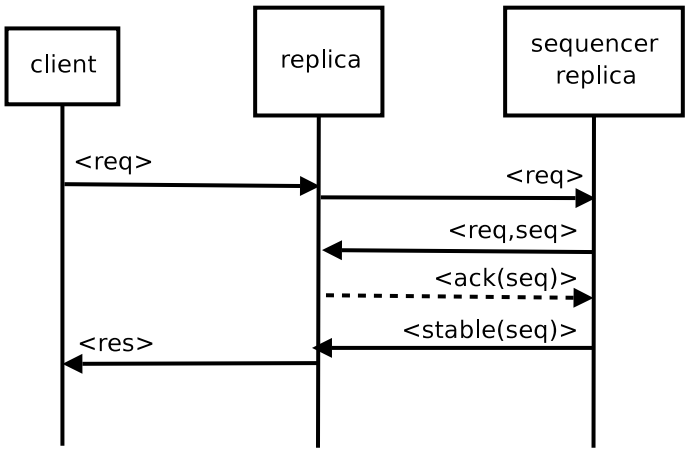

Figura 3.2: Replicación basada en máquina de estados

3.3.3. Soluciones de replicación síncrona versus asíncrona

Hasta ahora, hemos considerado que la replicación realizada por ambas soluciones Primario-Respaldo y Máquina de Estados es síncrona, es decir, el cliente recibe la respuesta a su petición una vez correctamente propagados los cambios de estado a las réplicas de respaldo. Sin embargo, ambas aproximaciones pueden modelarse de manera asíncrona, es decir, la respuesta a una petición proveniente del cliente es enviada antes de que el cambio de estado que ha producido sea propagado a las réplicas de respaldo. Aunque el esquema síncrono garantiza que todos las réplicas son copias equivalentes en cualquier momento, dicho esquema introduce un retardo en los tiempos de respuesta a los clientes. Para nuestros propósitos, el esquema de replicación asíncrono [8] es preferido puesto que ofrece una rápida respuesta a los clientes. De esta forma se reduce la degradación que pueda sufrir el rendimiento del sistema al deplegar la tarea de replicación. Esta ventaja viene con un coste asociado, el aumento la complejidad de la solución propuesta, puesto que resulta dificil saber cuántos estados contiene las réplicas de respaldo en el momento de fallo de una réplica activa.

3.3.4. Replicación basada en RDMA

Recientemente ha sido propuesta una arquitectura de replicación basada en mecanismos de acceso remoto directo a memoria, en inglés *Remote Direct Memory Access* (RDMA) [34]. Dicha tecnología permite la propagación de los estados mediante el movimiento de datos desde la memoria de una réplica que padece una falla a una réplica segura sin necesidad de involucrar la pila de protocolos de red y los mecanismos de mensajería ofrecidos por ésta. Esto teóricamente permite tranferencias de datos a baja latencia y reduce la penalización introducida por la pila de protocolos. La solución propuesta se basa en la migración de los estados de la réplica de respaldo a la réplica segura en el momento en el que se produzca un fallo, lo cual ofrece una solución que consume pocos recursos para garantizar una alta disponibilidad. Sin embargo, son múltiples las limitaciones asociadas a esta solución:

- La aproximación se basa en el supuesto de que el error que se produce en el sistema no está relacionado con la zona de memoria en la que se almacenan los estados.
- La arquitectura propuesta requiere un nuevo componente hardware, una tarjeta inteligente, lo que supone un coste extra,
- Se hace necesario la migración de las soluciones de alta disponibilidad existentes a dicha tarjeta inteligente.
- La falta de madurez de RDMA como tecnología, así como las críticas recibidas por introducir un aumento de rendimiento despreciable con respecto al paradigma *send/receive* basado en mensaje, padecer problemas de escalabilidad para transferencias de datos de gran tamaño e introducir problemas de sincronización en el acceso a la memoria.
- Los mecanismos de RDMA están específicamente diseñados para computación de alto rendimiento, en inglés *high performance computing* (HPC), el uso de esta tecnología en el ámbito de la alta disponibilidad resulta intrusivo.

3.3.5. Replicación orientada al protocolo TCP

Múltiples soluciones basadas en implementaciones tolerantes a fallos del protocolo TCP han sido propuestas [22] [37]. Dichas soluciones mantienen una réplica de respaldo que almacena todos los paquetes TCP recibidos por la réplica primaria mediante técnicas de escucha pasiva, el software encargado de almacenar los paquetes escuchados recibe la denominación de *logger*.

Concretamente, la réplica de respaldo escucha todo el tráfico existente entre la réplica activa y los clientes pero no actúa en la comunicación, únicamente almacena todo el tráfico TCP circulante. Es en caso de fallo cuando la réplica de respaldo toma todos los paquetes almacenados y realiza una repetición de la comunicación completa hasta alcanzar el último estado consistente antes del fallo. Tales soluciones están principalmente enfocadas a implementar servidores TCP tolerantes a fallos, por tanto, la arquitectura podría ser aplicada a este trabajo. Esta solución se adapta bien a un entorno homogéneo en el que las conexiones tengan corta duración y la cantidad de datos transferida se relativamente pequeña, tales como ficheros HTML e imágenes

en un entorno de servidor HTTP.

Sin embargo, la solución propuesta no se adapta bien a entornos en los que las conexiones presentan una alta duración en el tiempo e importantes cantidades de datos transferidos, puesto que se requeriría el almacenamiento de todos y cada uno de los paquetes involucrados en la comunicación, lo cual supondría un consumo de memoria considerable, así como un tiempo de recuperación de fallo significativo. Además, la solución está íntimamente ligada a la arquitectura Primario-Respaldo, puesto que la réplica de respaldo y sus recursos no pueden ser aprovechados hasta que se produzca un fallo. Esto puede resultar una limitación si se requiren escenarios avanzados, tales como configuraciones de compartición y balanceo de carga entre las diferentes réplicas. Además, la solución propuesta es dependiente del protocolo de transporte empleado, concretamente TCP, aunque podría ser fácilmente a los nuevos protocolos de última generación, tales como SCTP, requiriendo un *logger* para cada protocolo. Para concluir, con respecto al modelo descrito en este trabajo (Sec. 2), para nuestros propósitos, únicamente el último estado alcanzado es significativo, por tanto, no es necesario mantener la evolución completa de los estados de la conexión tal y como realiza esta solución.

3.3.6. Técnicas de replicación en bases de datos

Generalmente, los algoritmos de replicación de bases de datos están construidos sobre primitivas síncronas de broadcast atómico. Los esquemas de replicación usados en entornos de bases de datos basados en protocolos de commit de dos y tres fases son complejos. Dichos protocolos tienen el objetivo de garantizar que las diferentes réplicas de una base de datos sean copias equivalentes. Sin embargo, tales protocolos, aplicados al modelo descrito en este trabajo, introducen un retardo significativo en las respuestas a los clientes, lo cual no se adapta a los objetivos específicos de maximización de la disponibilidad y el rendimiento del sistema que se plantean en este trabajo.

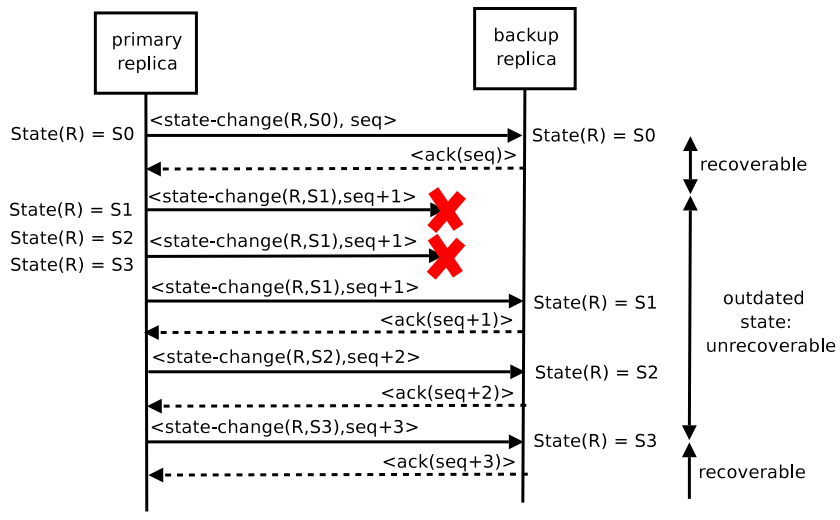

Figura 3.3: Reducción de la disponibilidad debido a la pérdida de un mensaje

Generalmente los protocolos de broadcast atómico imponen un reparto de los mensajes en orden total, en inglés *total order delivery*, con el objetivo de garantizar la linearizabilidad, en

inglés *linearizability*, el criterio que garantiza un comportamiento secuencia aceptable. En este trabajo se proponen protocolos de replicación que relajan dichas imposiciones con el objetivo de maximizar la disponibilidad del software con estados en situaciones en las que se produzcan pérdidas de mensajes (Fig. 3.3). Al contrario de lo que puede parecer, la pérdida de mensajes en un entorno del área local es posible bajo altas cargas debido a colisiones en la red y desbordamientos del buffer de transimisión [8]. Por desgracia, esto hace que en los periodos en los que el sistema esté sometido a cargas altas, la durabilidad de los estados se vea reducida, por tanto disminuyendo la disponibilidad global del sistema ante un posible fallo.

3.3.7. Replicación en entornos de Tiempo Real

Las aplicaciones de tiempo real deben dar respuesta a una serie de eventos síncronos y asíncronos dentro de un plazo de tiempo estricto. En este escenario, el tiempo invertido en la replicación no debe interferir en el cumplimiento de los plazos asignados a las tareas de tiempo real, es por ello que se deben limitar los recursos asignados a la replicación. Para ello, es necesario relajar el grado de consistencia entre las réplicas.

Una serie de autores formalizan el concepto de ventanas de inconsistencia con márgenes conocidos, consistencia temporal y replicación planificada de forma que garantizan el despliegue apropiado de las tareas de tiempo real junto a una maximización de la disponibilidad del sistema [23]. En esencia, las preocupaciones de estos trabajos son similares a las expuestas en éste: Dar una rápida respuesta a eventos y peticiones provenientes de los clientes y garantizar una rápida recuperación de fallos. Sin embargo, presentan dos diferencias fundamentales:

- Los entornos estudiados tienen un carácter fundamentalmente síncrono, lo cual garantiza la formalización de compensaciones entre las restricciones impuestas por la aplicaciones de tiempo real y las de replicación, frente al entorno estudiado en nuestro trabajo que es fundamentalmente asíncrono.

- Una vez superado el plazo de cumplimiento estricto de una tarea de tiempo real, dicha tarea carece de utilidad para el sistema, siendo por tanto abortada. Para los fines descritos en este trabajo, sería aceptable reducir ligeramente el rendimiento del sistema con el fin de garantizar una alta disponibilidad, por tanto, el entorno de tiempo real presenta más restricciones que el estudiado en este trabajo.

3.4. CORBA tolerante a fallos: FT-CORBA

Dentro del ámbito de CORBA se han presentado diferentes soluciones para implementar tolerancia a fallos. De entre ellas, cabe citar la extensión estándar propuesta por la OMG denominada *FT-CORBA*. En [25], se presenta una infraestructura basada en la intercepción de llamadas a rutinas, la idea fundamental consiste en la introducción de un *middleware* transparente a las aplicaciones de forma que no se requieren modificaciones en el sistema operativo o una recompilación de la aplicación. La infraestructura propuesta es escalable, además ofrece tolerancia a fallos, dependabilidad proactiva, adaptación de los recursos en base a fallos y rápida detección y recuperación de fallos. En dicho trabajo se discute también la dificultad de ofrecer una solución completa para aplicaciones de tiempo real *RT-CORBA* tolerantes a fallos.

3.5. Otras contribuciones a la alta disponibilidad

En [29] se exponen una serie de ideas preliminares que son base para la definición de la arquitectura orientada a eventos propuesta en este trabajo. Además se presenta la implementación de la librería *Stateful Networking Equipments* (SNE), tal librería extiende a las soluciones de alta disponibilidad existentes. Dicho trabajo hace un enfoque particular sobre firewalls y routers. De manera similar, una solución de tolerancia a fallos para VoIP que se implementa mediante un middleware orientado a eventos que recupera y salva los estados cuando se produce un fallo [17].

En [28] se elabora un trabajo preliminar que tiene como objetivo la definición de una arquitectura de replicación basada en la migración de los estados en caso de producirse un fallo corregible. Para ello se asume una infraestructura de detección de errores que permita diagnosticar errores corregibles y evaluar la posible migración a una réplica segura.

En [38] se introduce el concepto de durabilidad de estados, un concepto íntimamente ligado a la definición de disponibilidad. Se entiende por durabilidad de estados a la probabilidad de que un estado sobreviva a fallos. Con el objetivo de garantizar una alta durabilidad de estados, dicho trabajo propone el uso de un proxy de durabilidad, un software que se encarga de propagar los cambios de estados entre réplicas. Tal proxy intercepta los cambios de estado de un objeto mediante la *Java Reflexion API*.

Candea et al. proponen una interesante y original solución para mejorar la disponibilidad del software: los *microreboots* [9][10][11][12]. La idea consiste en aumentar la disponibilidad de un software mediante breves reinicios. La solución propuesta trata de buscar el momento óptimo en el cuál se debe realizar el reinicio del software. Este trabajo trata fundamentalmente de reducir el tiempo de indisponibilidad producido por fallos en el software. Los resultados expuestos en dicho trabajo demuestran que es posible aumentar la disponibilidad de un software mediante esta técnica.

Capítulo 4

Arquitectura de alta disponibilidad para software con estados

Con el fin de replicar los cambios de estados, y así mejorar la disponibilidad del servicio software con estados, proponemos una arquitectura basada en un *modelo dirigido a eventos*. La arquitectura descrita es modular y se presenta como una extensión a las soluciones existentes. De esta forma cualquier servicio software con estados que implemente la arquitectura descrita en este trabajo puede mejorar su disponibilidad de manera sencilla. Esta arquitectura mantiene en mente dos problématicas: *a)* las respuestas a los clientes deben ser lo más rápidas posibles, para de esta forma no degradar el servicio y así garantizar un alto rendimiento, y *b)* recuperación rápida de fallos.

El concepto clave de la arquitectura es el uso de un *proxy* de cambios de estados, esta idea fue propuesta en [38]. Dicho proxy se trata de un software dedicado que se encarga de garantizar la replicación de los estados entre las diferentes replicas. Concretamente, un proxy procesa los eventos de cambios de estado que tienen lugar en el servicio software con estados y mantiene una caché con el estado actual de las peticiones que están siendo procesadas (Fig.4.1). El proxy puede a su vez comunicarse con otros proxys a través de un enlace dedicado para evitar que los eventos de cambios de estados puedan ser empleados por una tercera parte con propósitos malintencionados, de esta forma se cierra la puerta a un posible problema de seguridad en la arquitectura propuesta. En este trabajo se asume que cada réplica que despliega un servicio software con estados dispone de un proceso proxy que garantiza que los estados son replicados apropiadamente.

Todo servicio software con estados que implemente la arquitectura propuesta debe implementar una serie de métodos que permitan al proxy procesar los eventos de cambios de estado que tengan lugar en dicho software. Concretamente son tres lo métodos a implementar:

- *dump()* se usa para obtener la tabla de estados completa, este método se usa para realizar una resincronización completa entre el proxy y el software con estados.
- *inject(array of states: states_t)* permite introducir en el software con estados la tabla de estados mantenida por el proxy. Este método es invocado durante el proceso de recuperación de un fallo, es decir, cuando la replica en la que se invoca el método ha sido seleccionada para reemplazar a una de las replicas activas que ha fallado.

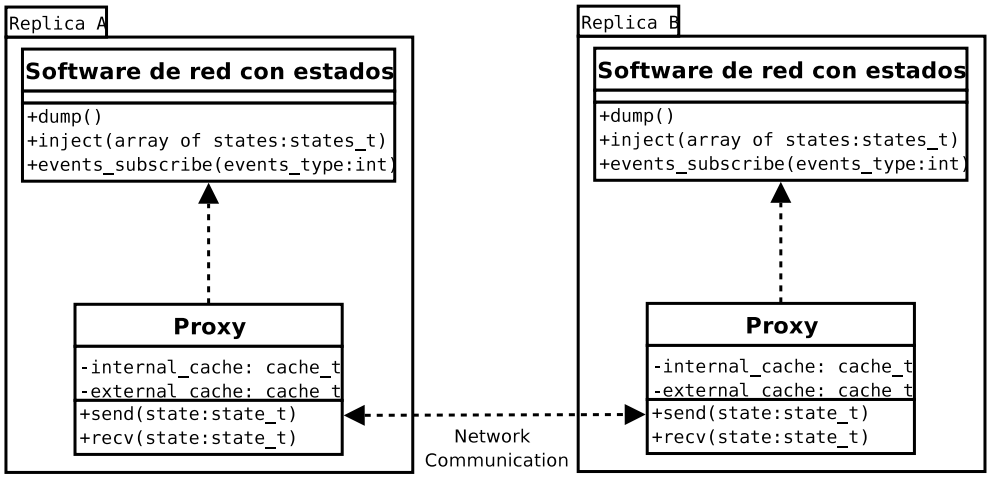

Figura 4.1: Diagrama UML de la arquitectura propuesta

- *events_subscribe(events_types: int)* ofrece un mecanismo al proxy para subscribirse a los eventos de cambio de estado que se producen en el software con estados.

Mediante dichos métodos, el proxy mantiene una caché de cambios de estados que se producen en el software denominada *caché interna*. El proxy, al comenzar su ejecución, invoca el método *dump()* con el fin de almacenar en dicha caché el estado actual de todas las peticiones activas, después procede a suscribirse al sistema de notificación de eventos mediante el método *events_subscribe(events_types: int)* para mantener actualizada la caché.

Para garantizar una alta durabilidad de los estados [38], dichos cambios de estado son propagados a través del enlace dedicado a otros proxy que mantendrán dichos estados en la *caché externa*, que es la caché empleada para mantener los estados de las peticiones que están manejando otras réplicas. Así, en caso de fallo de una de las réplicas activas, una nueva replica se selecciona para retomar las peticiones que estaba procesando la réplica en falla. Para la transmisión de mensajes se emplea un canal dedicado con el fin de asegurar que terceras partes no puedan obtener obtener la información de los estados que se propagan entre las diferentes réplicas, evitando así la posibilidad de posibles inyecciones de estados falsos o el robo de información crucial relacionada con la configuración desplegada.

En resumen, la arquitectura propuesta se compone de un proxy que mantiene al menos dos cachés, una interna, en la que se mantiene el estado de las peticiones activas en el software con estados, y otra externa, en la que se almacenan el estado de las peticiones manejadas por otras réplicas. De esta manera, la arquitectura propuesta ofrece la posibilidad de establecer configuraciones avanzadas de compartición de carga con relativa facilidad. El número de cachés externas dependerá del número de replicas para las que se actúa como respaldo, es decir, si se pretende garantizar que un cierto proxy pueda recuperar las peticiones manejadas por N réplicas en fallo, se deberán mantener N cachés externas.

Para comprobar que el proxy funciona apropiadamente se puede emplear software de vigilancia y monitorización, también conocido como *watchdogs*. Así, en caso de que se produzca un fallo en el proxy, se puede proceder simplemente a relanzarlo o alternativamente a seleccio-

nar una versión diferente que no sufra el problema observado, solucionando posibles problemas en la implementación del propio proxy mediante técnicas de diversificación basadas en el N-versionado [36].

El mecanismo de subscripción permite al proxy recibir eventos de cambio de estados está basado en un sistema de mensajería, por tanto, los eventos vienen encapsulados en mensajes. Dicho mensajes pueden contener la información asociada al evento en formato XML o alternativamente en algún otro formato flexible y extensible que reduzca sustancialmente el tamaño de los mensajes y el tiempo requerido para el *parsing* tales como Netlink [31]. Los mensajes recibidos son procesados por el proxy para mantener la caché interna actualizada, además de ser enviados a otros proxy con el fin de mantener la caché externa al día.

Así, una cierta petición en una réplica activa puede producir un cambio de estado en el software, tal cambio de estado será notificado al proxy mediante un mensaje que será empleado para actualizar la caché interna, y a su vez, enviado a otras réplicas que actúen como respaldo. En caso de fallo de la réplica activa, una réplica de respaldo se seleccionará para retomar las peticiones que están aún pendientes de ser servidas, para ello la réplica candidata invocará el método *inject()*, de esta forma se introducirán los estados mantenidos en la caché externa en el software, garantizando así que no se produce ninguna interrupción en el servicio desplegado.

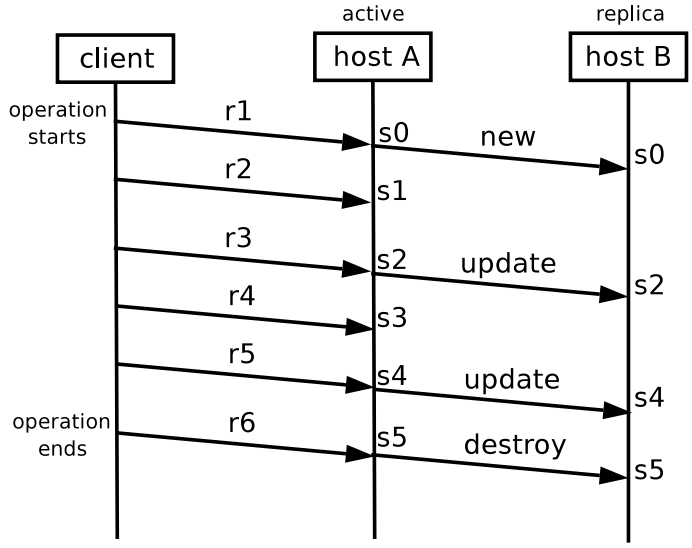

Figura 4.2: Propagación de cambios de estado en la red

Con el fin de no reinventar la rueda, para resolver el problema de la selección de la réplica candidata a reemplazar una que falla se emplean técnicas de alta disponibilidad ya existentes, es por ello que el problema de la selección de una réplica segura no es tratado en este trabajo.

Para concluir, la arquitectura propuesta mejora la disponibilidad del software con estados mediante pequeñas modificaciones, concretamente la implementación de los tres métodos descritos anteriormente. La solución descrita es modular y permite reutilizar el trabajo realizado por los investigadores y la industria en las últimas décadas. Además, los conceptos de cachés interno y externo permiten el despliegue de configuraciones en compartición de carga de trabajo.

Capítulo 5

Soluciones de replicación propuestas

5.1. Replicación basada en secuencias incrementales y recuperación de pérdidas

Una de las propuestas de este trabajo se trata de un protocolo de replicación confiable para software con estados con las características descritas en la sección 2. Dicho protocolo de replicación se basa en números de secuencia incrementales y control explícito de pérdida de mensajes. El objetivo principal de este protocolo de replicación consiste en reducir al máximo el tiempo requerido para recuperar un cambio de estado contenido en un mensaje perdido. Del lado del emisor, cada petición se representa mediante un objeto que almacena, además de los atributos asociados a la petición manejada, el último número de secuencia de un mensaje que contenía un cambio de estado de dicho objeto.

Para el envío de mensajes se emplea el método `broadcast()`, dicho método anota el número de secuencia empleado e inserta el objeto en una cola. En caso de que el objeto ya existiera en la cola, el objeto es borrado de la cola y reinsertado de nuevo. Con el fin de asegurar que los mensajes han sido correctamente recibidos se emplean mecanismos de retransmisión explícitas, también conocidos por *negative acknowledgements*, y confirmaciones explícitas de correcta recepción, también conocidos por *positive acknowledgements*. En caso de recibir una petición de retransmisión explícita, únicamente se envía un mensaje con el último cambio de estado que ha sufrido el objeto que representa la petición. La retransmisión de todos los cambios de estados no es necesaria puesto que, en base al modelo definido en este trabajo, únicamente se debe garantizar una alta durabilidad del último estado alcanzado.

```
sender:
  initialization:
    exp := 1
    seqnum := 1
    queue := {}

  procedure broadcast(m, obj)
    obj.sn := seqnum
    seqnum := seqnum + 1
    if obj belongs to queue:
```

```
      remove(obj, queue)
      add(obj, queue)
    else
      add(obj, queue)
    send(m, obj.sn) to all

  event_handler:
    on state change for obj:
      m := build_message(obj)
      broadcast(m, obj)

  retransmission:
    while receive_retransmission(m):
      if is_nack(m):
        for each obj in queue:
          if obj.sn >= m.from and
             obj.sn <= m.to:
            broadcast(m, obj)
      if is_ack(m):
        for each obj in queue:
          if obj.sn >= m.from and
             obj.sn <= m.to:
            remove(obj, queue)
```

Del lado del receptor, siempre se procesan todos los mensajes recibidos, incluidos aquellos que están fuera de secuencia. En caso de que algunos mensajes se hubieran perdido, una petición explícita de reenvío con el rango de mensajes perdidos sería enviada, es decir, un mensaje de *negative acknowledgement* indicando desde qué secuencia a hasta qué secuencia b no han sido recibidas. Observe que el número que el número de mensajes n reenviado por el emisor siempre cumple que $n \leq b - a$ ya que aquellos mensajes que contenían cambios de estados $s_{k-1}, s_{k-2}, s_{k-3}, ...s_{k_n}$ anteriores a un cambio de estado s_k no son reenviados.

```
destinations:
  initialization:
    exp := 1
    seqnum := 1
    queue := {}

  procedure send_nack(seqrcv, exp)
    m.from := exp
    m.to := seqrcv - 1
    add(m, queue)
    send(m, seqnum)

  procedura send_ack(from, to)
    m.from := from
    m.to := to
```

```
    add(m, queue)
    send(m, seqnum)

  while receive(m, seqrcv):
    if seqnum <> exp:
       window := WINDOW_SIZE
       send_nack(seqrcv, exp) to all
    if --window = 0:
       window := WINDOW_SIZE
       from := seqrcv - WINDOW_SIZE
       to := seqrcv
       send_ack(from, to) to all
    deliver(m)
```

En la figura 5.1 se representa una posible pérdida de mensajes. Concretamente, los mensajes con número de secuencia `seq+1` y `seq+2` son perdidos. El mensaje `seq+3` es correctamente recibido, no obstante, la pérdida de los mensajes resulta en una petición de retransmisión explícita desde `seq+1` hasta `seq+2`. El emisor procesa dicha petición de forma que un único mensaje que contiene el último cambio de estado alcanzado por la petición `R1` es enviado, es decir, el mensaje `seq+1` contenía un cambio de estado S_1 que precedía al mensaje `seq+2` que contenía el estado S_2 de la petición `R1`, es por ello que únicamente es reenviado el mensaje `seq+2` pero con un nuevo número de secuencia `seq+4`. De esta forma se garantiza una rápida recuperación de estados ante posibles pérdidas de mensajes. Obsérvese que el número de secuencia siempre es incremental, es decir, aquellos mensajes que contienen estados que han sido reemplazados por nuevos cambios de estado no son reenviados.

El algoritmo propuesto reduce el número de mensajes retransmitidos y aumenta la durabilidad de los estados en escenarios de pérdida de mensajes. De esta forma las réplicas que actúan como respaldo alcanzan un estado consistente antes.

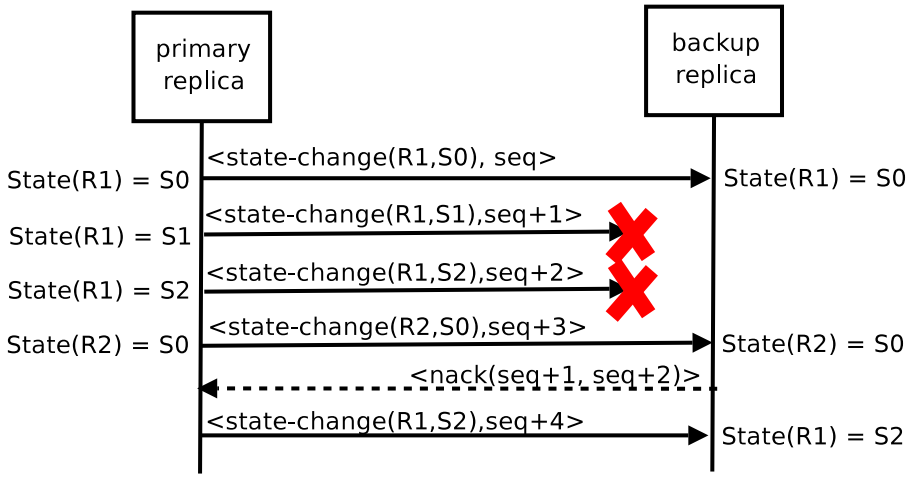

Figura 5.1: Recuperación explícita de pérdida de estados

El protocolo propuesto emplea mecanismos de confirmación explícita de correcta recepción de mensajes con el objetivo de reducir el tamaño de la cola de objetos transmitidos, y así reducir

el impacto que supone la retransmisión de los objetos cuyos mensajes han sido perdidos, ya que en el tiempo supone un procesamiento de $O(n)$. El tamaño de la ventana de *acknowledgements* es un parámetro configurable que debe ser afinado con el objetivo de establecer una compensación entre el excesivo envío mensajes de confirmación de correcta recepción y un tamaño de cola de reenvío de tamaño demasiado grande.

5.2. Replicación probabilística

El protocolo de replicación probabilística no confiable que se describe en esta sección es sencillo de implementar y ofrece buenos resultados en práctica. El objetivo de esta solución es el de asegurar una alta durabilidad de los estados asociados a peticiones de larga duración, es decir, aquellas que requieren más tiempo para terminar, por tanto, aquellas en las que los recursos invertidos son elevados. La solución se basa en un protocolo de replicación no confiable, es decir, no hay certeza de qué estados mantienen las réplicas. La idea principal consiste en el envío periódico de un mensaje que contiene el estado de una cierta petición, independientemente de si dicha petición ha cambiado de estado o no.

```
sender:
  initialization:
    exp := 1
    seqnum := 1
    queue := {}

  procedure broadcast(m, obj)
    seqnum := seqnum + 1
    send(m, seqnum) to all

  procedure set_timer(obj)
    obj.timeout := factor(obj.times)

  event_handler:
    on state change for obj:
      m := build_message(obj)
      obj.times := 1
      broadcast(m, obj)
      if is_timer_set(obj):
        del_timer(obj)
      set_timer(obj)
      add_timer(obj)

  timer_handler:
    on timeout for obj:
      m := build_message(obj)
      obj.times := obj.times + 1
      broadcast(m, obj)
      if is_timer_set(obj):
        del_timer(obj)
      set_timer(obj)
      add_timer(obj)
```

Obsérvese que cada objeto dispone de un atributo *times* que indica cuántas veces se ha enviado dicho estado. Cuando se produce un cambio de estado tal contador es puesto a uno. El envío de mensajes sin esperar confirmación de la correcta recepción de dicho mensaje. Con

	1	2	3	4	5	6
0.01	10e-2	10e-4	10e-6	10e-8	10e-10	10e-12
0.05	5e-2	2,5e-3	1,2e-4	6,2e-6	3,1e-7	1,5e-8
0.10	10e-1	10e-2	10e-3	10e-4	10e-5	10e-6
0.15	1,5e-1	2,5e-2	3,3e-3	6,2e-4	3,1e-5	1,5e-6
0.20	2e-1	4e-2	8e-3	16e-4	32e-5	64e-6

Cuadro 5.1: Probabilidad de N pérdidas consecutivas de un cierto estado

el fin de reducir el impacto de la posible pérdida de mensajes, cada cierto tiempo se envía un mensaje que contiene el estado de una petición. La frecuencia de envío viene determinada por la función *factor()*, dicha función determina el tiempo asociado al temporizador que indica cuándo se volverá enviar un mensaje conteniendo dicho estado.

La función *factor()* debe optimizar el número de mensajes envíados frente a garantizar una alta durabilidad de los estados de aquellas peticiones de larga duración. Obsérvese que la probabilidad de que una réplica mantenga un estado s_{k-1} cuando debería mantener un estado s_k por culpa de una pérdida viene determinado por la siguiente fórmula:

$$p(t) = \left(\frac{\text{lost messages}}{\text{sent messages}}\right)^{\lfloor t/n \rfloor}$$

En la tabla 5.1 hemos representado de forma teórica la probabilidad de que un cierto mensaje que contiene un cambio de estado s_k sea perdido hasta seis veces consecutivas, las columnas reflejan la probabilidad de pérdida de mensajes, entre 0.01 y 0.20, frente al número de envíos realizados. Los resultados muestran que para una probabilidad de pérdida del 5 %, con un total de seis envíos del mismo estado, la probabilidad de pérdida de un mensaje es de $1,5625e-8$, lo cual es un evento de baja probabilidad.

5.3. Replicación basada en la detección de errores

En la actualidad, las plataformas *off-the-shelf* comienzan a disponer de mecanismos de detección y corrección de errores, como es el caso de la memoria RAM ECC y la detección de errores en las transferencias mediante mecanismos de paridad en el bus PCI. Muchos de estos errores, siempre que no sean críticos, son corregidos sin que el administrador del sistema tenga constancia de ellos. En esta tesitura, dichos errores corregibles son un indicio de una probable degradación progresiva del sistema que posiblemente con el tiempo derive en una falla completa, ya sea de *crash* o de corrupción de datos. De esta forma, podemos catalogar los fallos que tienen lugar en el hardware en dos tipos: *a)* corregibles, aquellos que son susceptibles de ser subsanados, *b)* incorregibles, aquellos que producen una falla completa del sistema.

Existen proyectos que empiezan a emplear dichas nuevas funcionalidades con el fin de mejorar la disponibilidad de los sistemas en producción, tal es el caso del proyecto EDAC [1] en el sistema operativo GNU/Linux. Dicho proyecto ofrece una interfaz accesible mediante la cual tanto el administrador del sistema como cualquier tarea automatizada de monitorización disponen de información acerca del estado del sistema, pudiendo tomar la decisión de migrar, de manera preventiva, el servicio desplegado por un software con estados a una réplica segura en caso de que se estén produciendo errores corregibles.

La solución propuesta en este apartado consiste en la implementación de un software monitor que realice tareas de vigilancia y diagnosis a partir de la información de estado del sistema. Dicho software se encargará de tomar la decisión de migrar el servicio desplegado por el software con estados a una réplica segura tan pronto como surga una situación comprometida. Dicha solución se basa en el siguiente supuesto: Sea F el tiempo requerido por el monitor para decidir si la situación es comprometida o no junto al tiempo requerido para migrar el servicio a otra réplica segura. Sea T el tiempo entre el primer fallo corregible y uno incorregible. Entonces F debe cumplir en todo momento que $T > F$ con el fin de garantizar que el proceso de migración se podrá realizar de forma apropiada (Fig 5.2).

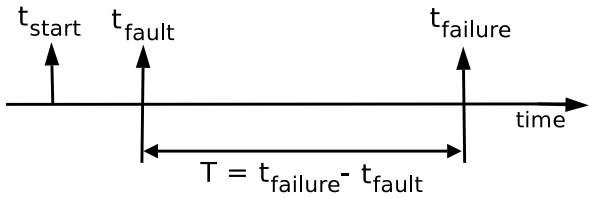

Figura 5.2: Time between Fault and Failure

En caso de que este supuesto no sea cierto, el software de monitorización no podrá asegurar que la migración del servicio de una réplica que padece fallos correglibles a una segura se realice con éxito. Es por ello que el software de monitorización deberá tomar decisiones en el menor tiempo posible, empleando para ello métodos sencillos pero a la vez potentes. Además, se debe garantizar que la infraestructura de detección de fallos es completa, es decir, es capaz de detectar todos aquellos posibles fallos que se produzcan en el sistema.

La principal ventaja de esta solución se fundamenta en el reducido coste de despliegue, puesto que el servicio será migrado cuando el software de monitorización lo crea oportuno, eg. cuando se produzca un fallo corregible, reduciendo así el consumo de recursos dedicados a la replicación.

Capítulo 6

Optimizaciones a la replicación

En este capítulo se proponen dos optimizaciones a la replicación que tienen como objetivo la reducción de los recursos que se invierten para garantizar una alta disponibilidad. Estas optimizaciones están extraídas del modelo formal discutido en la sección 2.

6.1. Garantía de alta durabilidad de estados útiles

El objetivo de esta optimización es el de dar prioridad a la replicación de aquellos estados que mejoran la disponibilidad del servicio desplegado por el software con estados. Para ello realizamos la siguiente clasificación de tipos de mensajes:

- *New*: Una réplica ha recibido una nueva petición, por tanto este mensaje contiene el estado inicial S_0 asociado a dicha petición.
- *Update*: El estado de una cierta petición ha cambiado, este mensaje contiene un cambio de estado desde $S_k \rightarrow S_k + 1$,
- *Destroy*: La petición ha concluido, i.e. el último estado S_n ha sido alcanzado.

La evolución de los mensajes se puede reprensentar mediante la figura 6.1.

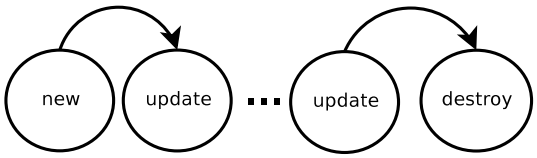

Figura 6.1: Secuencia de Mensajes de Replicación

De dicha clasificación se deduce que la replicación de los mensajes de tipo *Destroy* pueden ser pospuestos indefinidamente puesto que dichos estados no mejoran la disponibilidad del servicio puesto que hablan de una petición que ya no existe, en realidad cumple una función sincronizadora, es decir, mantienen a las réplicas de repaldo sincronizadas con las réplicas activas. Por supuesto, esto implica que las réplicas mantendrán objetos obsoletos, es decir, objetos que representan peticiones que ya han concluido. Esto supone un consumo extra en memoria y hace que las réplicas de respaldo no sean copias equivalentes de la réplica activas, sin embargo,

esta optimización aumenta la durabilidad de estados útiles para recuperar el servicio desplegado por el software con estados en caso de fallo. Así, se pueden introducir diferentes algoritmos de planificación que decidan en qué momento emitir los mensajes de tipo *Destroy* frente a los mensajes de *New* y *Update* de forma que se mejore la disponibilidad del servicio.

Con el fin de reducir el efecto polución que podrían padecer las réplicas de repaldo debido al mantenimiento de objetos que representan peticiones obsoletas, se plantean dos soluciones:

a) Un estado final S_k es mantenido en una réplica de repaldo un tiempo máximo igual al tiempo máximo de vida T_k de dicho estado, tal y como se explicita en el modelo del sistema en la capítulo 2. Una vez alcanzado dicho tiempo se libera la memoria asignada a dicho objeto.

b) En caso de que una réplica de respaldo reciba un estado S_0 asociado a una petición cuyas variables $v_1, v_2, ..., v_j$ identificativas sean iguales a las de un objeto existente, se libera el objeto existente para crear uno nuevo en estado S_0. Las variable identificativas son aquellos descriptores que permiten establecer una identificación unívoca de cierta petición.

6.2. Optimización basada en técnicas de aprendizaje supervisado: Naive Bayes

Con el objetivo de reducir el número de estados que han de ser transmitidos a otras réplicas que actúan de respaldo, y así reducir el consumo de recursos que supone la propia replicación, en esta sección proponemos una optimización basada en técnicas de aprendizaje. Para ello, en primer lugar vamos a clasificar los estados en dos tipos: transicionales y estables. Esta clasificación nos permite conocer si el esfuerzo requerido para replicar un estado es rentable o no. Los estados transicionales son aquellos cuya probabilidad de ser reemplazados por otro cambio de estado en un corto periodo de tiempo es alta, en otras palabras, los estados transicionales suelen tener una vida límitada. Por otro lado, los estados estables son aquellos estados en los que una petición tiende a permanecer un tiempo considerable.

En este trabajo hemos formalizado dicha clasificación de estados en función de la probabilidad (P) de un evento de cambio de estado (X). Sea T_m el tiempo máximo de vida de un cierto estado después del cual la petición expira, sea t la unidad de tiempo actual, la probabilidad de que un estado transicional sea reemplazado por otro estado puede ser expresada como la siguiente función:

$$P_x(t) = \begin{cases} 1 - \delta(t/T_m) & if \quad (0 \leq t \leq T_m < 1) \\ 0 & if \quad (t > T_m) \end{cases}$$

A su vez, la probabilidad de que un estado estable pueda ser reemplazo por otro estado se expresa en la siguiente función:

$$P_x(t) = \begin{cases} \omega(t/T_m) & if \quad (0 \leq t \leq T_m < 1) \\ 1 & if \quad (t > T_m) \end{cases}$$

Esta formalización es una representación de la probabilidad de que un cambio de estado sea reemplazo por otro estado con respecto al tiempo. Ambas definiciones dependen de las funciones $\delta(x)$ y $\omega(x)$ que determinan cómo crece o decrece la probabilidad con respecto al tiempo, eg. linealmente, exponencialmente.

Puesto que los estados transicionales son candidatos a ser reemplazados por otros estados en un corto espacio de tiempo, existe la posibilidad de que el tiempo requerido para propagar los cambios de estados sea mayor al tiempo de vida del estado. En ese caso, los recursos invertidos en la replicación no son rentabilizados. Por tanto, podemos deducir que la replicación de los estados transicionales puede supone un esfuerzo en recursos que probablemente no sólo no mejora la disponibilidad del servicio sino que además supone un consumo de recursos injustificado. Esta clasificación de estados abre la puerta al uso de técnicas de aprendizaje supervisado. Concretamente, proponemos el uso de un clasificador Bayesiano con el fin de obtener un indicador que justifique si la replicación del estado mejora la disponibilidad del servicio o no.

Los clasificadores bayesianos ofrecen una manera sencilla y a la vez potente para llevar a cabo la tarea de la clasificación. Sea la variable C que representa la clase, sea X_i una variable que representa una de las características. Dada una instancia específica x, es decir, un conjunto de valores $x_1, x_2, ...x_n$ de variables características. Un clasificador bayesiano nos permite calcular la probabilidad $P(C = c_k|X = x)$ para cada posible clase c_k, concretamente:

$$p(C = c_k|X = x) = \frac{\mathrm{P}(X=x|C=c_k)\mathrm{P}(C=c_k)}{\mathrm{P}(X=x)}$$

En dicha ecuación es $P(X = x|C = c_k)$ la parte que resulta más difícil de calcular. Concretamente, Naive Bayes [16] reduce el problema de la forma más restrictiva, ya que asume que toda variable x_i es condicionalmente independiente de cualquier otra característica x_j dada una clase C. De esta forma concluye que:

$$p(X = x|C = c_k) = \prod_i P(X = x_i|C = c_k)$$

Aplicando dicha teoría al problema planteado en este trabajo, definimos el dominio de la clase C en los dos posibles estados que resultan de la clasificación realizada anteriormente, es decir, en estados transicionales y estables. El conjunto de características es el dominio de posible estados por los que puede pasar el software con estados, es decir, $s_1, s_2, ...s_n$. De esta forma, la probabilidad de que un cierto estado s_k sea transicional, y por tanto no deba ser replicado es:

$$P(C = transicional|X = s_k) = \frac{\mathrm{P}(X=s_k|C=transicional)\mathrm{P}(C=transicional)}{\mathrm{P}(X=s_k)}$$

De igual manera, la probabilidad de que un estado s_k sea estable, y por tanto deba ser replicado es:

$$P(C = estable|X = s_k) = \frac{\mathrm{P}(X=s_k|C=estable)\mathrm{P}(C=estable)}{\mathrm{P}(X=s_k)}$$

De forma intuitiva se deduce que:

$$P(C = estable|X = s_k) + P(C = transicional|X = s_k) = 1$$

Con el fin de establecer los valores iniciales necesarios para poder resolver las ecuaciones planteadas, se establece un periodo inicial de entrenamiento, en el cuál se recopila información acerca del comportamiento del software con estados. En la práctica, y con el fin de simplificar la clasificación de estados posibles, se emplea un umbral que determina si un estado se comporta como transicional o estable. Dicho umbral puede fijarse mediante la variable u que es igual al doble del tiempo requerido para propagar un cambio de estado. Por tanto, si la vida de un cierto estado s_k es inferior a u consideramos que el estado es transicional, de lo contrario es estable.

Dado que existen métodos de aprendizaje supervisado que permiten clasificar datos basados en supuestos menos restrictivos como los impuestos por Naive Bayes, debe considerarse que la solución propuesta en este trabajo ofrece un planteamiento inicial del problema, es por ello que, debe servir de base para trabajos futuros.

6.2.1. Ejemplo

En base al ejemplo descrito en la sección 2.2, considérese un sistema de firewall con estados que realiza filtrado de tráfico en el que las transferencias y las latencias de la red son homogéneas y uniformes. Por tanto, mediante aprendizaje supervisado se observa que:

$$P(transicional|syn\_sent) = 0{,}199$$

$$P(transicional|syn\_recv) = 0{,}199$$

$$P(transicional|established) = 0{,}005$$

$$P(transicional|fin) = 0{,}199$$

$$P(transicional|fin\_wait) = 0{,}199$$

$$P(transicional|closed) = 0{,}199$$

Además, para cada estado se cumple que:

$$P(syn\_sent) = P(syn\_recv) = P(establish) = P(fin) = P(fin\_wait) = P(closed) = \frac{1}{6}$$

Siendo la probabilidad de estados transicionales y estables:

$$P(transicionales) = 0{,}995$$

$$P(estables) = 0{,}005$$

Se calcula que la probabilidad de que un estado syn_sent sea transicional es:

$$P(C = transicional|X = syn\_sent) = \frac{\mathrm{P}(X=syn\_sent|C=transicional)\mathrm{P}(C=transicional)}{\mathrm{P}(X=syn\_sent)}$$

$$P(C = transicional|X = syn\_sent) = \frac{0{,}199*0{,}833}{0{,}1667} = 0{,}9944$$

Con el fin de hacer una predicción más precisa en este ejemplo, se podrían emplear otras variables características del dominio, tales como los descriptores clásicos empleados en firewalls: IP fuente, IP destino, ...

Capítulo 7

Evaluación preliminar

Con el objetivo de evaluar la arquitectura propuesta en este trabajo, hemos implementado un demonio que actúa de proxy de replicación para firewalls con estados llamado conntrackd [26], dicho software está disponible con licencia libre. El sistema operativo *GNU/Linux* ha sido elegido como campo de experimentación por el acceso gratuito al código fuente así como por su extendido uso en la industria de la seguridad IT.

Este demonio proxy replica los estados entre las diferentes réplicas de firewall, el proxy usa un sistema de mensajería extensible, eficiente y estandarizado denominado Netlink [31] que permite insertar, eliminar objetos y obtener la tabla de estados del *connection tracking system* [27], el subsistema que se encarga de mantener el estado asociado al tráfico procesado por el firewall, así como suscribirse a eventos de cambio de estado.

El entorno de pruebas está compuesto por sistemas AMD Opteron dual core 2.2GHz conectadas a una red de 1 Gbps. El esquema de red está compuesto por cuatro sistemas, los sistemas A y B actúan como estaciones de trabajo y dos replicas FW1 y FW2 que actúan como firewalls (Fig. 7.1). Concretamente, FW1 actúa como réplica activa y FW2 como réplica de respaldo

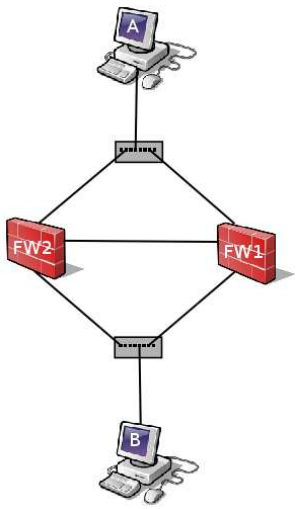

Figura 7.1: Entorno de pruebas

Concretamente, el sistema A actúa como cliente HTTP que genera peticiones de ficheros HTML de un tamaño de 4 KBytes de un servidor web instalado en el sistema B. Para la evaluación preliminar, hemos generado hasta 2500 peticiones HTTP por segundo con el objetivo de medir el consumo de CPU con y sin el proxy de replicación. El algoritmo de replicación es el descrito en la sección 5.1. El consumo de CPU ha quedado reflejado en una gráfica (Fig. 7.2), la herramienta *cyclesoak* [24] ha sido empleada para obtener mediciones precisas del consumo de CPU. Por otro lado, vale la pena mencionar que no se ha observado ningún impacto en el rendimiento del sistema en términos de reducción del número de peticiones HTTP manejadas. Por tanto, podemos concluir que la arquitectura propuesta requiere potencia extra de cálculo para mejorar la durabilidad de estados en un firewall con estados pero sin que por ello afecte al rendimiento del sistema.

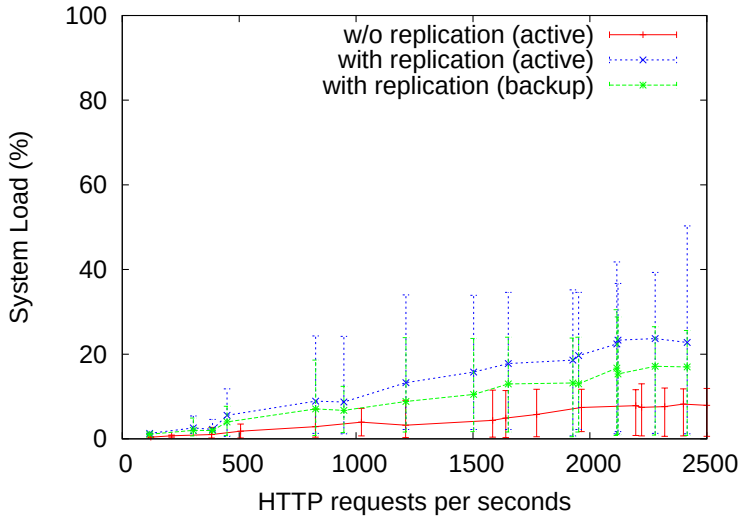

Figura 7.2: Carga del sistema

A continuación, hemos aplicado la optimización descrita en la sección 6.2 con el objetivo de reducir el número de mensajes de replicación transmitidos a la réplica de respaldo. Los resultados muestran (Fig. 7.3) que el consumo de CPU se reduce de manera significativa, particularmente del lado de la réplica de respaldo. Esto permite el despliegue de escenarios de compartición de carga en el que ambas réplicas actúen simultáneamente como activa y respaldo la una de la otra. Sin embargo, resulta significativo que la optimización no reduce sustancialmente el consumo de CPU en la réplica activa, esto se debe a que el proxy demonio debe procesar todos los eventos de cambio de estado incluso si estos deciden no ser replicados.

Con el objetivo de evaluar el tiempo requerido durante una recuperación de fallos, hemos medido el tiempo requerido para inyectar objetos que representan conexiones desde el caché del proxy hasta el connection tracking system. Los resultados (Fig. 7.4) muestran que el tiempo requerido para inyectar 20000 objetos es cercano a 180 ms, una penalización razonable. Otra conclusión obvia que se puede extraer de la gráfica es que el tiempo requerido para inyectar objetos aumenta linealmente.

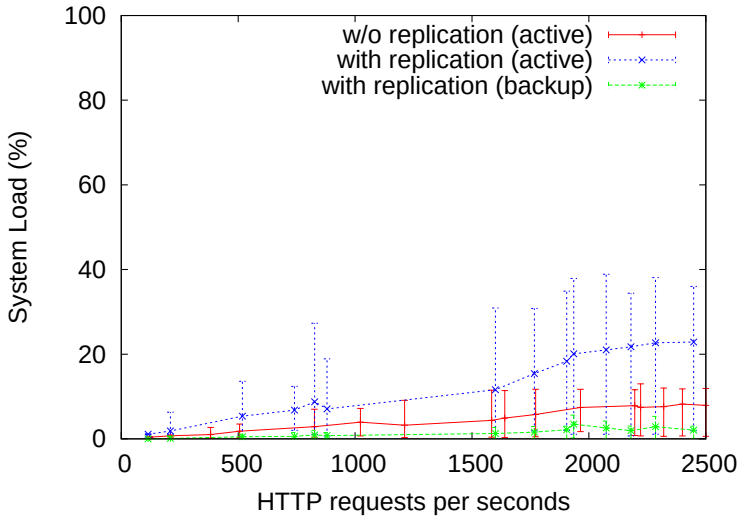

Figura 7.3: Carga del sistema con optimizaciones

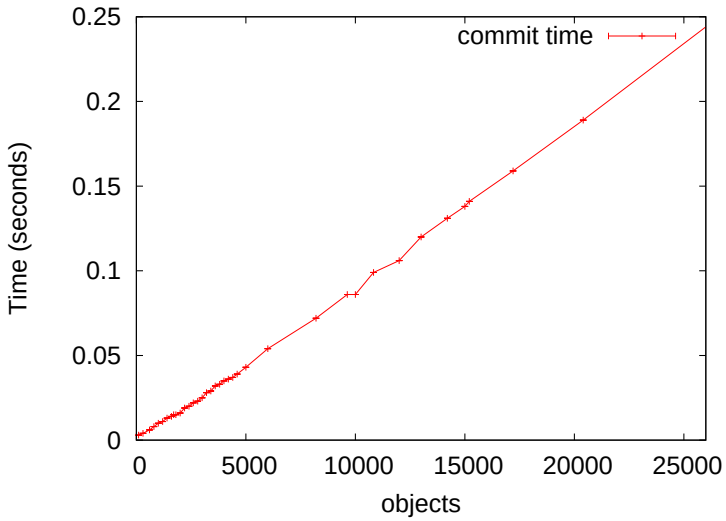

Figura 7.4: Tiempo requerido para añadir objetos

Capítulo 8

Conclusiones y Trabajos futuros

En este trabajo hemos propuesto una arquitectura completa para software de red de alto rendimiento con estados, un serie de protocolos de replicación, así como optimizaciones que reduzcan la cantidad de estados replicados, para de esta forma, disminuir los recursos dedicados a la replicación, principalmente el consumo de CPU.

La solución propuesta propuesta mantiene en mente rendimiento y disponibilidad como principales preocupaciones: se garantiza que las respuestas a clientes son rápidas y los tiempos de recuperación desde fallo son bajos.

Las optimizaciones están extraídas de la semántica del software con estados descrito en el modelo formalizado (Sec. 2). La primera de ellas se basa en la observación de que los estados finales S_n no mejoran la disponibilidad, por tanto su replicación puede ser pospuesta indefinidamente. La segunda consiste en el uso de técnicas de aprendizaje supervisado con el fin de reducir el número de estados replicados, garantizando que aquellos estados en los que una petición permanece prolongadamente son estados candidatos a ser replicados. Descartando la replicación de aquellos estados que no aumentan la durabilidad de manera sustancial.

Actualmente estamos trabajando para evaluar las optimizaciones propuestas, así como para mejorar la arquitectura y los protocolos de replicación detallados en este trabajo. Queda pendiente el estudio a fondo de una solución de replicación basada en la detección y diagnosis de errores que permita la migración de los estados cuando se produzca un fallo, evitando de esta forma la penalización inherentemente asociada a la replicación [28]. De cara a la evaluación, planeamos completar la evaluación de todos los protocolos propuestos así como implementar escenarios avanzados de balanceo y compartición de carga entre las diferentes replicas.

Bibliografía

[1] Edac (error detection and correction) project. http://bluesmoke.sourceforge.net.

[2] Keepalived: Healthchecking for lvs and high availability. http://keepalived.sourceforge.net/.

[3] Openbsd project. http://www.openbsd.org/index.html.

[4] Virtual router redundancy protocol (vrrp) charter. http://www.ietf.org/html.charters/vrrp-charter.html.

[5] Y. Amir and C. Tutu. From total order to database replication. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 494, Washington, DC, USA, 2002. IEEE Computer Society.

[6] A. Avizienis. Toward systematic design of fault-tolerant systems. In *IEEE Computer, Vol. 30, No. 4, pag.51-58*, apr 1997.

[7] N. Budhijara, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In *Distributed Systems, ACM Press, pp.199-216*, New York, USA, 1993.

[8] N. Budhiraja and K. Marzullo. Tradeoffs in implementing primary-backup protocols. In *Proceedings of 7th IEEE Symposium on Parallel and Distributeed Processing*, page 280, San Antonio, Texas, USA, October 1995.

[9] G. Candea, J. Cutler, and A. Fox. Improving availability with recursive microreboots: A soft-state system case study. In *Performance Evaluation Journal, vol 56*, mar 2004.

[10] G. Candea and A. Fox. Reboot-based high availability. In *OSDI Work-in-Progress Abstract*, San Diego, CA, USA, oct 2000.

[11] G. Candea and A. Fox. Designing for high availability and mesurability. In *1st Workshop on Evaluating and Architecting System Dependability*, Goteborg, Sweden, jul 2001.

[12] G. Candea and A. Fox. Crash-only software. In *9th Workshop on Hot Topics in Operating Systems*, may 2003.

[13] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. In *ACM journal*, pages 225–267, mar 1996.

[14] T. Chou. Toward an active network architecture. In *ACM SIGCOMM Computer Communication Review, vol.26, issue 2, pag.5-17*, 1996.

[15] T. Chou. Beyond fault tolerance. In *IEEE Computer Vol. 30, No. 4, pag.31-36*, 1997.

[16] I. Good. The estimation of probabilities: An essay on modern bayesian methods. *MIT press*, 1965.

[17] A. Gorti. A faul tolerant voip implementation based on open standards. In *IEEE Proceedings EDCC-6, pag. 35-38, Coimbra, Portugal*, oct 2006.

[18] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. In *IEEE Computer, Vol. 30, No. 4, pag.68-74*, apr 1997.

[19] R. Hinden. Rfc 3768: Virtual router redundancy protocol (vrrp), apr 2004.

[20] T. Li, B. Cole, P. Morton, and D. Li. Rfc 2281: Cisco hot standby router protocol, mar 1998.

[21] High-availability linux project. http://linux-ha.org/.

[22] M. Marwah, S. Mishra, and C. Fetzer. Tcp server fault tolerance using connection migration to a backup server. *In Proc. IEEE Intl. Conf. on Dependable Systems and Networks (DSN), pages 373–382, San Francisco, California, USA, June 2003.*, 2003.

[23] A. Mehra, J. Rexford, and F. Jahanian. Design and evaluation of window-consistent replication service. In *IEEE Transaction on Computers*, volume 46, pages 986–996, Sept 1997.

[24] A. Morton. cyclesoack: a tool to accurately measure cpu consumption on linux systems. http://www.zip.com.au/ akpm/linux/.

[25] P.Ñarasimhan, T. A. Dumitras, A. M. Paulos, S. M. Pertet, C. F. Reverte, J. G. Slember, and D. Srivastava. Mead: support for real-time fault-tolerant corba: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(12):1527–1545, 2005.

[26] P.Ñeira. conntrackd: The netfilter's connection tracking userspace daemon. http://people.netfilter.org/pablo/.

[27] P.Ñeira. Netfilter's connection tracking system. *In :LOGIN;, The USENIX magazine, Vol. 32, No. 3, pages 34-39*, 2006.

[28] P.Ñeira. Stop failures: Software-based approaches for higly available stateful services. In *IEEE Proceedings Supplemental EDCC-6, pag. 41-42, Coimbra, Portugal*, oct 2006.

[29] P.Ñeira, L. Lefevre, and R. M. Gasca. High availability support for the design of stateful networking equipments. In *IEEE proceeding ARES'06: The First International Conference on Availability, Reliability and Security, Vienna, Austria*, apr 2006.

[30] H. Roelle. A hot-failover state machine for gateway services and its application to a linux firewall. In *Management Technologies for E-Commerce and E-Business Applications, 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM 2002*, Montreal, Canada, oct 2002.

[31] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. Rfc 3549 - linux netlink as an ip services protocol, 2003.

[32] F. Schneider. Replication management using the state-machine approach. *Distributed Systems, ACM Press*, pages 169–197, 1991.

[33] A. K. Somani and N. H. Vaidya. Understanding fault-tolerance and reliability. In *IEEE Computer, Vol. 30, No. 4, pag.45-50*, apr 1997.

[34] F. Sultan, A. Bohra, S. Smaldone, Y. Pan, P. Gallard, I.Ñeamtiu, and L. Iftode. Recovering internet service sessions from operating system failures. In *IEEE Internet Computing*, apr 2005.

[35] G. Van Rooij. Real stateful tcp packet filtering in ip filter. In *10th USENIX Security Symposium*, Washington, D.C, USA, aug 2001.

[36] T. Wilfredo. Software fault tolerance: A tutorial. Technical report, NASA Langley Research Center, 2000.

[37] R. Zhang, T. Adelzaher, and J. Stankovic. Efficient tcp connection failover in web server cluster. In *IEEE INFOCOM 2004*, Hong Kong, China, mar 2004.

[38] X. Zhang, M. A. Hiltunen, K. Marzullo, and R. D. Schlichting. Customizable service state durability for service oriented architectures. In *IEEE Proceedings of EDCC-6: European Dependable Computing Conference*, Coimbra, Portugal, oct 2006.

[39] H. Zou and F. Jahanian. Real-time primary-backup (rtpb) replication with temporal consistency guarantees. In *ICDCS '98: Proceedings of the The 18th International Conference on Distributed Computing Systems*, page 48, Washington, DC, USA, 1998. IEEE Computer Society.