Máster en Computación Paralela y Distribuida

Departamento de Sistemas Informáticos y Computación

Trabajo de Fin de Máster

ACTUALIZACIÓN DE SERVICIOS DISTRIBUIDOS CON REPLICACIÓN PASIVA

Autor: **Javier Mendoza Paitán**
Director/Tutor: **Francisco Muñoz Escoí**

Septiembre de 2019

Curso 2018/2019

Contenido

Capítulo I: Introducción

Capítulo II: Conceptos Básicos de la Programación Concurrente

Capítulo III: Actualización Dinámica en Sistemas Distribuidos

Capítulo I: Introducción

1. Introducción

1.1. Antecedentes

Aunque los conceptos de programación secuencial y concurrente son claros al diferenciar sus características y aplicabilidad, el desarrollo de aplicaciones que impliquen mayor interacción y aleatoriedad de eventos presentan el desafío de elegir entre diferentes diseños de programación, incluso entre procesos paralelos, distribuidos o concurrentes, agregando dificultad a la implementación en la búsqueda de un equilibrio entre performance y claridad en la lectura del código de implementación. Los programas tienden a volverse más concurrentes. Comprender por qué la programación concurrente es difícil puede ayudarnos a resolver el problema. El mundo físico es altamente concurrente, y nuestra supervivencia depende de nuestra habilidad para razonar sobre la dinámica física concurrente [4].

Los sistemas distribuidos sirven a millones de usuarios en aplicaciones importantes, desde la banca hasta las redes sociales. Estos sistemas son difíciles de implementar correctamente debido a que tienen que lidiar con concurrencia y fallas, las máquinas pueden fallar en puntos arbitrarios y las redes pueden reordenar, duplicar o soltar paquetes. Además, el comportamiento es frecuentemente demasiado complejo como para permitir pruebas exhaustivas. Así, a pesar de décadas de investigación, las implementaciones en producción se realizan con errores críticos en el manejo de fallas, llevando a pérdida de información y caída de servicios. Por ejemplo, en Abril de 2011 un problema en la recuperación de fallas en el sistema cloud de Amazon EC2 ocasionó una gran caída importante de varios sitios web, incluyendo Foursquare, Reddit, Quora y PBS. Buscando facilitar a los programadores la implementación de sistemas distribuidos tolerantes a fallos correctos y con alta performance es que se han desarrollado frameworks como Verdi, que permite verificar de manera práctica y formal si las implementaciones cubren sus especificaciones respecto de sus propiedades de fiabilidad (safety) [8].

Los bancos tienen que lidiar con sistemas heredados heterogéneos que son difíciles de cambiar e integrar. La reducción de los márgenes de ganancia y el rápido crecimiento de la banca online impactan en el mercado financiero y requieren una alta capacidad de adaptación. Las restrictivas leyes de la banca [24] han resultado en una infraestructura tecnológica muy fragmentada, la cual debe ser adaptada para responder a las nuevas demandas requeridas por una banca virtual especializada, altamente automatizada y orientada al cliente. Por ello, la habilidad para adquirir una nueva arquitectura tecnológica es un factor competitivo clave para soportar las nuevas líneas de negocios financieras en banca. Estratégicamente, los bancos presentan cuatro principales desafíos relacionados a la arquitectura. En primer lugar, necesitan implementar la integración de sus aplicaciones de modo que sean capaces de conectar e incorporar nuevas funcionalidades a su entorno existente. En segundo lugar, necesitan establecer procesos de reconfiguración de valores y asegurar la preservación de sus valores después de procesos de fusión y/o adquisición. También, necesitan formas más ágiles de desarrollo

de sistemas de información, de modo que puedan hacer frente a entornos inestables que requieran cambios en sus sistemas con mayor frecuencia [7].

El entorno financiero presenta dos principales desafíos en el procesamiento, primero la gran cantidad de información generada debido a la diversidad de fuentes de origen, la variedad de información de los tarjetahabientes, el carácter temporal de los eventos que se registran y el número de instituciones que procesan, y segundo, el análisis del entrecruzamiento de la información espacial y temporal multiplicada por el número de instituciones diferentes en las que pueden participar los tarjetahabientes. De esto se desprende la necesidad de resolver estos desafíos mediante el desarrollo de aplicaciones concurrentes que sean fiables y seguras.

Uno de los servicios críticos de los procesos financieros es el monitoreo y generación de alertas de fraudes, lo cual consiste en analizar las diferentes transacciones financieras realizadas por los tarjetahabientes a lo largo del tiempo, siguiendo criterios de montos, lugares, simultaneidad de eventos, etc. Para ello, las instituciones envían la información de las transacciones en tramas con una estructura de campos con datos como fecha de transacción, hora de la transacción, lugar, monto, número de cuenta, medio de pago, moneda, tipo de tarjeta, número de identificación del tarjetahabiente, etc., la cual se procesa y almacena en una base de datos como información histórica. En la aplicación se establecen criterios o consultas condicionales a la base de datos, con carácter acumulativo, cuyo cumplimiento asigna un factor de riesgo a cada transacción, de modo que pasando cierto umbral son incluidas en la lista de transacciones inusuales. Luego, el analista del área de fraudes consulta el histórico de dicha transacción y descarta o confirma la transacción como fraude procediendo a bloquear la tarjeta.

Otro servicio crítico es la transferencia segura de información entre instituciones, de manera particular la transferencia de archivos batch para diversos procesos, como la información contable sobre las transacciones realizadas, las tarjetas bloqueadas y sitios de riesgo, las transacciones administrativas a realizar, etc. Por un lado, este servicio debe garantizar la seguridad del canal de transferencia y la consistencia de la información intercambiada, de modo que las instituciones cuenten con la información correcta y en los tiempos establecidos para garantizar la consecución de los procesos asociados. Por otro lado, las aplicaciones que lo implementan deben ser suficientemente automatizadas para permitir la mejor actuación del operador de los sistemas de producción, e inteligentes para resolver diferentes tipos de errores.

De manera concreta, en la implementación del Sistema de Transferencia de Archivos Segura de Unibanca, un proceso obtiene la lista de archivos de validación, otro proceso valida los archivos de la carpeta de entrada utilizando la lista previamente obtenida, lo que implica que debe obtener primero la lista de validación. A partir de la lista de archivos válidos, otro proceso genera tareas asíncronas y paralelas que comprimen y encriptan los archivos, y otro proceso gestiona el backup de los archivos procesados. De igual modo, se establecieron procesos background para la visualización de la lista de validación, las listas de archivos válidos e inválidos, los tasks generados y las listas de backup, estableciendo los criterios de visualización y procesamiento de información utilizando los estados de la información, el análisis del flujo de información y el test de los criterios. Esta división de funciones en hilos separados convenientemente puede

permitir que las etapas gestionen y encapsulen la obtención, actualización, procesamiento y manejo de errores de la información correspondiente optimizando los tiempos de ejecución de las diferentes etapas.

La interrupción de servicios como los descritos pueden poner en riesgo la continuidad de la entidad, tanto por las pérdidas económicas que significan las actividades fraudulentas mediante la captación de información de los tarjetahabientes, así como la pérdida de confiabilidad en la institución. Por ello, es necesario que los servicios implementados sigan diseños de alta disponibilidad y los sistemas distribuidos representan un modelo que se ajusta a esta necesidad. Estos sistemas deben ser capaces de tolerar las fallas de sus componentes y de permitir su actualización de manera dinámica sin necesidad de interrumpir o reiniciar los servicios que tiene asociados.

La actualización dinámica es una técnica que automatiza la gestión y el performance de los procesos de actualización, de modo que el sistema permanece operacional durante el tiempo de actualización [10]. Solarski en su tesis doctoral desarrolla un modelo que denomina "Deployment and Upgrade Facility", en el cual describe las capacidades necesarias para la gestión y el performance de las actualizaciones dinámicas así como las propiedades de los componentes de un programa actualizable. Dentro de este modelo, considera algoritmos para la actualización dinámica de sistemas distribuidos de replicación pasiva siguiendo el modelo primario-backup. Este modelo de replicación comprende una réplica o nodo primario, que recibe los requerimientos de los clientes, los procesa y envía las respuestas, además de distribuir su estado a varias réplicas o nodos backup, las cuales no procesan los requerimientos de los clientes pero sí actualizan su estado con los mensajes enviados por el primario.

El algoritmo planteado por Solarski para un sistema distribuido siguiendo el modelo primario backup considera como unidad de proceso de actualización el encendido de un nuevo nodo backup con la actualización requerida en el mismo ambiente de ejecución, su adición al sistema y la remoción del nodo backup antiguo correspondiente, lo cual se repite con cada nodo backup, para luego culminar con la selección de un nuevo nodo primario.

Sin embargo, es posible lograr un algoritmo que desacople dentro del proceso de actualización el encendido del nuevo nodo backup con el apagado del backup antiguo, de modo que se eliminen dependencias y agregar más concurrencia al proceso de actualización dinámica, de modo que la adición de controles o nuevos módulos resulte más sencillo y permita un mejor monitoreo del sistema.

1.2. Problema

Para agregar mayor concurrencia al proceso de actualización dinámica según el modelo de Solarski para un sistema distribuido siguiendo el modelo primario backup, se requiere plantear un modelo que desacople el proceso de encendido del nuevo nodo backup del apagado del nodo backup antiguo.

1.3. Planteamiento

Por lo tanto, planteo un modelo que implemente procesos concurrentes que permitan desacoplar los procesos de adición de los nodos backup actualizados y los procesos de eliminación de los nodos backup antiguos durante el proceso de actualización dinámica de un sistema distribuido siguiendo el modelo primario-backup. Este modelo considera que la actualización de las variables de estado de los nodos backup también se realice de modo concurrente, de modo que el incremento del grado de concurrencia en el diseño del sistema distribuido puede generar una mayor asincronía del proceso de actualización dinámica y es posible esperar que los costes de tiempo sean menores. El modelo considera que el tamaño del sistema distribuido durante el proceso de actualización asegura que se cumpla con el número de nodos correspondiente al nivel de tolerancia de fallos para el cual haya sido diseñado. Es decir, que garantiza que siempre habrá f + 1 réplicas funcionando asumiendo que podrá haber f fallos simultáneos.

2. Organización del TFM

El desarrollo del TFM abarca conceptos de concurrencia, replicación pasiva, actualización dinámica de un sistema distribuido, así como el proceso de implementación y evaluación del modelo propuesto, lo que será descrito en los siguientes capítulos.

En el Capítulo II se revisarán los conceptos de programación concurrente, primitivas de sincronización y una introducción a la programación concurrente orientada a objetos. Esto proporciona la base necesaria para comprender la implementación de los nodos clientes y servidores (primario y backups). Estas aplicaciones fueron diseñadas considerando procesos concurrentes para el envío de los requerimientos, el envío y la recepción de las respuestas, la ejecución de tareas, el control de viveza, la elección de líder, la actualización de la variables de estado de las réplicas y la actualización dinámica. La implementación de estos procesos concurrentes requiere el uso de listas compartidas entre diferentes procesos, utilizándose para ello clases de colecciones que implementan el acceso concurrente de varios procesos o threads.

Dentro del contexto del servicio a las entidades bancarias, uno de los servicios requeridos por las instituciones es el intercambio de archivos con información crítica que requiere diversos mecanismos de compresión y seguridad para transferir la información de manera segura desde el centro de procesamiento hacia las entidades y viceversa. Por otro lado, estos servicios son brindados por aplicaciones preexistentes con un diseño procedural y secuencial que se ven sobrecargados a medida que se adicionan más clientes (entidades bancarias) y la dimensión de los archivos crece acorde al crecimiento del número de clientes y de transacciones de cada institución bancaria.

Estando trabajando en el área de Soporte de Sistemas de la empresa Unibanca que brinda servicios a diferentes entidades bancarias, se me asignó el proyecto de migración del sistema de transferencia segura de archivos que poseía la empresa debido a que funcionaba en Windows XP, sistema operativo que está fuera de soporte por parte de Microsoft y por lo tanto no cumple con las normas de seguridad PCI que la empresa está obligada a cumplir, lo que ponía en peligro la continuidad del servicio. La aplicación estaba elaborada en C/C++ 32bits utilizando Turbo C, la codificación era procedural (principalmente desarrollada en C), los archivos se procesaban de manera secuencial (uno por uno) y utilizaba menús de consola

de modo que las operaciones de envío y recepción de archivos se realizaba de manera manual paso a paso. El requerimiento incluía la migración hacia una plataforma soportable y portable entre diferentes versiones de Windows con soporte (Windows 7 en adelante), la optimización del procesamiento de los archivos, la automatización del envío y recepción de los archivos y la menor interacción del operador con la aplicación, de modo que su función se reduzca lo más posible a sólo la inspección del flujo de procesos y el inicio y/o fin del servicio. También requería la gestión de los archivos, un proceso de depuración opcional para los clientes y el proceso de cifrado y descifrado.

Analizando los diferentes procesos involucrados, se observó que los archivos, que podían tener diferentes dimensiones y diferentes destinos, y se les aplicaban procesos de cifrado/descifrado y compresión, no presentaban dependencia de procesos entre sí, por lo que un diseño paralelo en su gestión se ajustaba perfectamente. Por otro lado, con la finalidad de segmentar adecuadamente el servicio de manera que sea monitoreable por los operadores, se identificaron y separaron las diferentes actividades involucradas de modo que sean ejecutadas por diferentes hilos de manera asíncrona. Sin embargo, al ser procesos interrelacionados por la información a procesar, requerían listas o diccionarios de objetos compartidos entre los diferentes hilos, además de un grado de sincronización entre ellos para lograr la consistencia requerida respecto de la información intercambiada.

La implementación se realizó utilizando .NET, C/C++ y programación orientada a objetos. Los procesos se implementaron de manera concurrente utilizando las clases BackgroundWorker, ConcurrentBag y ConcurrentDictionary, mientras que los archivos se gestionaron mediante Tasks.

Basado en esta experiencia, apliqué un diseño concurrente para los diferentes procesos del sistema distribuido implementado en el presente TFM, tanto en el cliente como en los servidores y en los diferentes procesos de comunicación y gestión de la actualización dinámica.

En el Capítulo III se revisarán los conceptos de Actualización Continua como mecanismo de Actualización Dinámica que se va a implementar en el sistema distribuido del presente TFM, debido a que el objeto de actualización será un nodo y el proceso de actualización nodo a nodo se ajusta más al modelo implementado.

En el Capítulo IV se revisarán el modelo primario - backup de replicación pasiva utilizado para implementar el sistema distribuido cliente/servidor y el modelo de actualización dinámica propuesto por Solarski. Este modelo será tomado como base para el desarrollo del modelo de actualización desacoplado propuesto en el presente trabajo.

En el Capítulo V se revisará el modelo del sistema, en el cual se describirá el modelo de actualización desacoplado planteado e implementado para la actualización de los nodos del sistema distribuido.

En el Capítulo VI se presentará el diseño de la aplicación distribuida, mostrando los procesos de los nodos clientes, primario y backup, los módulos que los componen según las funciones a realizar mostrando su carácter concurrente, los mecanismos de comunicación entre los nodos y el proceso de envío y recepción de mensajes de requerimientos y respuestas, así como

la ejecución de los procesos que componen la actualización dinámica planteada para un sistema de 3 nodos.

En el Capítulo VII se discutirán los resultados de los tiempos obtenidos durante el proceso de actualización y se presentarán las conclusiones correspondientes.

Capítulo II: Conceptos básicos de la programación concurrente

3. Programación Concurrente

La programación concurrente se ha convertido importante para todo tipo de aplicaciones, incluyendo sistemas de bases de datos, computación paralela a gran escala, y sistemas de control embebido y en tiempo real. Un programa concurrente especifica dos o más programas secuenciales que pueden ser ejecutados concurrentemente como procesos paralelos. Por ejemplo, un sistema de reservación de aerolíneas que involucra el procesamiento de transacciones desde muchos terminales, en el cual cada terminal es controlado por su propio proceso secuencial. Aun cuando los procesos no sean ejecutados simultáneamente, es más fácil estructurar un sistema como una colección de programas secuenciales cooperativos que como un simple programa secuencial.

Un programa concurrente puede ser ejecutado permitiendo que los procesos compartan uno o más procesadores, o corriendo cada proceso en su propio procesador. La primera aproximación es referida como multiprogramación, que es soportada por un kernel de sistema operativo que multiplexa los procesos en los procesadores. La segunda aproximación es referida como multiprocesamiento si los procesadores comparten memoria, o como procesamiento distribuido si los procesadores están conectados por una red de comunicaciones.

Siendo la intención comprender un programa concurrente en términos de sus procesos secuenciales y sus interacciones, sin importar como sean ejecutados, no hacemos ninguna suposición sobre la tasa de ejecución de los procesos ejecutados concurrentemente, salvo que todas son positivas. Esta es llamada la suposición de progreso finito. Entonces, la exactitud del programa es independiente de si es ejecutado en múltiples procesadores o en un solo procesador.

La implementación del sistema distribuido con actualización dinámica desarrollado en el presente TFM es inherentemente concurrente. El servicio ofrecido por el sistema, los procesos propios del sistema distribuido como elección de líder, pertenencia, viveza, y la comunicación entre las réplicas se han diseñado considerando una segmentación adecuada de las funciones que las componen, de modo que se ejecuten en hilos separados de manera concurrente y siguiendo primitivas tanto de variable compartida como de paso de mensajes.

El punto de partida para el desarrollo del diseño concurrente fue la necesidad de migrar una aplicación bancaria requerida para la transferencia segura de archivos entre instituciones que pudiera manejar una cantidad creciente de archivos de diferentes tipos con una cantidad creciente de clientes y en ambas direcciones, requiriendo a la vez el intercambio de listados de validación y pasos de mensajes entre los clientes y el servidor en tiempo de ejecución. Mi objetivo en el desarrollo de ese proyecto fue mejorar el rendimiento de la nueva aplicación mediante la gestión de múltiples hilos de ejecución en cada uno de sus procesos. Por este motivo resulta conveniente la revisión de los conceptos básicos de programación concurrente

pues constituyen la base del proceso de implementación del modelo presentado en el presente TFM.

En el caso del servicio ofrecido por el sistema distribuido, el esquema global en el servidor incluye la recepción de requerimientos, la generación de tareas ("Tasks") para cada requerimiento, la generación de las respuestas por cada requerimiento, los objetos de actualización del estado del sistema y los mensajes de envío de respuesta hacia el cliente, entre otros. Los requerimientos se almacenan en un ConcurrentBag que se comparte con el proceso de generación de Tasks, el cual a su vez comparte el ConcurrentBag de tareas generadas para que el proceso de generación de mensajes de respuesta pueda enviar éstos hacia el cliente.

Los diferente procesos que implementan el sistema distribuido y la actualización dinámica, utilizan primitivas de sincronización de paso de mensajes mediante funciones *send* y *receive*.

3.1. Interacción entre procesos

Con la finalidad de cooperar, los procesos ejecutados concurrentemente deben comunicarse y sincronizar. La comunicación interprocesos está basada en el uso de variables compartidas o en paso de mensajes. La sincronización es necesaria cuando los procesos se comunican. Los procesos son ejecutados con velocidades impredecibles. Para la comunicación, un proceso debe realizar acciones que los otros detecten. Esto sólo funciona si los eventos "realizar una acción" y "detectar una acción" están restringidos a ocurrir en tal orden. Así, uno puede ver la sincronización como una serie de restricciones en la ordenación de eventos. De este modo, se emplean mecanismos de sincronización para retrasar la ejecución de un proceso en orden a satisfacer dichas restricciones.

La ejecución de un programa concurrente puede verse como una secuencia de acciones atómicas, cada una resultante de la ejecución de una operación indivisible. Esta secuencia presentará algún entrelazado de las secuencias de acciones atómicas generadas por los procesos componentes. Raramente todas las ejecuciones entrelazadas resultarán en un comportamiento aceptable del programa. Los comportamientos anómalos pueden ser evitados previniendo la ejecución entrelazada de las sentencias, esto es, controlando el orden de los eventos correspondientes a las acciones atómicas.

Los mecanismos de sincronización controlan las interferencias de dos modos. Primero, pueden suspender la ejecución de un proceso hasta que una determinada condición (afirmación) sea verdadera. Haciendo esto, ellos aseguran que la precondición de la subsecuente sentencia sea verdadera. Segundo, un mecanismo de sincronización puede ser usado para asegurar que un bloque de sentencias sea una operación indivisible.

Sin embargo, existen limitaciones dado que el número de ejecuciones entrelazadas crece exponencialmente con la cantidad de procesos secuenciales.

3.2. Primitivas de sincronización basadas en variables compartidas

Cuando variables compartidas son usadas para comunicación interprocesos, dos tipos de sincronización son útiles: exclusión mutua y sincronización condicional. La exclusión mutua asegura que una secuencia de sentencias sea tratada como una operación indivisible. Una secuencia de sentencias que debe ser ejecutada como una operación indivisible se llama *sección crítica* [1]. Debe notarse que los efectos de las ejecuciones entrelazadas son visibles sólo si dos cálculos acceden a variables compartidas. Si este es el caso, un cálculo puede ver resultados intermedios producidos por la ejecución incompleta de otros. Si dos rutinas no tienen variables en común, entonces sus ejecuciones no necesitan ser mutuamente excluyentes.

Otra situación en la cual es necesario coordinar la ejecución de procesos concurrentes ocurre cuando un objeto de datos compartido está en un estado inapropiado para la ejecución de una operación particular. Cualquier proceso intentando dicha operación debe ser suspendido hasta que el estado del objeto de datos cambie como resultado de la ejecución de otros procesos. Llamamos a esto *sincronización condicional* [1].

3.2.1. Espera activa

Un modo de implementar la sincronización es tener procesos que asignen y evalúen variables compartidas. Esta aproximación trabaja razonablemente bien para la implementación de sincronización compartida pero no para la exclusión mutua. Para señalar una condición, un proceso asigna el valor de una variable compartida; para esperar por esta condición, un proceso evalúa repetidamente la variable hasta que encuentra que tiene el valor deseado. Como el proceso esperando una condición debe repetidamente evaluar la variable compartida, esta técnica de suspender un proceso es llamada *espera activa* [1] y el proceso se dice que está en spinning (giro). Las variables que son usadas de este modo son llamadas spin locks (bloqueos de giro).

Para implementar la exclusión mutua usando Espera Activa, las sentencias que señalan y esperan por condiciones son combinadas en protocolos cuidadosamente elaborados. Por ejemplo, la solución de Peterson [15] para el problema de exclusión mutua de dos procesos involucra un protocolo de entrada, que un proceso ejecuta antes de entrar en su sección crítica, y un protocolo de salida, que un proceso ejecuta después de finalizar su sección crítica:

```
PROCESO P1;
        loop
           Entry Protocol;
            Critical Section;
           Exit protocol;
           Noncritical Section;
        end
end
```

```
PROCESO P2;
        loop
          Entry Protocol;
           Critical Section;
          Exit protocol;
          Noncritical Section;
        end
end
```

En adición a la implementación de la exclusión mutua, esta solución tiene otras propiedades deseables. Primero, es libre de interbloqueos [25] (estado en el cual dos procesos están esperando por eventos que nunca ocurrirán). Segundo, es Equitativo, esto es, si un proceso está tratando de entrar a su sección crítica, eventualmente lo hará, dado que el otro proceso saldrá de su sección crítica. La Equidad es una propiedad deseable para un mecanismo de sincronización porque su presencia asegura que la suposición de progresión finita no es invalidada por suspensiones debidas a la sincronización. En general, un mecanismo de sincronización es equitativo si ningún proceso es suspendido por siempre esperando por una condición que ocurre infinitamente a menudo. Es de Espera Limitada si existe un límite superior sobre cuánto un proceso será suspendido esperando por una condición que ocurre infinitamente a menudo.

Los protocolos de sincronización que usan solo espera activa son difíciles de diseñar, comprender y probar si son correctos.

3.2.2. Semáforos

Un *semáforo* [16] es una variable entera no negativa sobre la cual se definen dos operaciones: P y V. Dado un semáforo s, P(s) se retrasa hasta que s > 0 y entonces ejecuta s = s - 1; la evaluación y decremento son ejecutados como una sola operación indivisible. V(s) ejecuta s = s + 1 como una operación indivisible. La mayoría de las implementaciones de semáforos son equitativas: ningún proceso suspendido mientras se ejecuta P(s) permanecerá así siempre si las operaciones de V(s) son realizadas infinitamente a menudo. La necesidad de equidad surge cuando un número de procesos son simultáneamente suspendidos, todos intentando ejecutar una operación P con el mismo semáforo. Claramente, la implementación debe escoger cuál se permitirá proceder cuando V es ejecutado. Un modo simple de asegurar la equidad es despertar los procesos en el orden en que fueron suspendidos.

Los semáforos son una herramienta muy general de resolver los problemas de sincronización. Para implementar una solución al problema de exclusión mutua, cada sección crítica es precedida por una operación P y seguida por una operación V con el mismo semáforo. Todas las secciones críticas mutuamente excluyentes usan el mismo semáforo, el cual es inicializado a uno. Como tal semáforo solo toma valores uno y cero, son llamados semáforos binarios.

Para implementar la sincronización condicional, variables compartidas son utilizadas para representar la condición, y un semáforo asociado con la condición es utilizado para conseguir la sincronización. Después que un proceso ha tornado la condición verdadera, señala que esto se ha hecho ejecutando una operación V; un proceso es suspendido hasta que la condición sea verdadera ejecutando una operación P.

Un semáforo que puede tomar cualquier valor no negativo es llamado semáforo general o de conteo. Los semáforos generales son utilizados frecuentemente para la sincronización condicional en el control de la distribución de recursos. Tales semáforos tienen como valor inicial el número de unidades del recurso; P es utilizado para suspender un proceso hasta que una unidad de recurso esté disponible; V es ejecutado cuando una unidad de recurso es devuelta.

VAR **MUTEX**: *semáforo initial*;

```
PROCESO P1;
        loop
           P(MUTEX);                {Entry Protocol}
      Critical Section;
     V(MUTEX);                      {Exit protocol}
     Noncritical Section;
        end
end
```

```
PROCESO P2;
        loop
    P(MUTEX);                       {Entry Protocol}
     Critical Section;
    V(MUTEX);                       {Exit protocol}
    Noncritical Section;
        end
end
```

El uso de P y V asegura la exclusión mutua y la ausencia de interbloqueos. Además, si la implementación del semáforo es equitativa y ambos procesos siempre salen de sus secciones críticas, cada proceso eventualmente entrará en su sección crítica.

Los semáforos pueden ser usados para resolver problemas de exclusión mutua selectiva. En este caso, las variables compartidas son particionadas en conjuntos disjuntos. Un semáforo es asociado con cada conjunto y usado para controlar el acceso a las variables en cada conjunto. Las secciones críticas que referencian variables en el mismo conjunto se ejecutan con exclusión mutua, pero las secciones críticas que referencian variables en diferentes conjuntos se ejecutan concurrentemente.

Los semáforos pueden ser implementados utilizando espera activa. Sin embargo, comúnmente son implementados con llamadas a sistema de un kernel. Un kernel

implementa procesos en un procesador. En cualquier momento, un proceso o está listo para ejecutarse en el procesador o está bloqueado, esperando que se complete una operación P. El kernel mantiene una lista de espera (*ready list*) - una cola de descriptores para procesos listos - y multiplexa el procesador entre los procesos, corriendo cada proceso por un periodo de tiempo. Los descriptores para procesos que están bloqueados en un semáforo son guardados en una cola asociada con el semáforo, no son guardados en la lista de espera (*ready list*), de ahí que estos procesos no serán ejecutados. La ejecución de las operaciones P o V causa un llamado a las rutinas del kernel. Para una operación P, si el semáforo es positivo, es decrementado, sino el descriptor para el proceso en ejecución es movido a la cola del semáforo. Para una operación V, si la cola del semáforo no está vacía, un descriptor es movido de esa cola a la lista de espera (*ready list*), sino el semáforo es incrementado.

3.2.3. Región crítica condicional

Aunque los semáforos pueden ser usados para programar casi cualquier tipo de sincronización, P y V son primitivas no estructuradas y podrían llevar a errores durante su uso. La ejecución de cada sección crítica debe empezar con un P y terminar con un V (en el mismo semáforo). La omisión de P o V, o la codificación de ellos en semáforos diferentes puede llevar a efectos desastrosos, dado que la exclusión mutua no estaría asegurada. El programador puede olvidar incluir en la sección crítica todas las sentencias que referencian a objetos compartidos, con el mismo efecto sobre la exclusión mutua. Una segunda dificultad es que tanto la sincronización condicional como la exclusión mutua son programadas usando el mismo par de primitivas, lo que hace difícil identificar el propósito de una operación P o V dada sin mirar las otras operaciones en el semáforo. Dado que la exclusión mutua y la sincronización condicional son conceptos distintos, deberían tener diferentes notaciones.

La *región crítica condicional* [17] supera dichas dificultades proveyendo una notación estructurada para la sincronización. Las variables compartidas son colocadas en grupos llamados recursos. Cada variable compartida puede estar solo en un recurso y solo puede ser accedida por las sentencias de la región crítica condicional (RCC) que llamen a dicho recurso. La exclusión mutua se logra garantizando que la ejecución de diferentes sentencias de la RCC, que llaman al mismo recurso, no se superponga. La sincronización condicional se logra por las condiciones booleanas explícitas en las sentencias de la RCC.

A pesar de sus virtudes, la implementación de las RCC puede ser costosa. Dado que las condiciones en las sentencias de la RCC pueden contener referencias a variables locales, cada proceso debe evaluar sus propias condiciones.

3.2.4. Monitores

Un *monitor* [18] se forma encapsulando la definición de un recurso y las operaciones que lo manipulan. Esto permite ver como un módulo al recurso sujeto al acceso concurrente. Por lo tanto, el programador puede ignorar los detalles de la implementación del recurso y su modo de uso.

Un monitor consta de una colección de variables permanentes, usadas para guardar el estado del recurso, y algunos procedimientos, los cuales implementan las operaciones del recurso. También tiene un código de inicialización de las variables, que es ejecutado previo a la ejecución de sus procedimientos. Los valores de las variables permanecen entre las activaciones de los procedimientos y solo son accesibles desde dentro del monitor. Los procedimientos pueden tener parámetros y variables locales, cada uno de los cuales toman nuevos valores para cada activación. La ejecución de los procedimientos en un monitor dado es mutuamente excluyente, lo que asegura que las variables permanentes nunca son accedidas concurrentemente.

En la propuesta de Hoare [19], una variable condicional es usada para retrasar la ejecución de los procesos en un monitor, que solo puede ser declarada dentro del monitor. Se definen dos operaciones para las variables condicionales: *signal* y *wait*. Sea *cond* una variable condicional, la ejecución de *cond.wait* causa que el invocador sea bloqueado en *cond* y que abandone el control de su exclusión mutua por el monitor. La ejecución de *cond.signal* permite lo siguiente: si ningún proceso es bloqueado en *cond*, el invocador continúa, de lo contrario, el invocador es temporalmente suspendido y un proceso bloqueado en *cond* es reactivado. Un proceso suspendido debido a una operación *signal* continúa cuando no hay otro proceso ejecutándose en el monitor. Las variables condicionales se consideran equitativas en el sentido de que un proceso no permanecerá suspendido por siempre en una variable condicional que ejecuta *signal* infinitamente a menudo. La introducción de las variables condicionales permite que más de un proceso se encuentren en el mismo monitor, aunque todos menos uno serán retrasados en las operaciones *wait* y *signal*. El programador requerirá mayor control en el orden en el cual los procesos retrasados serán activados.

3.3. Primitivas de sincronización basadas en paso de mensajes

Cuando se usa paso de mensajes para la comunicación y sincronización, los procesos envían y reciben mensajes en lugar de la lectura y escritura de variables compartidas. La comunicación se logra porque un proceso, al recibir un mensaje, obtiene valores desde otros procesos. La sincronización se logra porque un mensaje puede ser recibido únicamente después de que haya sido enviado, lo cual restringe el orden en el cual estos dos eventos pueden ocurrir.

Un mensaje es enviado ejecutando:

send *expression_list* **to** *destination_designator*

y es recibido ejecutando:

receive *variable_list* **from** *source_designator*

El diseño de primitivas de paso de mensaje implica decidir sobre la forma y la semántica de estos mandatos generales. Dos aspectos deben tenerse en cuenta: Cómo especificar la fuente y el destinatario, y cómo se sincroniza la comunicación.

3.3.1. Canales de comunicación

En conjunto, la fuente y el destinatario definen un canal de comunicación. El esquema de nombres más simple para el canal de comunicación es utilizando los nombres de los procesos, referido como nombrado directo (*direct naming*) [1].

El nombrado directo es fácil de implementar y usar, permitiendo a un proceso tener control de las veces que recibe mensajes de otros procesos. Un paradigma importante para la interacción de los procesos es la tubería (*pipeline*). Una tubería es una colección de procesos concurrentes en los cuales la salida de cada proceso es utilizado como la entrada de otro. Otro paradigma importante es la relación *cliente/servidor* [1]. Los procesos servidores ofrecen servicios a los procesos clientes. Un cliente puede solicitar un servicio mediante el envío de un mensaje a alguno de los servidores. El servidor recibe repetidamente solicitudes de servicio por parte de los clientes, ejecuta los servicios y, si es requerido, retorna mensajes de finalización de ejecución del servicio a los clientes.

Desafortunadamente, el nombrado directo no se ajusta bien para la interacción cliente/servidor, dado que puede haber muchos clientes para un servidor o el cliente puede enviar un mensaje que puede ser recibido por cualquier servidor. Por lo tanto, se requiere de esquemas más sofisticados para definir los canales de comunicación. Uno de estos esquemas está basado en el uso de nombres globales, llamado buzones de correo (*mailboxes*). Un buzón de correo puede aparecer como el destinatario de las sentencias *send* de cualquier proceso y como la fuente de las sentencias *receive* de cualquier proceso.

Este esquema se adecúa bien para sistemas cliente/servidor. Los clientes envían sus requerimientos de servicio a un buzón de correo, los servidores reciben los pedidos de servicio desde este buzón de correo. El caso particular de buzones, en el cual el nombre de un buzón aparece como la fuente en las sentencias *receive* en un único proceso, reciben el nombre de puertos. Los puertos son fáciles de implementar, dado que todos los *receive* que señalan a un puerto ocurren en un mismo proceso. Los puertos permiten una solución directa al problema de múltiples clientes/un servidor, aunque el problema de múltiples clientes/múltiples servidores no es tan fácil de resolver con puertos.

3.3.2. Sincronización

Otra importante propiedad de las sentencias de paso de mensajes es si su ejecución podría causar una suspensión. Una sentencia es no bloqueante si su ejecución nunca suspende al que lo invoca, de otro modo se dice que es bloqueante. En algunos esquemas de paso de mensajes, éstos son almacenados en un buffer durante su envío y recepción. Entonces, si el buffer está lleno cuando se ejecuta el *send* hay dos opciones, el *send* puede ser suspendido hasta que haya espacio en el buffer, o el *send* puede retornar un código indicando que, debido a que el buffer está lleno, no pudo enviar el mensaje. De manera similar con el *receive*.

Si el sistema posee un buffer de capacidad ilimitada, entonces un proceso nunca es suspendido cuando se ejecute el *send*. Esto es llamado paso de mensajes **asíncrono** y *send no-wait*. El paso de mensajes asíncrono permite al emisor estar arbitrariamente por delante del receptor. Consecuentemente, cuando un mensaje es recibido, contiene información sobre el estado del emisor que no necesariamente corresponde con el estado actual. Por otro lado, cuando no se establece un buffer, la ejecución del *send* es siempre suspendida hasta que un correspondiente *receive* haya sido ejecutado. Esto es llamado paso de mensajes **síncrono**. Cuando es usado, el intercambio del mensaje representa un punto de sincronización en la ejecución del emisor y receptor. De ahí que el mensaje recibido siempre corresponderá con el estado actual del emisor. Además, cuando el *send* finaliza, el emisor puede afirmar sobre el estado del receptor. Entre estos dos extremos está el paso de mensajes con buffer en el cual el buffer tiene límites finitos.

4. Programación Concurrente orientada a objetos

Los objetos y la concurrencia han estado ligados desde mucho antes. El primer lenguaje concurrente orientado a objetos, Simula, fue también el primer lenguaje orientado a objetos y entre los primeros lenguajes concurrentes. Otros lenguajes también brindaron concurrencia y construcciones orientadas a objetos, como las primeras versiones de C++ y Ada [5].

La propiedad que inicialmente sugiere una directa relación entre las ideas de concurrencia y la orientación a objetos es la remarcable similitud entre las construcciones básicas de ambos. Es difícil obviar la analogía entre objetos y procesos, o más precisamente entre las abstracciones: clases y tipos de proceso [6]. Ambas categorías implican:

- Variables locales (atributos de una clase, variables de un proceso o tipo de proceso).
- Datos persistentes, manteniendo sus valores entre sucesivas activaciones.
- Encapsulamiento (un ciclo para un proceso, cualquier número de rutinas para una clase).
- Restricciones en cómo los módulos pueden intercambiar información.
- Un mecanismo de comunicación basado en alguna forma de paso de mensajes.

4.1. El modelo SCOOP (Simple Concurrent Object-Oriented Programming)

Este modelo utiliza la programación orientada a objetos (POO) en la forma de los conceptos del Diseño por Contrato y los extiende para cubrir la concurrencia y

distribución. Para cubrir los requerimientos de la programación concurrente, esto es, exclusión mutua, sincronización y condiciones de espera, SCOOP [2] da nueva semántica a estructuras bien conocidas (paso de argumentos, precondiciones) donde la semántica secuencial estándar no podría ser aplicada. El modelo es aplicable en diferentes entornos, tales como multiprocesamiento, multihilos, redes, servicios web y computación distribuida. Toma ventaja de la inherente concurrencia implícita en la programación orientada a objetos al blindar al programador de conceptos de bajo nivel tales como semáforos, permitiéndole producir aplicaciones concurrentes de modo equivalente a las secuenciales.

4.1.1. Arquitectura

SCOOP tiene una arquitectura de dos niveles. El nivel superior del mecanismo es independiente de la plataforma. Para realizar los cálculos concurrentes, las aplicaciones usan el mecanismo "**separate**" implementado a este nivel. Internamente, se basa en las implementaciones de arquitecturas concurrentes disponibles en diferentes librerías, que pueden incluir una implementación de .NET utilizando los espacios de nombres *System.Runtime.Remoting* y *System.Threading*, una implementación en hilos utilizando POSIX, etc.

4.1.2. Procesadores

En POO el mecanismo básico es una llamada a la propiedad del modo *x.f(a)* con la siguiente semántica: el objeto cliente llama a la propiedad *f* del objeto asociado a *x*, con el argumento *a*. En un escenario secuencial, tales llamadas son **síncronas** y bloquean al cliente hasta la ejecución de la propiedad.

Para soportar la concurrencia, SCOOP permite el uso de más de un procesador para gestionar la ejecución de las propiedades. Cada objeto tiene un gestor: un procesador encargado de la ejecución de las llamadas a las propiedades del objeto. Si el cliente y el objeto cuya propiedad se invoca tienen diferentes gestores, la llamada se vuelve **asíncrona**, esto es, el cómputo del objeto 1 puede continuar sin esperar a que termine la llamada del objeto 2.

Los procesadores son el nuevo concepto principal para agregar concurrencia a la programación orientada a objetos secuencial: mientras un sistema secuencial está limitado a un procesador, un sistema concurrente puede tener cualquier número de procesadores. Un procesador es un hilo de control autónomo capaz de soportar la ejecución secuencial de las instrucciones de uno o varios objetos. Es un concepto abstracto. No tiene que estar asociado a un procesador físico, y puede ser implementado por un proceso del sistema operativo o un hilo en un contexto multihilos. En .NET los procesadores pueden ser asociados a dominios de aplicación (*application domains*).

4.1.3. Llamadas separadas

Dado que el efecto de una llamada depende de si el cliente y el objeto invocado son gestionados por el mismo procesador o por diferentes procesadores, la

codificación debe distinguir claramente entre estos dos casos. La palabra clave "**separate**" en la declaración de una variable, función o clase indicará que es gestionada por un procesador diferente, de modo que la llamada será asíncrona. Con esta declaración, cualquier instrucción de creación **create** *x.make(...)* usará un nuevo procesador para gestionar las llamadas a *x*.

4.1.4. Sincronización y comunicación

SCOOP dirige las necesidades de sincronización y comunicación de la programación concurrente, incluyendo la exclusión mutua, el bloqueo y la espera, por medio del paso de argumentos y el diseño por contrato.

4.1.4.1. Paso de argumentos

Una regla básica de SCOOP es que una llamada separada de *x, x.f(...)*, donde *x* es separada, sólo es permitida si *x* es un argumento de la rutina que la encierra, y la llamada a dicha rutina con tal argumento separado esperará hasta que el objeto separado esté exclusivamente disponible para el que lo invoca. Así, si se llama *r(a)* o *y.r(a)* con

r(x: **separate** *SOME_TYPE)* **is** ...

la llamada esperará hasta que ningún otro cliente esté usando a de este modo. Esta regla provee el mecanismo básico de sincronización.

4.1.4.2. Precondiciones

Una precondición que involucra una llamada a un objeto separado es llamada precondición separada. En programas secuenciales, las precondiciones son requerimientos de exactitud (*correctness*) que el objeto cliente debe cumplir antes de llamar la rutina del objeto proveedor. Si una o más precondiciones no son cumplidas, el cliente ha roto el contrato. Por ejemplo, tratar de guardar un valor en un buffer lleno. Si la ejecución es secuencial, el estado del buffer no puede cambiar (ningún otro cliente puede tratar de consumir un elemento del buffer en el intermedio).

En un contexto concurrente esto no aplica, el buffer puede estar lleno cuando el cliente está tratando de guardar un valor en él, pero nada previene que otro cliente consuma un elemento del buffer más tarde. Una precondición separada no satisfecha no necesariamente rompe el contrato, esto solo fuerza al cliente a esperar hasta que la precondición sea satisfecha.

5. Caso de estudio: Sistema de Transferencia Segura de Archivos de Unibanca

Como parte de sus servicios, Unibanca ofrece a sus clientes la transferencia de archivos para su procesamiento mediante un canal seguro por medio de mecanismos de cifrado basados en SSH y un protocolo propietario ofrecido por la aplicación SPAZIO. Este mecanismo presenta una capa de gestión implementada con un diseño estructural y secuencial, utilizando pasos de mensajes mediante colas y un sistema de menús para la selección de las operaciones a realizar. Sin embargo, la creciente demanda de dicho servicio, el incremento del número y tamaño de los archivos intercambiados y el procesamiento manual generaron problemas en los tiempos de respuesta de los procesos que dependen de los archivos intercambiados, lo que originó el reclamo de los clientes. Por otro lado, el sistema estaba desarrollado para Windows XP, sistema que ya no posee soporte por parte de Microsoft, lo que generó observaciones en las auditorías a las que regularmente están sujetas las instituciones cliente. Como consecuencia, se presentó el requerimiento de migrar la aplicación utilizando una versión más reciente de SPAZIO y el rediseño de la capa de gestión que permita el procesamiento automático y concurrente de los archivos, añadiendo mecanismos de depuración y una mejora en la interface de usuario a utilizar por parte de los operarios.

El rediseño de esta capa de gestión involucró la utilización de programación orientada a objetos, la utilización de programación concurrente, una adaptación del patrón MVC para el diseño de la aplicación y el uso de clases de tipo FORM para la generación de la interfaz gráfica de usuario. Se separaron las capas de visualización, de control y lógica de negocio y de persistencia. Tanto a nivel de visualización como de backend se separaron las diferentes funciones a realizar en procesos concurrentes, los cuales requirieron el uso de listas compartidas que soporten su acceso de manera concurrente. Esto permitió escalar en el número de funciones a implementar y una mayor granularidad en el proceso de monitoreo del sistema, aunque también agregó un mayor número de variables de estado a interrelacionar. Los procesos implementados incluyeron la gestión de los archivos en los proceso de cifrado, descifrado, compresión, descompresión y derivación a los clientes o el servidor según el flujo en el cual se encuentre, la validación de los archivos según la lista de control proveniente del servidor, los procesos de backup y depuración de los archivos, la visualización del estado de las tareas desarrolladas, los procesos de intercambio de mensajes entre los clientes y el servidor, y la generación de información del sistema mediante archivos log, entre otros.

Las pruebas durante la etapa de certificación mostraron una gestión concurrente y automática de archivos de diferentes tamaños en tiempos esperables, y de aprobación de los clientes durante la etapa de evaluación en producción, lo cual mostró que el nuevo diseño mejoró el funcionamiento de la aplicación que se tradujo en un mejor tiempo de respuesta a los requerimientos de los clientes.

6. Conclusiones

La implementación de los procesos asociados a los servicios que ofrece un sistema distribuido, a la gestión de la integridad del sistema y la comunicación entre las diferentes réplicas y los clientes, requieren de la aplicación de mecanismos concurrentes que permitan el intercambio de información entre los diferentes procesos. Esto se puede lograr mediante primitivas de sincronización basadas en variable compartida o mediante primitivas de sincronización basadas en paso de mensajes. Dada la naturaleza altamente comunicativa de los procesos propios del sistema distribuido, una comprensión de las capacidades que puede brindar la programación concurrente nos permitirá proyectar los mecanismos y modelos más

adecuados durante el proceso de diseño e implementación del sistema y los servicios que ofrecerá.

Capítulo III: Actualización Dinámica en Sistemas Distribuidos

7. Evolución Online y crecimiento de sistemas altamente disponibles

Uno de los principios tradicionales de los sistemas altamente disponibles es el cambio mínimo. Esto entra directamente en conflicto con las tasas de crecimiento de los servicios a gran escala y la liberación de nuevas versiones. Para estos servicios, se deben crear planes de crecimiento continuo y la actualización frecuente de su funcionalidad. Podemos considerar los procesos de mantenimiento y actualización como caídas controladas del sistema, de modo que la evolución online [11], esto es, la aplicación de actualizaciones sin la caída del servicio, es importante dado que estos servicios a gran escala son actualizados muy frecuentemente.

El tiempo de actualización [11] es esencialmente un tiempo de recuperación y depende del tipo de cambio. Lo más común es la actualización de aplicaciones y puede ser gestionada rápidamente con un área de ensayo (*staging area*) que permita la coexistencia de las versiones nueva y anterior en el nodo. Los administradores de sistemas pueden hacer un reinicio controlado, en el peor de los casos, para actualizar el nodo dentro del tiempo medio de reparación (MTTR: mean-time-to-repair). Esta área de ensayo también permite el retorno a la versión anterior en caso sea requerido. En la evolución online se utilizan tres enfoques principales:

- Reinicio rápido. El enfoque más simple es el reinicio rápido de todos los nodos a la nueva versión. El reinicio rápido es directo pero genera un tiempo de caída del servicio.
- Actualización continua. En este enfoque, actualizamos los nodos uno a la vez en un proceso que recorre todo el clúster. Esto minimiza el impacto global pues sólo un nodo está caído en un momento dado. En un sistema particionado, el retorno se reducirá durante las n ventanas de actualización, mientras que en un sistema replicado esperamos 100% de rendimiento y 100% de retorno dado que actualizamos una réplica a la vez y probablemente tenemos suficiente capacidad para prevenir la pérdida de rendimiento. La desventaja es que las versiones nueva y anterior deben ser compatibles pues deberán coexistir. Ejemplos de versiones incompatibles incluyen cambios en los esquemas, espacios de nombres o protocolos.
- Gran Tirón (big flip). Este enfoque es el más complicado y consiste en actualizar una mitad del clúster a la vez bajando y actualizando la mitad de los nodos de una sola vez. Durante el cambio, atómicamente redirigimos el tráfico a la mitad actualizada usando un switch de capa 4. Luego se actualiza la segunda mitad del clúster y se agregan al uso.

De los tres enfoques, el de Actualización continua es el que más se ajusta al modelo implementado en el sistema distribuido del presente TFM, debido a que considera un nodo como unidad de actualización y el proceso se asemeja a la actualización nodo por nodo descrito en este enfoque, desarrollándose primero a lo largo de todo el grupo de réplicas backup para terminar con la réplica primaria, por lo que se desarrollará en lo sucesivo.

8. Actualización continua: Dependencias y Requerimientos

8.1. Tipos de Dependencia

Uno de los principales problemas para la ejecución de la actualización continua es la interrelación entre los componentes del sistema. Para graficar las dependencias entre los componentes del sistema, escogeremos tres diferentes tipos de componentes de la aplicación: El código ejecutable (independiente o biblioteca), los datos y los metadatos. El código representa tanto los subsistemas y aplicaciones que se pueden inicializar independientemente, como las bibliotecas a las cuales se pueden vincular. Los datos corresponden a los datos de la aplicación almacenados en una base de datos o en un sistema de almacenamiento persistente o de tiempo de ejecución. Los metadatos corresponden a las declaraciones del esquema de la base de datos, tales como la estructura de tablas o datos y las reglas de integridad. Una versión de la aplicación está típicamente ligada a una versión de esquema y puede no trabajar apropiadamente con un cambio de esquema.

Para diagramar las dependencias entre los componentes de la aplicación se utiliza la Cadena de Dependencias de Actualización (Upgrade Food Chain: UFC).

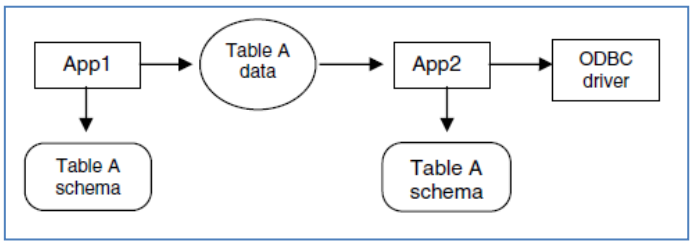


Figura 1. Ejemplo de diagrama UFC[1]

Consideremos una situación donde dos aplicaciones, App1 y App2 son actualizadas a una versión $x + 1$. App1 usa datos almacenados en una nueva tabla A, lo que requiere también una actualización del esquema de la base de datos incorporando la tabla A. Los datos de la tabla A usados por App1 son producidos por la versión actualizada de App2. Adicionalmente, App2 necesita una nueva versión de un driver ODBC para funcionar apropiadamente. Las dependencias mostradas en la figura 1 son dependencias de actualización.

8.1.1. Dependencia de función

[1] Wolski, A.(2005) *Example UFC (Upgrade Food Chain) diagram.* [Figura]. Recuperado de Wolski, A., Laiho, K. "Rolling Upgrades for Continuous Services". In: Malek, M. et al (eds) ISAS 2004, Lecture Notes in Computer Science, Vol. 3335, 175 - 189, 2005.

Un componente A se dice que es función-dependiente del componente B si requiere algunos servicios o características del componente B para funcionar apropiadamente.

Si el componente A utiliza servicios del componente B, se dice que es un consumidor de los servicios producidos por B. Las dependencias de función entre componentes son usualmente estáticas e invariantes con la versión. La razón es que, desde el momento de la generación del componente, su propósito y naturaleza implica las dependencias de funciones relacionadas. Por ejemplo, todas las aplicaciones asociadas a una base de datos son función-dependientes del esquema de la base de datos, por definición. Excepciones a esta regla pueden suceder si la funcionalidad de un componente cambia significativamente.

El conocimiento de las dependencias de funciones es una condición suficiente pero no necesaria para la ejecución de una actualización multicomponente segura. Dada una versión existente x y la versión objetivo $x + 1$, la condición necesaria es el conocimiento de la dependencia de función específica de la versión, llamada dependencia de actualización.

8.1.2. Dependencia de actualización

Asumimos que estamos actualizando los componentes A y B de la versión x a la versión $x + 1$. El componente A es actualización-dependiente del componente B si el componente A actualizado requiere de la funcionalidad o características adicionales, introducidas en la actualización de B, para funcionar apropiadamente.

Es posible ver que el propósito de la dependencia de actualización es representar las nuevas dependencias que son introducidas con una nueva versión. Si los dos componentes involucrados son versionados de diferentes modos, ambas nuevas versiones deberían ser indicadas en la dependencia. Por otro lado, si la dependencia de función de un componente con otro no ha cambiado o desaparece con una actualización dada, no es considerada una dependencia de actualización.

8.1.3. Diagrama de Cadena de Dependencias de Actualización

Es una gráfica dirigida en la cual cada nodo corresponde a una instancia de uno de los tres tipos de componentes (código, datos y metadatos), y flechas apuntando a los componentes dependientes de actualización.

Intuitivamente, los componentes deberían ser actualizados en el orden inverso a las flechas dirigidas, empezando desde los componentes más externos. Todos los componentes capturados en un diagrama UFC son considerados como parte de una suite de actualización. La actualización de los componentes en una suite de actualización tiene que estar coordinado (ordenado) de modo que los componentes puedan funcionar apropiadamente durante el proceso de actualización.

8.2. Argumentaciones sobre el sistema

8.2.1. Granularidad de la actualización

La granularidad se da e incluye los componentes y las unidades de servicio. Un componente es la entidad más pequeña reconocida por el Sistema Gestor de Disponibilidad (AMF: Availability Management Framework) y es una unidad natural de desarrollo de software. Una unidad de servicio (que está compuesta de componentes) es una unidad de redundancia, de modo que la conmutación ("switchover") se realiza a este nivel.

Debido a que ambos conceptos son irrelevantes a nivel del sistema operativo y del sistema de Alta Disponibilidad, la granularidad de actualización para ambos es el nodo (clúster).

8.2.2. Distribución

Un sistema de Alta Disponibilidad es inherentemente distribuido. Además de asumir que los componentes de una unidad están ubicados en el mismo nodo, no hacemos ninguna referencia a la naturaleza distribuida del sistema. Asumimos que las dependencias de funciones entre los componentes no dependen de si los componentes están ubicados en un nodo o no.

8.2.3. Transparencia de la actualización

Cuando la conmutación sucede (switchover), las direcciones de red (puntos de acceso de servicios) de los componentes relacionados cambian en cada conmutación. La transparencia de actualización significa que el consumidor del servicio, que no es dependiente de la actualización, no debería verse afectado por la actualización. Si la actualización se basa en la conmutación de unidades (switchover), el modo de lograr la transparencia de la actualización es el mismo que el realizado durante las fallas en un sistema Altamente Disponible.

8.3. Actualización Trivial: Componente Independiente

Si la actualización de un componente no es dependiente de la actualización de algún otro componente, puede ser actualizado por sí mismo, debido a que su suite de actualización no incluye otros componentes.

Para actualizar un componente independiente [12], se puede aplicar una conmutación simple. Sean App_a^n y App_s^n instancias de la aplicación de versión *n* corriendo como componentes en las unidades activa y en espera, respectivamente. Para actualizar la componente independiente App de la versión *x* a la versión *x + 1*:

1. Detener el componente App_s^x en la unidad en espera.
2. Instalar la nueva versión del componente en la unidad en espera.
3. Reiniciar el componente como App_s^{x+1}.
4. Desarrollar una conmutación controlada de las unidades (App_a^x debe convertirse en App_s^x)
5. Detener App_s^x en la nueva unidad en espera.
6. Instalar la nueva versión del componente en la unidad en espera.

7. Reiniciar el componente como App_s^{x+1}.
8. De manera opcional, desarrollar una conmutación más si la asignación original de unidades activa y en espera era preferible.

Después de realizar el paso 4, las instancias App_a^{x+1} y App_s^x tienen que interactuar como un par relacionado. Si la operación activa/en espera a nivel del componente involucra comunicaciones entre la componente activa y en espera, debe tenerse en cuenta de la necesidad de comunicación entre la nueva versión App_a^{x+1} y la versión anterior App_s^x, y posiblemente en sentido contrario. Si no hay comunicación entre nodos, por ejemplo, si las componentes intercambian datos vía una base de datos, este punto es irrelevante.

Nótese que entre los pasos del 2 al 7 el sistema es vulnerable pues está corriendo en modo autónomo (stand-alone mode): no hay una componente en espera disponible que pueda asumir las funciones de una componente activa fallida. Por esta razón, se deben tomar precauciones adicionales si el periodo entre los pasos del 2 al 7 es prolongado. Típicamente, se utiliza una unidad de repuesto (spare unit) para realizar la instalación si ésta requiere mayor tiempo. Las unidades de repuesto son aquellas que no tiene un rol de activo o en espera asignado.

9. Escenarios de actualización

9.1. Actualización del Sistema Operativo

Si bien el sistema operativo es independiente del software de Alta Disponibilidad, la capacidad de realización de conmutaciones puede ser utilizada en la actualización del sistema operativo. Debido a que la instalación de una nueva versión del sistema operativo puede ser una operación que consuma mucho tiempo, se deberían usar nodos de repuesto para realizar la instalación en segundo plano, sin poner en peligro la disponibilidad de los servicios. Luego que la unidad de repuesto está actualizada, el nodo en espera puede detenerse y rápidamente reemplazarse con la unidad de repuesto, reduciendo el tiempo de vulnerabilidad del sistema.

Un escenario de actualización del sistema operativo es el siguiente:

1. Instalar el sistema operativo en una unidad de repuesto.
2. Instalar el sistema de Alta Disponibilidad, el Gestor de Base de Datos (DBMS) y las aplicaciones si fueran necesarias.
3. Desconectar el nodo en espera (esto es, el nodo que contiene las unidades en espera) del nodo activo (resultando en una operación autónoma temporal).
4. Transferir la base de datos del nodo en espera al nodo de repuesto actualizado.
5. Asignar el rol de nuevo nodo en espera al nodo de repuesto. El anterior nodo en espera se vuelve un nodo de repuesto.
6. El sistema inicializa los componentes y las operaciones activa/en espera son retomadas. Las bases de datos activa y en espera se reconectan y resincronizan.
7. Desarrollar una conmutación controlada.
8. Repetir los pasos del 1 al 7 empezando con el nuevo nodo de repuesto y el nuevo nodo en espera.

El escenario descrito debe repetirse para todos los pares, en un sistema $2N + M$ redundante, donde M es el número de nodos de repuesto. Si no hay nodos de repuesto en un sistema, los periodos de operación autónoma serán largos.

9.2. Actualización de un sistema Altamente Disponible

Un sistema Altamente Disponible tiene instancias interconectadas corriendo en cada nodo. Las actualizaciones de sistemas Altamente Disponibles pueden ser dependientes de las actualizaciones de los esquemas y de nuevos archivos de configuración si existen. Otra dificultad es que los componentes son dependientes del software del sistema dado que están típicamente ligados a las librerías del sistema.

Como la revinculación de las aplicaciones puede tomar una considerable cantidad de tiempo, el uso de nodos de repuesto es preferible como en el caso anterior.
Un escenario de actualización del sistema Altamente Disponible es el siguiente:

1. Desarrollar la actualización del esquema del modelo de la base de datos para soportar la actualización del sistema Altamente Disponible, si es aplicable.
2. Actualizar el sistema Altamente Disponible en el nodo de repuesto.
3. Revincular otros subsistemas y aplicaciones con las bibliotecas del sistema actualizado, en el nodo de repuesto.
4. Desconectar el nodo en espera actual del nodo activo, resultando en una operación autónoma temporal.
5. Transferir la base de datos del nodo en espera al nodo de repuesto actualizado, si es aplicable.
6. Asignar al nodo de repuesto el rol de nuevo nodo en espera. El viejo nodo en espera se vuelve un nodo de repuesto.
7. Desarrollar una conmutación controlada.
8. Repetir los pasos del 2 al 7 empezando con el nuevo nodo de repuesto y el nuevo nodo en espera.

El escenario descrito debe repetirse para todos los pares, en un sistema $2N + M$ redundante, donde M es el número de nodos de repuesto. Con el fin de que el escenario descrito suceda, la actualización del sistema Altamente Disponible tiene que ser diseñada de modo que la vieja y la nueva versión del sistema puedan coexistir. Si esto no sucede, no será posible la actualización continua del sistema Altamente Disponible, requiriendo el apagado de todo el sistema.

10. Actualización Automática: Modelo de Ajmani

El principal objetivo es crear un sistema de actualización automática flexible y genérico que permita a los sistemas proveer los servicios durante la actualización, por lo que debe cumplir una serie de requerimientos.

1. **Simplicidad**. El modelo de actualización debe ser fácil de usar. En particular, se necesita *modularidad*: para definir una nueva actualización, el actualizador sólo debería necesitar comprender la relación entre la versión actual del sistema y la nueva.

2. **Generalidad**. El modelo de actualización debe permitir el cambio de la aplicación del sistema de modos arbitrarios. Tiene dos partes:
    a. **Incompatibilidad**. La nueva versión debe poder ser incompatible con la anterior, esto es, puede finalizar el soporte de funciones actuales y puede cambiar los protocolos de comunicación. Esto es importante pues de otro modo las versiones posteriores del sistema deben continuar con el soporte de funciones previas, lo cual complica la aplicación y la hace menos robusta.
    b. **Persistencia**. Los sistemas de interés tienen estados persistentes valorables que deben sobrevivir a las actualizaciones. Por lo tanto, las actualizaciones deben permitir la preservación y transformación del estado persistente. Esto puede ser costoso, pues cada nodo puede tener estados muy grandes y la transformación de este estado a la representación requerida por el nuevo software puede significar la lectura y escritura de todo el estado (por ejemplo, añadir una nueva propiedad a cada archivo en un sistema de archivos).

       No es explícitamente un requerimiento que las actualizaciones preserven los estados volátiles. Las actualizaciones no son muy frecuentes, dado que las organizaciones no desean liberar un nuevo software hasta que, idealmente, esté libre de errores. Por lo tanto, es aceptable perder los estados volátiles en tales eventos infrecuentes, pero no es aceptable perder estados persistentes, dado que puede no haber manera de recuperarlos. Por ejemplo, es aceptable perder conexiones abiertas y operaciones de escritura en archivos del sistema, pues esas conexiones y operaciones podrán repetirse, pero no es aceptable perder los archivos en sí.
3. **Despliegue Automático**. Los sistemas de interés son demasiado grandes como para actualizarlos manualmente. Por lo tanto, las actualizaciones deben ser desplegadas automáticamente: el actualizador define una actualización en una ubicación central y el sistema de actualización propaga la actualización y la instala en cada nodo.
4. **Despliegue Controlado**. El actualizador debe ser capaz de controlar cuando los nodos se actualizan con la misma precisión como si lo hiciera manualmente. Hay muchas razones para el despliegue controlado, incluyendo: permitir que el sistema siga brindando servicio mientras la actualización sucede, actualizando réplicas uno a la vez en un sistema replicado (esto es importante cuando la actualización involucra transformaciones de estado que consumen tiempo), probando la actualización en pocos nodos antes de realizarlo en todo el sistema, y programando las actualizaciones para los momentos en que la carga en los nodos sea ligera.
5. **Modo de Operación Mixta**. El despliegue controlado implica que las actualizaciones sean asíncronas, esto es, los nodos pueden actualizarse independientemente y en cualquier momento. Esto significa que puede haber largos periodos de tiempo en los cuales el sistema corra de manera mixta, es decir, donde algunos nodos estén actualizados y otros no. No obstante, el sistema debe proveer el servicio, aun cuando la actualización sea incompatible. Esto implica que el sistema de actualización debe proveer un modo de interoperación entre los nodos con diferentes versiones (sin restringir los tipos de cambios que una actualización pueda hacer)

10.1. Modelo del Sistema

El sistema distribuido es modelado como una colección de objetos que se comunican vía llamada a métodos. Un objeto tiene una identidad, un tipo y un estado. Un *tipo* identifica el comportamiento de todos los objetos para ese tipo. Una *especificación* describe dicho comportamiento de manera informal o con cierta notación matemática. Una especificación define un estado abstracto para los objetos del tipo y define cómo cada método del tipo interactúa con dicho estado (en términos de precondiciones y postcondiciones para los métodos). Un objeto es una instancia de una clase que define cómo el objeto implementa su tipo.

Dado que los nodos y la red pueden fallar, los objetos están preparados para las fallas de cualquier llamada a método remota. Los sistemas basados en llamadas a procedimientos remotos o invocación de métodos remotos corresponden a este modelo.

Una porción del estado de un objeto puede ser persistente, esto es, puede residir en disco o en otros nodos. Los objetos están preparados para las fallas de sus nodos, las cuales pueden ocurrir en cualquier momento de su procesamiento. Cuando el nodo se recupera, el objeto se reinicializa a partir de la porción persistente de su estado.

El modelo permite múltiples objetos por nodo, pero para simplificar, se asume un solo objeto por nodo. Así, cada nodo corre una clase superior, esto es, la clase que implementa su objeto. Cuando hay muchos objetos por nodo, cada clase del objeto puede actualizarse independientemente.

Se asume que las definiciones de las clases están almacenadas en repositorios confiables y definen totalmente la implementación de un objeto, incluyendo sus subcomponentes y bibliotecas.

10.2. Modelo de Actualización

El *esquema* de un sistema es un conjunto de clases del tipo correcto para los nodos del sistema, esto es, cada clase en un esquema se basa sólo en los tipos de otras clases en dicho esquema. Una actualización define un nuevo esquema y un mapeo entre el viejo y el nuevo esquema de clases. Algunas clases viejas son directamente incluidas en el nuevo esquema, pero otras son reemplazadas por nuevas clases. Este reemplazo corresponde a una *actualización de clase*.

Asociamos un número de versión con cada esquema. El esquema inicial tiene el número de versión 1. Cada subsecuente esquema tiene el siguiente número. Así, una actualización mueve el sistema de una versión a la siguiente.

Una actualización de clase define cómo reemplazar las instancias de una clase vieja con instancias de una nueva clase. Tiene seis componentes identificadas como $\langle oldClassID, newClassID, TF, SF, pastSO, futureSO \rangle$:

- ***oldClassID*** identifica la clase que es obsoleta.
- ***newClassID*** identifica la clase que va a reemplazar la clase vieja.
- ***TF*** identifica una *función de transformación* que genera un estado persistente inicial para el nuevo objeto a partir del estado persistente del viejo objeto.

- ***SF*** identifica una *función de programación* que dice a un nodo cuándo debería actualizarse.
- ***pastSO*** y ***futureSO*** identifican clases para *objetos de simulación* que permiten a los nodos operar entre versiones. Un objeto *futureSO* permite a un nodo soportar el comportamiento de la nueva clase antes que sea actualizado. Un objeto *pastSO* permite a un nodo soportar el comportamiento de la clase vieja tras ser actualizado.

Este diseño permite a los nodos actualizados interoperar con los nodos no actualizados. De hecho, una serie de objetos de simulación pueden permitir a los nodos separados por varias versiones interactuar, lo cual es importante cuando suceden actualizaciones de manera lenta o cuando los nodos pueden estar desconectados por largos periodos. El diseño es modular: definir los componentes de una actualización de clase requiere una comprensión de sólo las viejas y nuevas clases, sin importar cuántas versiones existan y cuántas versiones separen a los nodos en comunicación.

Aunque cada actualización de clase tiene seis componentes, muchos de estos pueden ser omitidos para la mayoría de actualizaciones. Una función de transformación sólo es necesaria cuando la actualización cambia el modo en que un objeto organiza su estado persistente. Los objetos de simulación sólo son necesarios cuando la actualización cambia un tipo del objeto, por lo que, dependiendo del tipo de cambio, la actualización puede omitir el *pastSO* o el *futureSO* o ambos. Las funciones de programación no pueden ser omitidas pero son simples de implementar.

10.3. Proceso de Actualización

El sistema de actualización consiste en un servidor de actualización, una base de datos de actualización y capas de actualización en los nodos. El *servidor de actualización* provee un repositorio central de información sobre las actualizaciones, y la *base de datos de actualización* (*UDB* - upgrade database) provee un almacén central de información sobre el estatus de actualización de los nodos. Cada nodo ejecuta una *capa de actualización* (*UL* -upgrade layer) que instala las actualizaciones y maneja las llamadas entre versiones y también mantiene una base de datos local en la cual almacena información sobre el estatus de actualización de los nodos con los cuales se ha comunicado recientemente.

La figura 2 muestra la estructura de un nodo. La versión actual del nodo identifica la actualización más recientemente instalada (o la versión inicial cuando todavía no haya habido actualizaciones); el objeto actual del nodo es una instancia de su clase actual, que es la nueva clase de esta actualización. El nodo puede también estar ejecutando objetos de simulación: *futureSOs* para simular versiones aún no instaladas en el nodo y *pastSOs* para simular versiones que son anteriores a la versión actual.

Estos objetos de simulación son implementados utilizando *delegación*: ellos llaman a métodos de la versión siguiente o previa, que pueden estar en el objeto actual o en otro objeto de simulación. La capa de actualización del nodo etiqueta las llamadas salientes con el número de versión: las llamadas hechas por el objeto actual son etiquetadas con la versión actual del objeto, mientras que las llamadas realizadas por los objetos de simulación son etiquetadas con sus respectivas versiones. La capa de actualización

distribuye las llamadas entrantes observando sus versiones y enviándolas al objeto local que maneja dichas versiones.

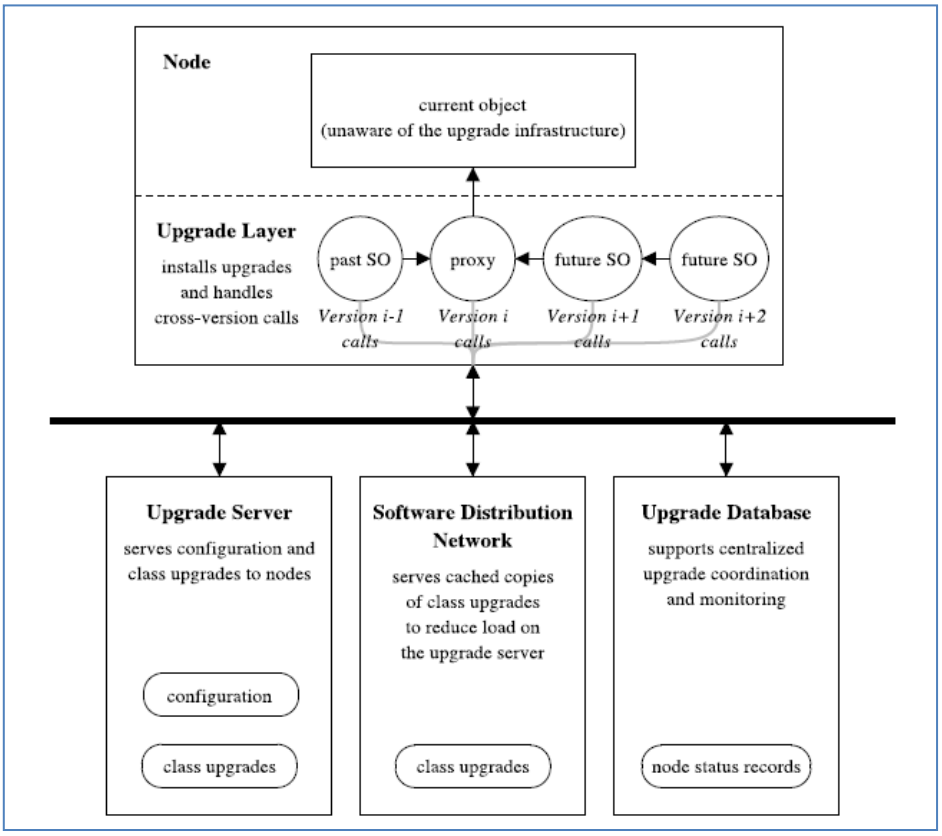


Figura 2. El sistema de actualización. Los componentes se comunican sobre una red. Las flechas indican la dirección de las llamadas a métodos. Los nodos corren una instancia de su clase de versión *i* y objetos de simulación de versiones *i-1*, *i+1*, *i+2*.[2]

Los nodos aprenden sobre las actualizaciones debido a que reciben una llamada de un nodo con una versión posterior, por medio de comunicaciones periódicas con el servidor de actualización, o vía chismorreo: los nodos chismean con otros periódicamente sobre las más recientes versiones y sus propios estados, esto es, sus números de versión actual y clases.

Cuando la capa de actualización aprende sobre una versión más reciente, se comunica con el servidor de actualización para descargar una pequeña descripción de la actualización. Luego chequea los posibles efectos, por ejemplo, si la actualización contiene una clase vieja que se esté ejecutando en el nodo (un nodo puede estar varias versiones atrás, pero puede procesar las actualizaciones una por una). Si el nodo es afectado, la capa de actualización extrae los componentes de la actualización de la clase que requiere, elimina cualquier RPC en ejecución y luego inicia un objeto de simulación *futureSO* si es necesario, esto es, si el nuevo tipo es un subtipo del anterior, o si la actualización es incompatible.

[2] Ajmani, S.(2004) *The upgrade infrastructure...* [Figura]. Recuperado de Ajmani, S. "Automatic Software Upgrades for Distributed Systems". PhD Thesis, Department of Electrical Engineering and Computer Science, Massachusets Institute of Technology, 2004

Luego, la capa de actualización invoca a la función de programación de actualización de la clase, la cual se ejecuta en paralelo con los otros procesos del nodo. La función de programación notifica a la capa de actualización cuando sea el momento de actualizar.

Para actualizar, la capa de actualización reinicia el nodo y ejecuta la función de transformación para convertir el estado persistente del nodo a la representación requerida por la nueva clase. Después de esto, la capa de actualización realiza la recuperación normal del nodo, durante el cual crea el objeto actual y los objetos de simulación. Debido a que los objetos de simulación delegan hacia el objeto actual, la capa de actualización debe crearlos en un orden que lo permita. Primero, crea el objeto actual, el cual se recupera a partir del estado persistente recientemente transformado. Luego crea cualquier objeto de simulación que se requiera, en el orden de sus distancias al objeto actual. Finalmente, la capa de actualización notifica a la base de datos de actualización que su nodo está ejecutando una nueva versión.

Cuando todos los nodos han migrado hacia una nueva versión, las versiones previas pueden ser retiradas. La información sobre el retiro llega mediante mensajes desde el servidor de actualización. En respuesta, la capa de actualización descarta los objetos de simulación con las versiones en retiro (*pastSOs*). Esto puede llevarse a cabo de manera lenta, dado que mantener objetos de simulación pasados no afecta el comportamiento o el performance de versiones posteriores.

11. Conclusiones

El enfoque con mayor cercanía al modelo implementado en el sistema distribuido del presente TFM es la Actualización continua, cuya granularidad de actualización es el nodo y el proceso de actualización es nodo a nodo a lo largo de todos los nodos del clúster. Los diferentes tipos de componentes de actualización pueden ser el código ejecutable, los datos y los metadatos. Las dependencias de actualización entre los diferentes elementos del sistema se pueden mostrar en una Cadena de Dependencias de Actualización, los cuales forman parte de una suite de actualización, en la cual el proceso debe estar suficientemente coordinado como para permitir el adecuado funcionamiento de los componentes durante el proceso de actualización. Los escenarios de actualización en los cuales se aplica implican la interacción entre los nodos activo, en espera y de repuesto, repitiéndose el proceso según el número de nodos del sistema. Ajmani describe los diferentes requerimientos que cubre su modelo de actualización automática, entre los cuales están la simplicidad, generalidad, despliegue automático y controlado, así como la capacidad de operación mixta.

El sistema de actualización que modela consiste en un servidor de actualización, una base de datos y capas de actualización en los nodos. Durante el proceso de actualización se generan objetos de simulación implementados por delegación, los cuales permiten la interacción con objetos de versiones diferentes. Las funciones de transformación ejecutan la actualización y las de programación indican cuándo realizar las actualizaciones.

Capítulo IV: Modelo de Actualización de un Sistema Primario - Backup

12. Modelo Primario - Backup

Un modo de implementar un servicio tolerante a defectos es usando múltiples servidores que fallan independientemente. El estado del servicio es replicado y distribuido entre los servidores, y las actualizaciones son coordinadas de modo que, aunque un grupo de servidores falle, el servicio permanece disponible. Un modo de realizar esto es designando un servidor como el primario y los otros como los backups. Los clientes hacen requerimientos enviando mensajes solo al primario. Si el primario falla, entonces la recuperación (failover) ocurre y uno de los backups toma el control. Esta arquitectura de servicio es comúnmente llamada *primario-backup* [20]. El objetivo es brindar a los clientes la ilusión de que el servicio es implementado por un solo servidor.

Se consideran tres métricas para cualquier protocolo primario-backup [21]:

- ***Grado de replicación***: el número de servidores usados para implementar el servicio.
- ***Tiempo de bloqueo***: el peor periodo entre un requerimiento y su respuesta en cualquier ejecución libre de recuperación.
- ***Tiempo de recuperación***: el peor periodo durante el cual los requerimientos pueden perderse debido a que no hay primario.

La pregunta fundamental es, dado que no más de **f** componentes pueden fallar, cuáles son los menores valores posibles para el grado de replicación, el tiempo de bloqueo y el tiempo de recuperación. Este estudio fue realizado por Budhiraja et al [21] y será expuesto en lo sucesivo.

12.1. Especificaciones del primario - backup

Informalmente, un servicio implementado usando el modelo primario-backup consiste de un conjunto de servidores, de los cuales no más que uno es el primario en cualquier momento. Un cliente envía un requerimiento al servidor que considera es el primario. Un cliente aprende del servicio cuando el primario cambia y redirige sus posteriores requerimientos. Asumimos que un cliente puede enviar cualquier requerimiento en cualquier momento.

Un protocolo primario-backup debe poseer cuatro propiedades:

Pb1: Existe un predicado local *Prmy_s* en el estado de cada servidor *s*. En cualquier momento, hay a lo más un servidor *s* cuyo estado satisface *Prmy_s*.

Podemos definir el tiempo de recuperación de un servicio primario-backup como el periodo más largo de tiempo durante el cual *Prmy_s* no es verdadero para cualquier *s*.

Pb2: Cada cliente *i* mantiene la identidad de un servidor *Dest_i* de modo que para hacer un requerimiento, el cliente *i* envía un mensaje a *Dest_i*.

Asumimos que los requerimientos enviados al servidor *s* son encolados en una cola de mensajes en *s*.

Pb3: Si un requerimiento llega a un servidor que no es el primario, entonces el requerimiento no es encolado ni procesado.

Por simplicidad, asumimos que cada requerimiento requiere una respuesta. Se define una interrupción del servidor que ocurre en el instante *t* en este servicio si algún cliente correcto envía un requerimiento en el instante *t* al servicio pero no recibe una respuesta. Este servidor es llamado ***(k,Δ)-bofo server*** (interrupciones acotadas, finitamente frecuentes) si todas las interrupciones del servidor pueden ser agrupadas en a lo más ***k*** intervalos de tiempo, teniendo cada intervalo una longitud de a lo más Δ. Aunque algunos requerimientos hechos a este servidor puedan perderse, el número de éstos es acotado.

Pb4: Existen valores fijos de ***k*** y Δ tales que el servicio se comporta como un único servidor ***(k, Δ)-bofo server***.

12.2. Protocolo simple primario - backup

Asumamos que todas las comunicaciones se realizan con conexiones sin defectos punto a punto y que cada conexión tiene una cota superior δ en el tiempo de envío de mensajes. Existe un servidor primario *p1* y un servidor backup *p2* conectados por una red de comunicaciones. Un cliente c inicialmente envía un requerimiento a *p1*. Cuando *p1* recibe el requerimiento, este:

- Procesa el requerimiento y actualiza su estado.
- Envía información sobre la actualización a *p2*. Este mensaje se llama mensaje de actualización de estado.
- Sin esperar un **ACK** desde *p2*, envía una respuesta al cliente.

El servidor *p2* actualiza su estado luego de recibir el mensaje de actualización de estado de *p1*. Además, *p1* envía mensajes simples a *p2* cada τ segundos. Si *p2* no recibe dicho mensaje en $\tau + \delta$ segundos, entonces *p2* se convierte en primario. Una vez que *p2* es primario, informa a los clientes y empieza a procesar los subsiguientes requerimientos de los clientes.

La propiedad **Pb1** requiere que nunca existan dos primarios. Esto se satisface mediante las siguientes definiciones de *Prmy*:

$$Prmy_{p1} \stackrel{\text{def}}{=} p1 \; no \; ha \; caído$$
$$Prmy_{p2} \stackrel{\text{def}}{=} p2 \; no \; ha \; recibido \; un \; mensaje \; de \; p1 \; en \; \tau + \delta$$

La afirmación $Prmy_{p1} \wedge Prmy_{p2}$ es siempre falsa en un sistema con este protocolo, de ahí que **Pb1** se cumple. El tiempo de recuperación es el intervalo durante el cual se cumple $\neg Prmy_{p1} \wedge \neg Prmy_{p2}$. En este protocolo, este intervalo ocurre cuando *p1* cae inmediatamente después de enviar un mensajes a *p2*, el cual toma δ segundos en llegar, por lo que el tiempo de recuperación es de $\tau + 2\delta$ segundos.

12.2.1. Modelo del sistema

Consideremos un sistema de ***n*** servidores y un conjunto de clientes, donde los relojes de los servidores están sincronizados de manera cercana al tiempo real. Asumimos que los clientes y los servidores se comunican por intercambio de mensajes por medio de una red punto a punto completamente conectada y que

hay una conexión **FIFO** entre cada par de procesos. Los mensajes son encolados en una cola mantenida por el proceso receptor y accede a esta cola ejecutando la sentencia *receive*. Además, asumimos que hay una constante conocida δ de modo que si dos procesos *pi* y *pj* están conectados por una conexión sin defectos, entonces un mensaje enviado de *pi* a *pj* en el tiempo *t* será encolado en la cola de pj a lo más en el tiempo $t + \delta$.

La ejecución de un sistema es modelada por una secuencia de eventos que involucra clientes, servidores y colas de mensajes. Los eventos incluyen: envío del mensaje, encolamiento del mensaje, recepción del mensaje, y cálculos en un proceso.

Asumiendo que los servidores son deterministas y que las fallas del servidor y de la conexión ocurren independientemente, se consideran los siguientes modelos de fallas:

- *Falla por caída*: Un servidor puede fallar deteniéndose prematuramente. Hasta que se detiene, el servidor se comporta correctamente. Luego de detenido, nunca se recupera.
- *Fallas por caída y conexión*: Un servidor puede caer o una conexión puede perder los mensajes (pero la conexión no retrasa, duplica o corrompe los mensajes).
- *Fallas por omisión de recepción*: Un servidor puede fallar, no sólo por caída sino también omitiendo la recepción de algunos mensajes en una conexión sin defectos.
- *Fallas por omisión de envío*: Un servidor puede fallar no sólo por caída sino también omitiendo el envío de algunos mensajes en una conexión sin defectos.
- *Fallas por omisión general*: Un servidor puede mostrar fallas por omisión de envío y recepción de mensajes.
- *Fallo-Parada*: Se distingue del modelo de caída por asumir un sistema síncrono, de modo que cuando hay un fallo de parada todos los demás procesos podrán detectar dicho fallo [26].
- *Fallos arbitrarios*: Los procesos que incurren en una situación de fallo dejan de comportarse según dicte su especificación. Es decir, a partir de ese momento podrán comportarse de manera arbitraria y debido a ello su ejecución y sus efectos serán impredecibles [27].

Un protocolo tolera **f** fallas en un modelo dado si trabaja correctamente a pesar del comportamiento defectuoso de hasta **f** componentes.

12.2.2. Protocolo de Alsberg y Day

En este protocolo [20], el cliente envía un requerimiento al servicio y luego se bloquea esperando la respuesta del servicio o el *timeout*.

- Si el requerimiento llega al primario, este procesa la operación y genera la modificación pertinente, envía un mensaje de modificación de estado al backup y se bloquea. El backup, tras recibir el mensaje, actualiza su estado, envía la respuesta al cliente y envía un **ACK** al primario diciendo que ha completado la operación. Luego de recibir el **ACK**, el primario se desbloquea y procesa el siguiente requerimiento.
- Si el requerimiento llega al backup, el backup envía el requerimiento al primario. El primario, luego de recibir el mensaje, realiza la modificación,

envía la respuesta al cliente y luego envía el mensaje de actualización de estado al backup, el cual modifica su estado y descarta el requerimiento.

Las caídas son detectadas por pérdida de los mensajes **ACK** y por el envío de mensajes periódicos "Estás vivo?". Si el primario falla, el backup se convierte en el nuevo primario. Y cuando el primario no tiene backup (sea que el backup haya caído o que se haya convertido en primario), el primario usa algún otro protocolo para reclutar algún otro servidor y que se convierta en backup.

12.2.3. Protocolo Tandem

Este protocolo [22] está diseñado para tolerar una falla simple por caída y conexión. Consiste de múltiples nodos conectados por una red de conexión. Cada nodo consiste de múltiples procesadores y controladores de entrada/salida interconectados por buses redundantes. Cada procesador en el nodo puede soportar procesos concurrentes.

Los procesos se hacen tolerantes a defectos usando pares de procesos. Los pares de procesos son implementados replicando cada proceso en dos diferentes procesadores en el nodo, siendo un proceso el primario y el otro el backup.

13. Modelo de Actualización de Solarski

El objetivo de este algoritmo de actualización [10] es actualizar un servidor replicado activamente, esto es, sustituir el código ejecutable de los procesos que corren en los nodos por otra versión de dicho código, considerando las siguientes afirmaciones:

- **R1**. El algoritmo es disparado bajo demanda por medio de un mensaje de actualización usando los mecanismos básicos del sistema de comunicaciones.
- **R2**. El algoritmo se autoestabiliza, esto es, puede automáticamente recuperarse luego de la ocurrencia de fallas.
- **R3**. El algoritmo preserva la consistencia del sistema mediante la transferencia de estados desde la versión antigua a la nueva en caso de servidores con estado.

La actualización de un sistema replicado puede incluir los siguientes pasos:

1) Se inicia una réplica con el nuevo código en el modo backup *B(n+1)*
2) Para todas las réplicas backup, se realiza una actualización de cada réplica *Bi*, donde $0 < i < n$:
    a. Se inicia la réplica *Bi** con el nuevo código y se une al grupo en modo pasivo.
    b. Se remueve la vieja réplica *Bi*.
3) Se fuerza un proceso de falla para activar una de las réplicas backup y el apagado del primario antiguo.

La siguiente figura muestra esta versión del algoritmo de actualización:

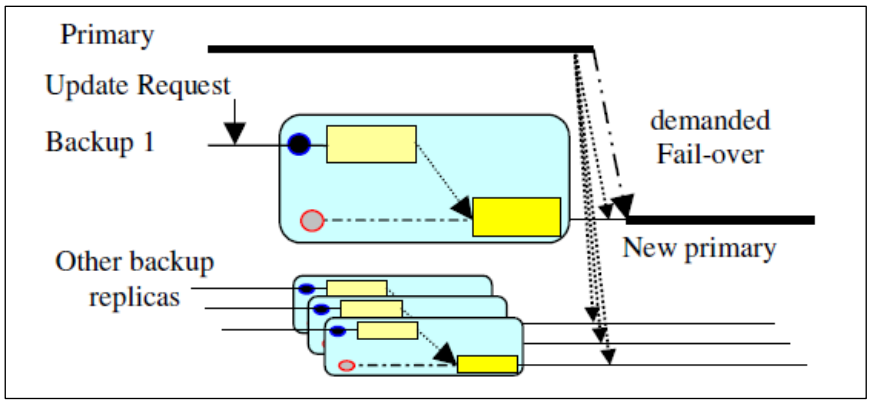


Figura 3. Proceso de actualización de un componente replicado pasivamente[3]

14. Conclusiones

El modelo de replicado seguido en el presente TFM es el modelo primario - backup en el cual los clientes se unen a la réplica primaria a quien envían sus requerimientos, y sólo la réplica primaria los procesa y responde. A la par, existe un grupo de réplicas backup que no procesan los requerimientos pero sí reciben de la réplica primaria la actualización de las variables de estado, de modo que si en algún momento la réplica primaria falla, una de las réplicas backup toma el lugar de la réplica primaria.

Solarski plantea un método de actualización para un sistema replicado siguiendo el modelo primario - backup, en el cual se reemplazan las réplicas una a una, de modo que se inicia la réplica con el nuevo código, se une al grupo, recibe los valores de las variables de estado de una réplica vieja correspondiente y luego ésta es removida del grupo, forzando al finalizar una falla de la réplica primaria para apagarla y reemplazarla por una réplica backup actualizada.

El modelo de Solarski considera una secuencialidad en el proceso de adición de réplica nueva, transferencia de información y eliminación de réplica vieja, por lo que se plantea un modelo que separe los procesos de adición y eliminación en procesos concurrentes y regulados tanto por pasos de mensajes como por listado de réplicas actualizadas y viejas.

[3] Solarsky, M.(2004) *Upgrade process of a passively replicated component.* [Figura]. Recuperado de Solarski, M. "Dynamic Upgrade of Distributed Software Components". PhD Thesis, Faculty of Electronic and Informatic, Technical University of Berlin, Berlin, 2004.

Capítulo V: Sistema de Actualización Concurrente

15. Modelo de actualización desacoplado

El modelo del sistema distribuido utilizado corresponde a un sistema primario-backup cuyos nodos están unidos por un sistema de conexión fiable, de modo que no se generen particiones de red por problemas en la transmisión de los mensajes. Los nodos pueden encontrarse en diferentes servidores, en un mismo servidor particionado virtualmente o en un sistema hipervisor que encapsule los recursos de hardware y brinde una capa de virtualización de máquinas virtuales.

La comunicación de los mensajes del servicio entre el cliente y el primario es asíncrona o no bloqueante, mientras que la comunicación entre los procesos de gestión entre nodos es parcialmente síncrona, estableciendo timeouts para la gestión de los mensajes ACK que permiten establecer la viveza de los nodos y la pertenencia al sistema. Así también, se establecen mecanismos de pasos de mensajes con timeouts para el proceso de actualización de las variables de estado de los nodos backup, para la validación de la versión de los nodos backup y la notificación de elección de primario luego de finalizado el proceso de reemplazo de los nodos antiguos durante el proceso de actualización dinámica.

El modelo de fallos considerado para los nodos es el de fallos parada, puesto que los nodos se comportan correctamente hasta el momento en que paran y dejan de funcionar sin volver a recuperarse. La detección de los fallos por parte de los procesos se realiza mediante timeouts, por lo que el sistema es parcialmente síncrono. Periódicamente, los nodos backup envían un mensaje de ACK (HEART-BEATS) al primario. En un hilo (thread) de ejecución, el primario recibe estos mensajes y responde a cada nodo backup con un mensaje PRYALIVE. Si los backups observasen que no reciben respuesta a sus HEART-BEATS, inician el proceso de elección del nuevo primario que corresponde al backup de menor identificador.

El proceso de actualización se basa en la comparación del orden de la versión entre las réplicas que existen en el sistema con las que se agregan, empezando por las réplicas backup y finalizando con la réplica primaria. Cuando una réplica backup se agrega al sistema, envía un mensaje a la réplica primaria con el número de versión que posee. Cuando la réplica primaria recibe el mensaje, compara la versión de la nueva réplica backup con el número de versión que posee. Si el número de versión es mayor, el primario cambia su estado por "*Upgrading*" y envía un mensaje "**BCKDIE**" a una de las réplicas backup existentes previamente la cual cambia su estado a "*Closing*", lo que genera la finalización de dicha réplica backup.Este proceso se realiza con todas las réplicas backup de versión igual a la de la réplica primaria. Una vez eliminadas las réplicas backup existentes previamente, al agregarse una nueva réplica backup de versión mayor, la réplica primaria envía un mensaje "**BCKCHGPRY**" a todas las réplicas backup nuevas para que inicien el proceso de elección de una nueva réplica primaria según la lógica propia del sistema distribuido, y luego cambia su estado por "*Closing*", generando su apagado.

Si bien el proceso de actualización inicia cuando el sistema detecta que se ha añadido una réplica de versión mayor a la versión de la réplica primaria, a partir de este momento a diferencia del modelo de Solarski, el proceso de adición y eliminación de réplicas backup de diferentes versiones no es secuencial y no está ligado uno a uno, sino que están gestionados por hilos diferentes, de modo que a medida que se van añadiendo réplicas backup actualizadas por el hilo ***Primary-Backup Thread***, la réplica primaria revisa el número de réplicas de versión anterior registradas y se van eliminando réplicas backup de versión anterior por el hilo ***Upgrade Thread***, hasta que no quede ninguna réplica backup de versión anterior registrada, finalizando con el apagado de la réplica primaria y la elección de una nueva réplica primaria. Este mecanismo agrega un mayor grado de concurrencia al proceso de actualización dinámica. Durante todo este proceso, el número de réplicas backup definido para el sistema distribuido asegura que siempre existirán el número de réplicas vivas necesarias para soportar el máximo número de fallos simultáneos.

El proceso de adición de réplicas no distingue entre aquellas con una u otra versión, de modo que la réplica primaria agregará aquellas cuyo **ID** se encuentre en el listado de réplicas backup vivas que internamente va generando y manteniendo basándose en el listado de **IDs** de su archivo de configuración. El archivo de configuración de cualquier réplica es un **XML** en el cual el parámetro "*id*" contiene el listado de **IDs** de las réplicas que participarán del sistema tanto en su estado estándar o "*Current*", como en su estado de actualización o "*Upgrading*". Para un sistema de ***n*** réplicas (**1** primario y ***n-1*** backups), el listado tiene la siguiente estructura:

- ***n*** IDs que corresponden al sistema que está siendo actualizado por ***n*** réplicas backup.
- ***n*** IDs que corresponden a los IDs de las ***n*** réplicas que actualizan el grupo anteriormente indicado.
- ***n*** IDs que corresponden a las réplicas que posteriormente actualizarán al grupo que al momento están actualizando el sistema.

Para el grupo inicial de réplicas backup que iniciarán el sistema distribuido, el listado constará de ***2n*** elementos (Grupo 0), mientras que para los posteriores grupos el listado de cada réplica constará de ***3n*** elementos (Grupos 1 y 2), tal como se puede ver en el ejemplo (Cuadro 1).

	ID's de nodos	Lista ID's Actual	Lista ID's Upgrade I	Lista ID's Upgrade II	Lista de ID's en XML
Grupo 0	24, 25, 26	---	24, 25, 26	27, 28, 29	24, 25, 26, 27, 28, 29
Grupo 1	27, 28, 29	24, 25, 26	27, 28, 29	30, 31, 32	24, 25, 26, 27, 28, 29, 30, 31, 32
Grupo 2	30, 31, 32	27, 28, 29	30, 31, 32	33, 34, 35	27, 28, 29, 30, 31, 32, 33, 34, 35

Cuadro 1. Ejemplo de listado de IDs para diferentes grupos de réplicas.

Durante el proceso de mantenimiento del listado de réplicas backup vivas, el primario recorre el listado de IDs y ejecuta un proceso ***receive*** con **TIMEOUT** por cada **ID** para obtener los mensajes **ACK** enviados por las réplicas backup a la cola correspondiente. Si obtiene el **ACK**, agrega el **ID** de la réplica correspondiente a la lista de IDs de réplicas vivas. Caso contrario,

si se cumple el **TIMEOUT** retira el **ID** de la lista de IDs de réplicas vivas. Luego, utilizando esta última lista envía los mensajes de viveza a las réplicas que contiene.

Otro proceso, utilizando igualmente paso de mensajes, se encarga de la comparación de versiones entre la réplica primaria y las réplicas backup, así como de la eliminación de las réplicas de versión antigua durante el proceso de actualización dinámica. Las réplicas backup envían mensajes **BCKVER** con su número de versión hacia la réplica primaria, la cual recorre el listado de IDs de réplicas backup vivas y ejecuta un proceso ***receive*** con **TIMEOUT** por cada **ID** para obtener estos mensajes. Si recibe el mensaje **BCKVER**, compara la versión enviada por la réplica backup con su versión:

- Si la versión enviada es igual a su versión, la agrega al listado de réplicas backup de versión actual.
- Si la versión enviada es menor a su versión, envía un mensaje **BCKDIE** a la réplica backup correspondiente con lo cual generará su apagado, y elimina el **ID** del listado de réplicas backup de versión actual.
- Si la versión enviada es mayor a su versión, la réplica primaria cambia su estado a "*Upgrading*" y la agrega al listado de réplicas backup de actualización.

Caso contrario, si se cumple el **TIMEOUT** elimina el **ID** del listado de réplicas backup de versión actual. Al inicio del proceso, luego de validar que haya elementos en la lista de réplicas backup de actualización, la réplica primaria recorre cada uno de los IDs que contiene y ejecuta el siguiente proceso:

- Valida que el **ID** no se encuentre en el listado de réplicas backup de actualización que ya generaron anteriormente la eliminación de una réplica backup de versión actual.
- Si se cumple lo anterior, valida si hay elementos en la lista de réplicas backup de versión actual:
    - Si se cumple lo anterior, obtiene el primer **ID** de la lista de réplicas backup de versión actual y valida que no esté en el listado de réplicas eliminadas.
    - Si se cumple lo anterior, envía un mensaje **BCKDIE** a la réplica backup seleccionada y agrega el **ID** al listado de réplicas eliminadas. Luego, agrega el **ID** de la réplica backup de actualización al listado de réplicas que ya generaron la eliminación de una réplica backup de versión actual.
- En caso no haya encontrado elementos en la lista de réplicas backup de versión actual, la réplica primaria envía un mensaje "**BCKCHGPRY**" a todas las réplicas backup nuevas para que inicien el proceso de elección de una nueva réplica primaria según la lógica propia del sistema distribuido y luego cambia su estado por "*Closing*" generando su apagado.

Si bien los procesos de adición y eliminación corresponden a procesos asíncronos regulados por pasos de mensajes entre las réplicas backup y el primario, el proceso de eliminación está regulado internamente por el orden entre la presencia de una réplica backup de actualización y de réplicas backup de version actual, de modo que primero ocurre la identificación de la réplica backup de actualización para luego pasar a la eliminación de la réplica backup de versión actual, eliminando la posibilidad de finalizar con un número menor de réplicas de versión superior.

El modelo de Solarski no muestra detalles explícitos del modo de actualización de las variables de estado de los nuevos backups, pudiendo deducir de la figura 3 que este proceso se realiza durante el encendido y apagado de cada backup. En el modelo planteado, el proceso de actualización de las variables se realiza por un proceso independiente mediante el envío de mensajes de actualización desde el primario hacia las colas generadas tanto para las réplicas backup presentes como para las que se conectarán posteriormente, de modo que al iniciar una nueva réplica backup, ésta podrá obtener las actualizaciones de manera inmediata.

Durante este proceso la réplica primaria revisa constantemente el listado de objetos de actualización y envía a las réplicas backup aquellos que aún no ha enviado, de modo que la transferencia de objetos es incremental en el transcurso del tiempo.

Para el modelo planteado, el mecanismo de elección de líder se basa en el menor **ID** de las réplicas backup y está asociado al mecanismo de viveza entre éstos y la réplica primaria. De manera cíclica, la réplica primaria recibe los mensajes **ACK** de las réplicas backup generando la lista de réplicas vivas, utilizando para ello los **ID** enviados en los mensajes **ACK**. Al no recibir el **ACK** de alguna réplica backup en el **TIMEOUT** definido, la réplica primaria elimina el **ID** correspondiente de la lista que mantiene. Luego envía los mensajes de viveza a las réplicas backup que ha registrado en su lista de réplicas vivas. Por otro lado, las réplicas backup envían sus mensajes **ACK** y esperan el mensaje de viveza de la réplica primaria. En caso no reciban este mensaje en el **TIMEOUT** definido, proceden a elegir un nuevo primario, seleccionando el menor **ID** de la lista de réplicas backup disponibles. Cada réplica backup define el primario que corresponde y el consenso se basa en la recepción y envío de respuestas de viveza del primario hacia los backups.

Estos mecanismos asumen que las réplicas que fallan no vuelven a recuperarse, y que el sistema de conectividad es redundante y no genera particiones, tal como está descrito en el modelo primario - backup.

Los mecanismos de elección de líder y viveza, actualización de variables de estado y actualización dinámica son desarrollados por procesos BackGroundWorker de manera concurrente y asíncrona, regulando el orden de la ejecución de los procesos mediante estados, según sea requerido.

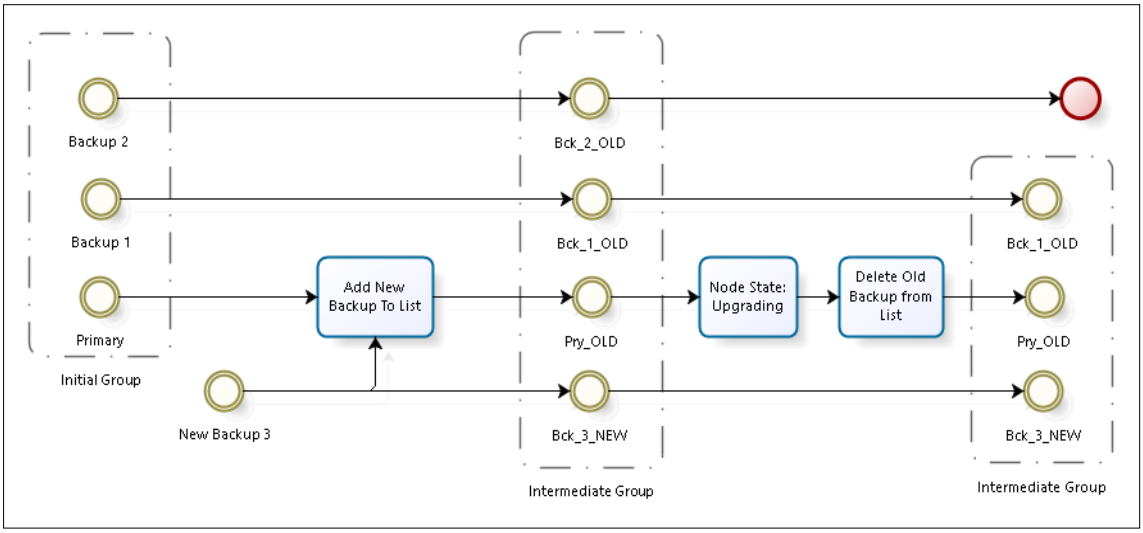


Figura 4. Adición y eliminación de una réplica backup

En la figura 4 se muestra el proceso de adición de una réplica backup actualizada y la eliminación de una réplica backup de versión igual a la de la réplica primaria. Las líneas horizontales dirigidas hacia la derecha representan la línea de tiempo de vida de las réplicas. Luego de adicionar la nueva réplica backup, la réplica primaria cambia su estado por "*Upgrading*" y elimina de la lista que mantiene una réplica backup de versión igual a la suya, enviándole un mensaje "**BCKDIE**" con lo cual esta réplica backup cambia su estado a "*Closing*" y se apaga, lo que se representa con el círculo rojo. Este proceso se realiza continuamente hasta que se eliminan todas las réplicas backup de versión igual a la de la réplica primaria.

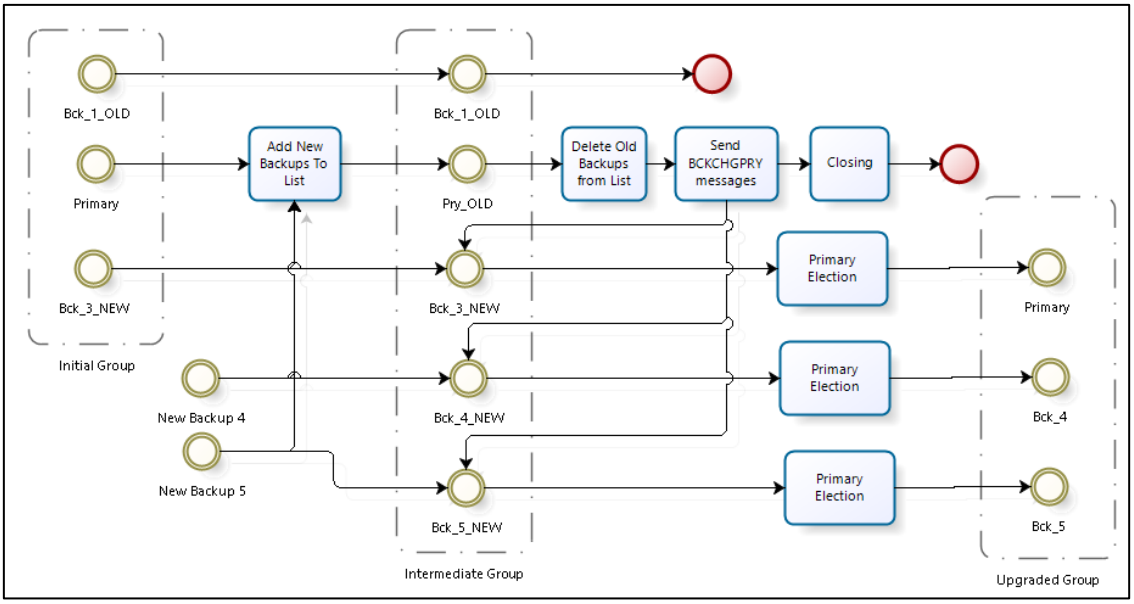


Figura 5. Eliminación de réplicas antiguas y selección de nuevo primario

Luego, se agrega una réplica backup nueva y, al quedar solamente la réplica primaria, ésta envía un mensaje "**BCKCHGPRY**" a todas las réplicas backup nuevas para que inicien el proceso de elección de una nueva réplica primaria según la lógica propia del sistema

distribuido y luego cambia su estado a "*Closing*" y se apaga, tal como se observa en la figura 5.

Capítulo VI: Implementación y Evaluación

16. Diseño Técnico

16.1. Comunicación Cliente-Servidor distribuido

El sistema distribuido implementado considera la conexión de varios nodos clientes a la réplica primaria y de ésta a las réplicas backup siguiendo el modelo cliente/servidor, utilizando pasos de mensajes y colas MSMQ [23] como mecanismos de comunicación y sincronización. Los clientes envían mensajes de requerimiento hacia la cola *Request Queue* de donde la réplica primaria los recibirá para luego procesarlos. A partir del resultado de estas operaciones la réplica primaria elaborará mensajes de respuesta y los enviará a la cola *Resp_Cli_# Queue*, según el código del cliente que envió el requerimiento, de donde el cliente recibirá los mensajes de respuesta. Los procesos de envío y recepción de mensajes de requerimiento y respuesta son asíncronos, pudiendo estar las colas en el dominio de la réplica primaria o en un gateway configurado para contenerlas y establecer la comunicación entre ambos entornos.

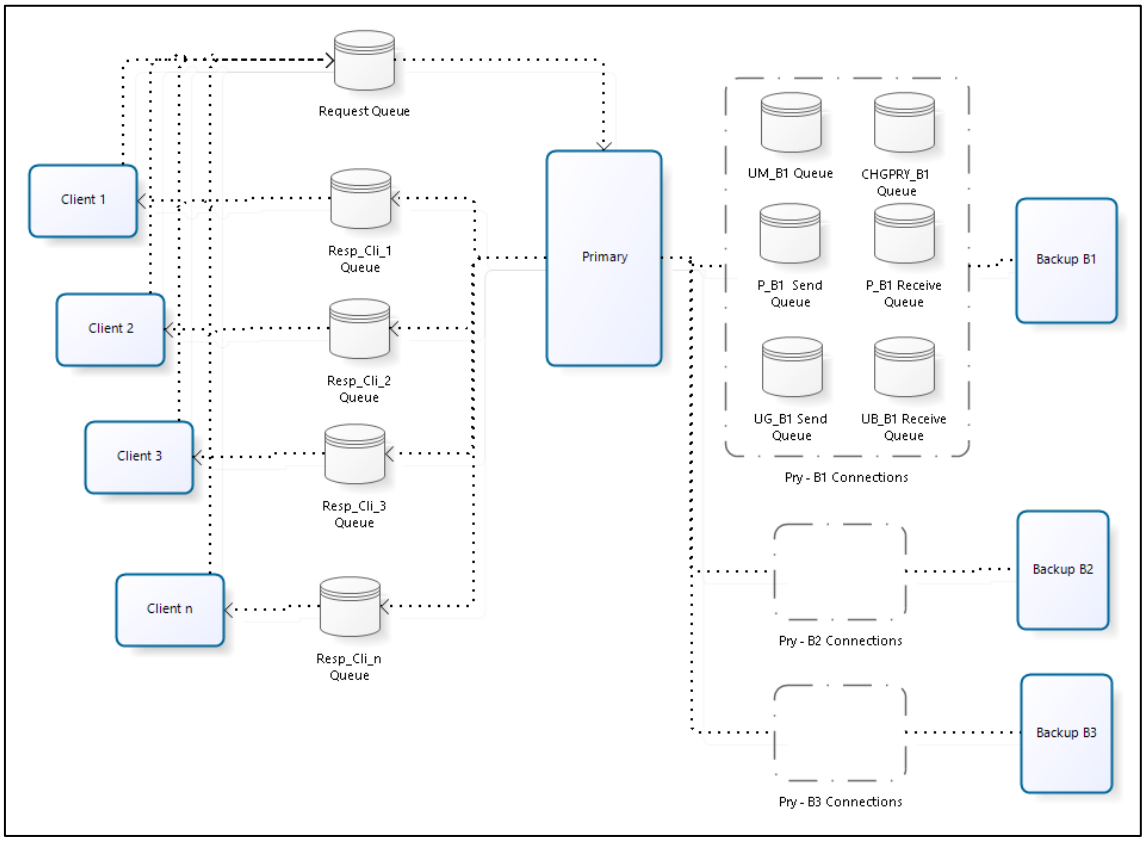


Figura 6. Arquitectura cliente/servidor del sistema distribuido siguiendo el modelo primario-backup

16.2. Comunicación Primario-Backup

La conexión entre la réplica primaria y las réplicas backup también sigue el modelo cliente/servidor mediante paso de mensajes y utilizando colas MSMQ. Cada conexión primario-backup consta de seis tipos de colas con las siguientes funciones:

UM_B# Queue: recibe los mensajes de actualización enviados por el primario, que luego son obtenidos por la réplica backup para procesar la actualización de su estado. Por esta cola también se envían los mensajes de actualización desde el primario hacia las nuevas réplicas backup para su actualización de estado.

P_B#_Receive Queue: recibe los mensajes **ACK** de la réplica backup, que luego son obtenidos por la réplica primaria para mantener la lista de réplicas backup vivas.

P_B#_Send Queue: recibe los mensajes de viveza de la réplica primaria, los cuales son obtenidos luego por la réplica backup correspondiente. En caso se cumpla el **TIMEOUT** sin adquirir un mensaje de viveza, la réplica backup inicia el proceso de elección de líder.

UG_B#_Send Queue: recibe los mensajes **BCKDIE** enviado por la réplica primaria y que al ser obtenidos por la respectiva réplica backup, ésta cambia su estado por "*Closing*" y luego se apaga.

UG_B#_Receive Queue: recibe los mensajes que contienen el número de versión de las réplicas backup que los envían. Al ser obtenidos por la réplica primaria, se realiza la comparación y validación de las versiones y, en caso la versión sea mayor que la de la réplica primaria, se iniciaría el proceso de actualización dinámica.

CHGPRY_B#_Queue: recibe los mensajes de **BCKCHGPRY** enviados por el primario hacia todas las nuevas réplicas backup, de modo que realicen el proceso de elección del nuevo líder entre estas nuevas réplicas.

16.3. Diseño del sistema distribuido primario-backup

A nivel de backend, el sistema consta de 6 procesos que se ejecutan en threads independientes de manera concurrente, los cuales se implementaron utilizando la clase BackgroundWorker para los threads y ConcurrentBag como colección de objetos para variables compartidas entre los procesos concurrentes.

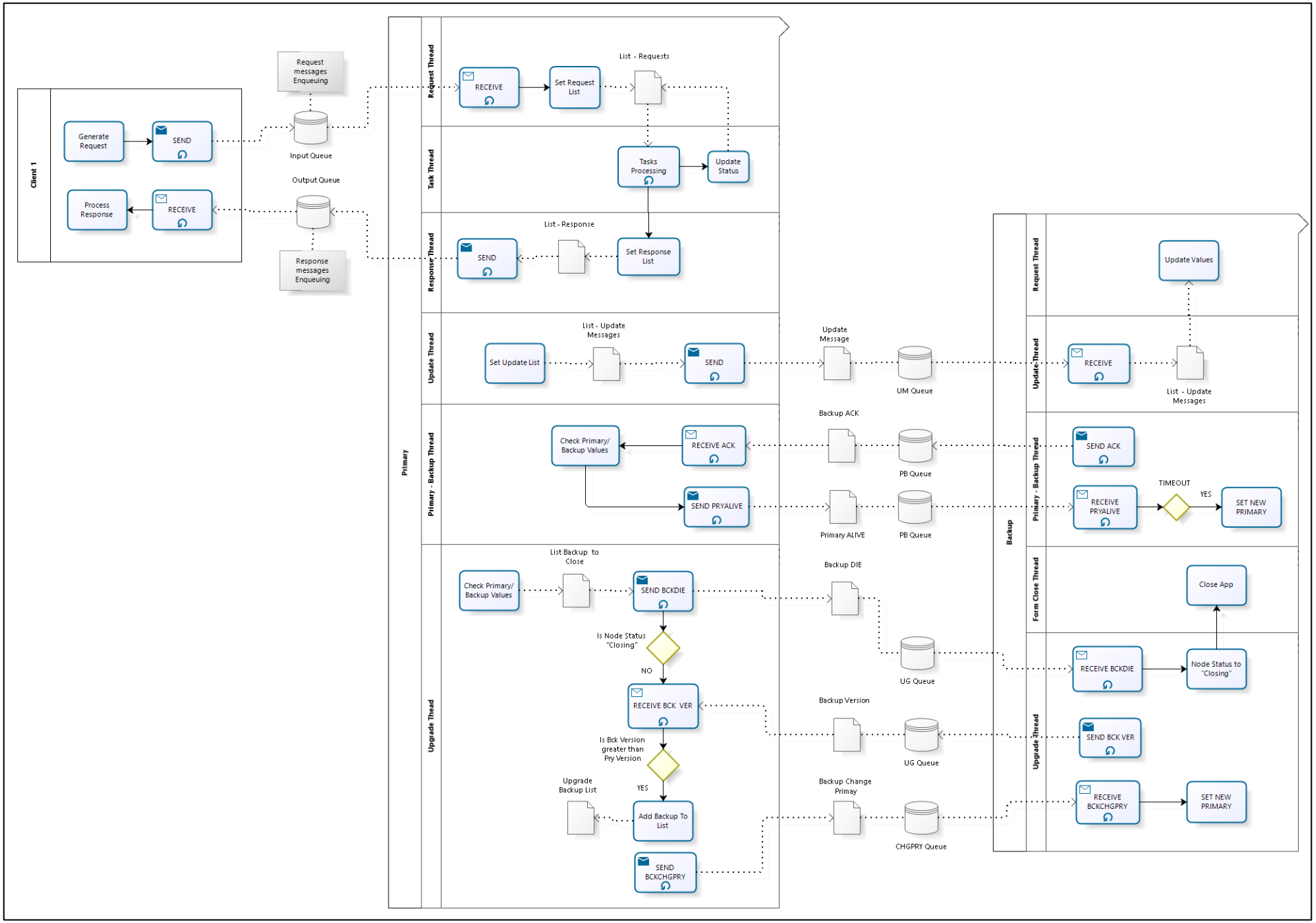


Figura 7. Diseño del sistema distribuido primario-backup

Los threads implementados son:

- ***Request Thread***: recibe los mensajes de requerimientos, genera los objetos correspondientes y los pone en la colección ConcurrentBag de requerimientos.

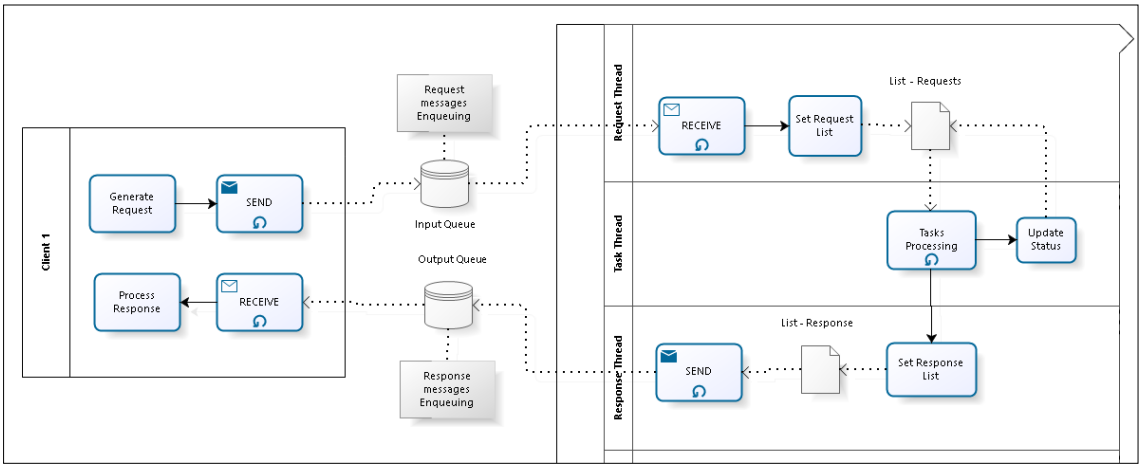


Figura 8. Flujo de procesos entre el cliente y el servidor distribuido

- ***Task Thread***: cada requerimiento genera un Task que ejecuta las acciones correspondientes y actualiza el estado de cada requerimiento según la acción realizada. Estos Tasks forman parte de objetos **SrvTask** que son listados en una colección ConcurrentBag.
- ***Response Thread***: Envía los objetos respuesta obtenidos a partir de la colección ConcurrentBag de objetos **SrvTask** generados en *Task Thread*, hacia la cola del cliente correspondiente. También genera la lista de objetos de actualización **UpdMsg** que utilizará la réplica primaria para enviar a las réplicas backup.
- ***Update Thread***: Para el caso de la réplica primaria, obtiene los objetos de actualización de la lista generada por *Response Thread* y los envía a las réplicas backup. Para el caso de las réplicas backup, reciben los mensajes de actualización **UpdMsg** y los agregan al listado correspondiente. Luego, *Request Thread* se encarga de aplicar las actualizaciones recibidas.

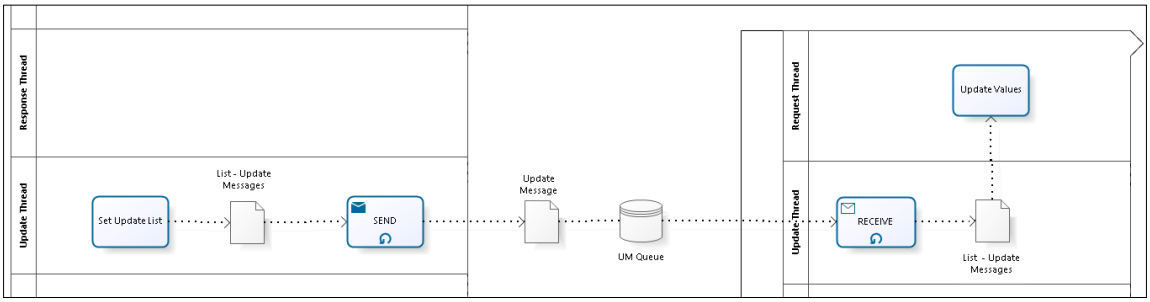


Figura 9. Flujo de proceso de actualización entre el primario y el backup

- ***Primary-Backup Thread***: Para el caso de la réplica primaria, recibe los mensajes **ACK** de las réplicas backup y luego envía los mensajes de viveza

PRYALIVE hacia las réplicas backup de la lista que mantiene. Para el caso de las réplicas backup, envían los mensajes **ACK** hacia la réplica primaria y luego reciben los mensajes de viveza enviados por la réplica primaria. En caso de no recibirlos en el **TIMEOUT** definido, inician la elección de un nuevo primario.

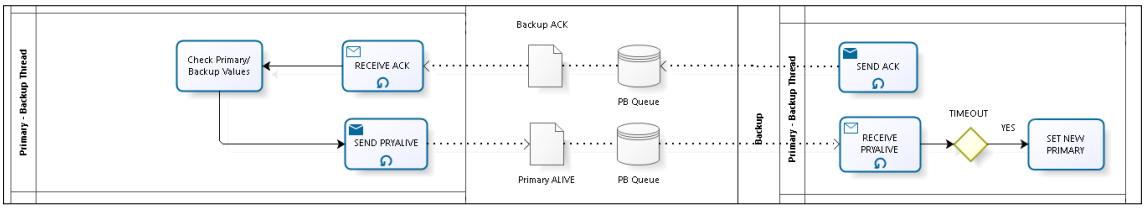


Figura 10. Flujo de proceso de elección de líder y viveza

- ***Upgrade Thread***: Para el caso de la réplica primaria, revisa el listado de réplicas backup de versión anterior y si encuentra les envía mensajes de **BCKDIE** para apagarlas. Si su estado no es de "*Closing*", recibe los mensajes enviados por las réplicas backup con las versiones que poseen y según esto las coloca o no en el listado de versión anterior. Para el caso de las réplicas backup, si no se encuentran en estado "*Closing*", envían los mensajes con la versión que poseen y luego esperan la recepción del mensaje **BCKDIE** durante un **TIMEOUT** definido. Cuando no existen más réplicas backup de versión anterior, envía el mensaje de **BCKCHGPRY** a las réplicas backup nuevas para que elijan el nuevo primario.

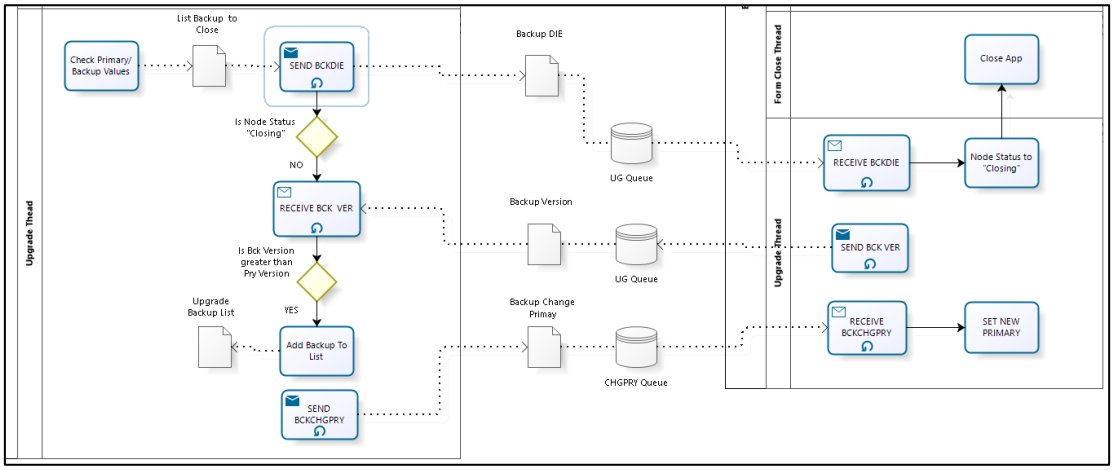


Figura 11. Flujo de proceso de actualización dinámica

17. Prueba de Actualización Dinámica

El proceso de actualización dinámica se realiza a partir de un sistema distribuido de ***n*** nodos desplegado con una réplica primaria y ***n -1*** réplicas backup las cuales poseen la versión v inicial. Para el caso de un sistema de 3 nodos, se ha considerado la versión inicial 2.1.0 y los nodos con ids 24, 25 y 26, donde el nodo 24 es el primario y los nodos 25 y 26 son las réplicas backup.

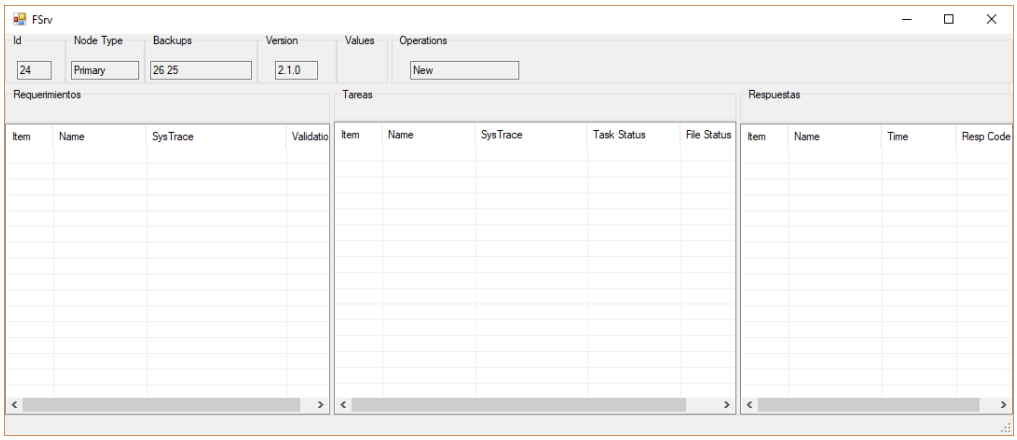


Figura 12. Nodo Primario

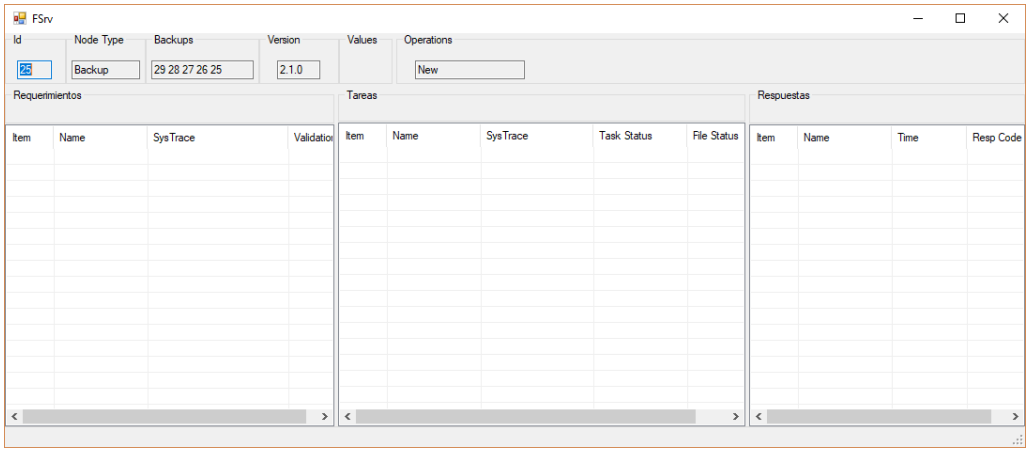


Figura 13. Nodo Backup con id 25

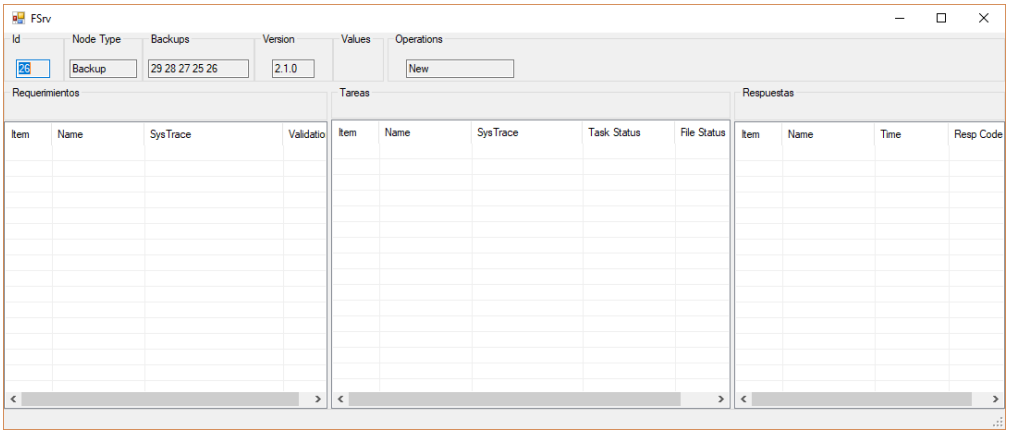


Figura14. Nodo Backup con id 26

La actualización dinámica requiere que el primario tenga información de los nodos que van a actualizar el sistema para que pueda ir agregándolos a la lista de nodos existentes. Por otro

lado, los nodos backup que van a actualizar requieren mantener la información de los ids de los nodos del sistema que actualizan, primero para poder identificar al nodo primario y poder integrarse al sistema, y luego del cierre de éste para ir recorriendo nodo por nodo hasta lograr conectarse al menor id del grupo de nodos actualizados y poder asignarlo como nuevo nodo primario. Esta información se encuentra en el archivo xml de configuración de todos los nodos. Para el sistema de 3 nodos, los nodos que actualizan tienen los ids 27, 28 y 29 y los que actualizarán a éstos tienen los ids 30, 31 y 32.

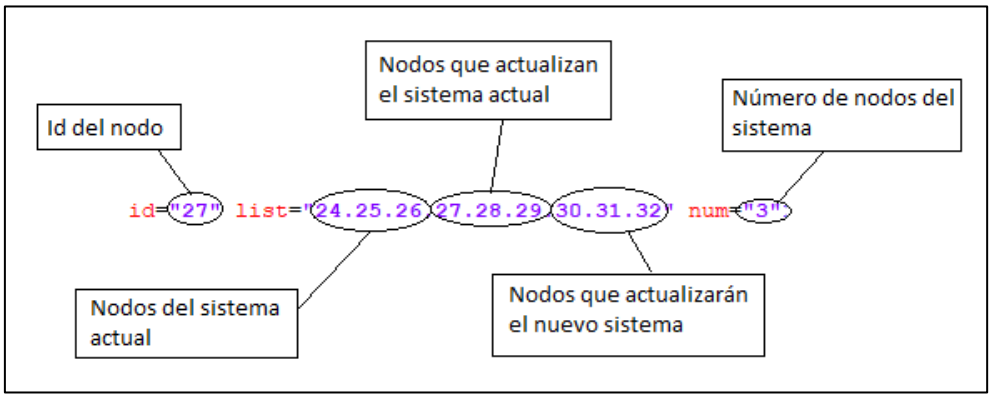


Figura 15. Ids de nodos del archivo xml

Cuando se agrega el primer nodo actualizado, el nodo primario pasa al estado de Upgrading y empieza el proceso de actualización. A medida que se agregan los nodos actualizados se van eliminando los nodos antiguos hasta que queda el nodo primario. Al agregar el último nodo actualizado, el nodo primario envía los mensajes para que los nuevos backups realicen el proceso de elección del nuevo primario entre los nodos actualizados. Para el caso de 3 nodos, la nueva versión es 2.1.1.

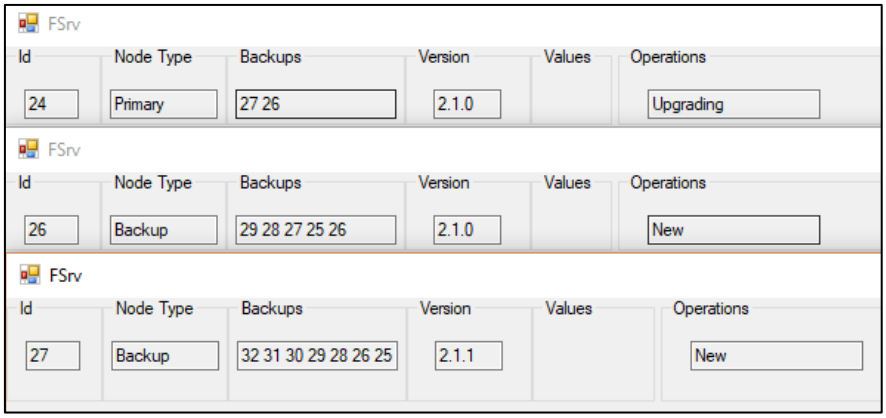


Figura 16. Eliminación del nodo 25

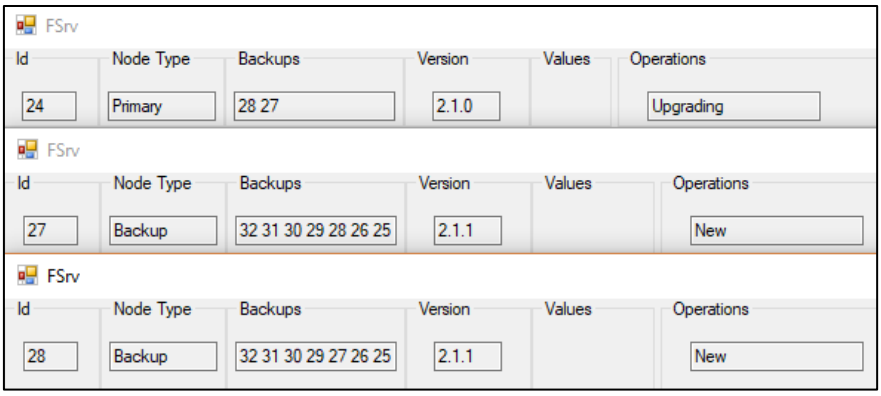


Figura 17. Eliminación del nodo 26

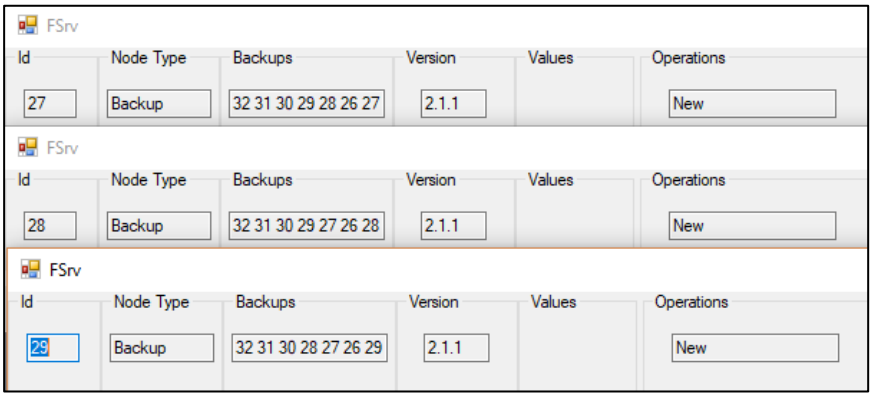


Figura18. Eliminación del primario antiguo

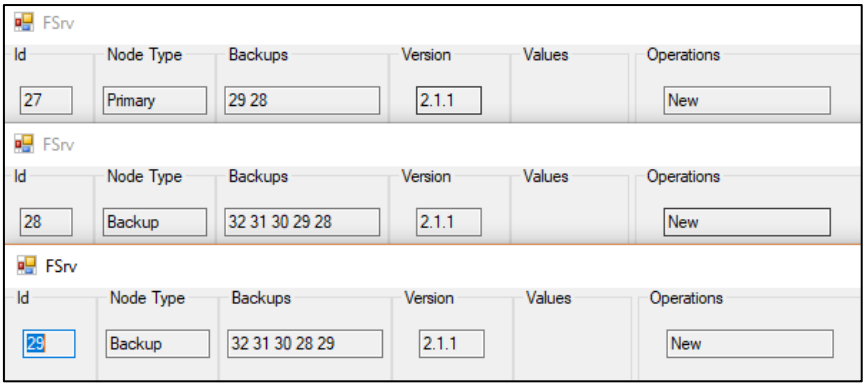


Figura 19. Elección del primario del sistema actualizado

Capítulo VII: Discusión y Conclusiones

18. Resultados y Discusión

La ejecución del sistema distribuido se realizó en una laptop Toshiba Satellite P75-A100 con un procesador Intel Core i7 (4 núcleos), 8 GB RAM y disco duro de 500 GB, con sistema operativo Windows 10 Home Edition. La implementación de las aplicaciones se realizó utilizando Visual Studio 2015 y .NET 4.5.2 utilizando el lenguaje Visual C/C++. La comunicación entre los nodos se realizó mediante colas MSMQ de Windows y se ejecutaron en el mismo equipo. La aplicación implementada para los nodos, primario o backups, permite que se puedan instanciar muchos nodos en la misma computadora o que cada nodo se despliegue en diferentes computadoras. Para evitar complicar la evaluación de prestaciones, los nodos se desplegaron en la misma computadora.

Se evaluaron sistemas compuestos por 2, 3, 4, 5, 6 y 7 nodos respectivamente, y se midió el tiempo de espera entre el envío del mensaje BCKCHGPRY, desde el primario hacia el nodo actualizado con el menor valor de id, y la selección del nuevo primario. Estos valores representan el tiempo de espera requerido por el sistema distribuido actualizado para retomar la continuidad del servicio. Se realizaron 20 repeticiones por cada sistema con la finalidad de obtener valores representativos para las medias, medianas y percentiles 25 y 75 calculados. El uso de 20 ejecuciones permite también trabajar fácilmente con percentiles múltiplos de cinco. Los valores registrados se muestran en la Tabla 1.

El modelo de actualización planteado considera que las réplicas backup actualizadas se agregan al sistema distribuido como si formara parte del sistema, puesto que el proceso de adición no diferencia entre una u otra versión. También, la eliminación de los nodos es tratada como una falla del nodo y el sistema no genera ningún proceso de elección hasta que no queda ningún nodo backup antiguo y el primario notifica a los nuevos nodos backup que ejecuten la elección del nuevo primario.

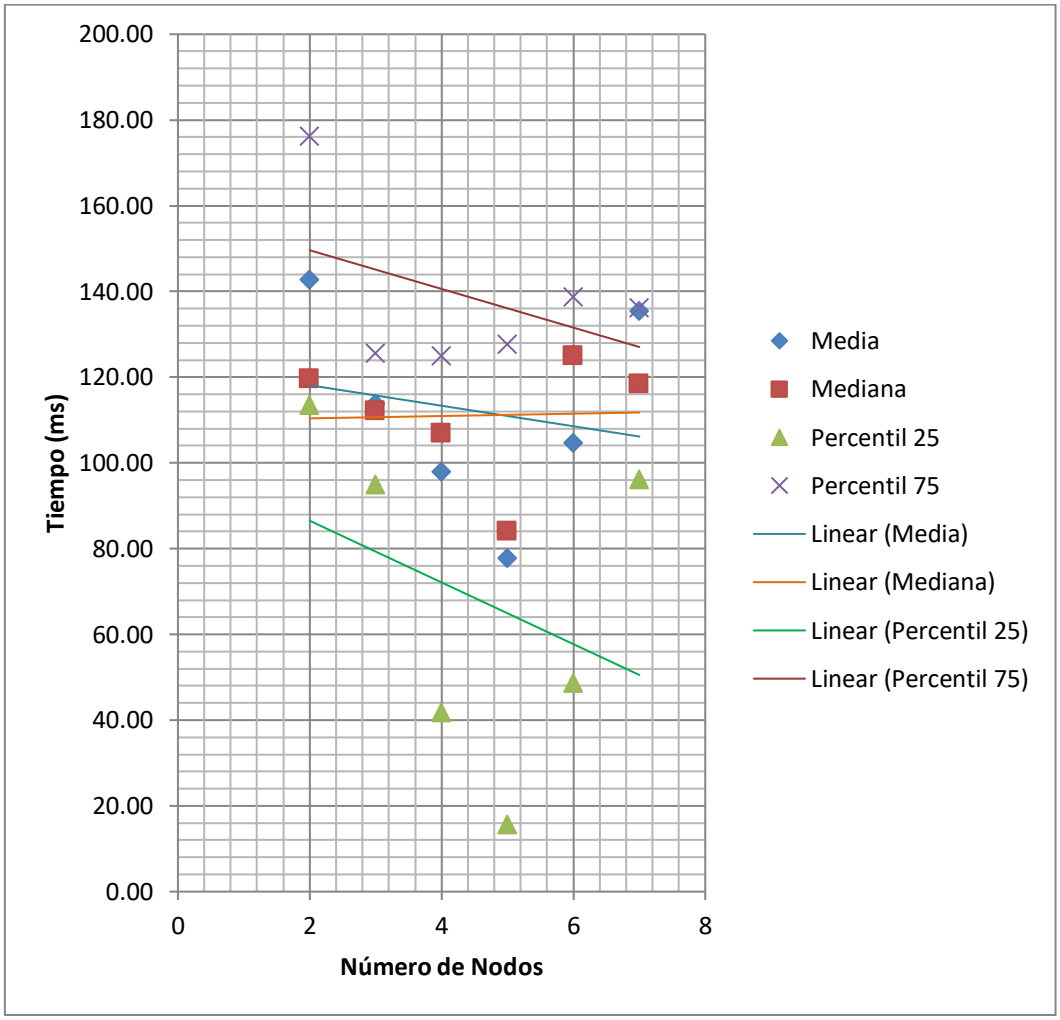
Es posible observar que el tiempo de espera entre la notificación del primario antiguo y la elección del nuevo primario está entre 80 y 126 ms según los valores de mediana obtenidos para los sistemas con diferentes número de nodos. Así también, es posible observar que esta tendencia es constante para los valores de la mediana, diferente a la tendencia decreciente que muestra la media y los percentiles 25 y 75 mostrados en la Gráfica 1.

Por otro lado, las gráficas de dispersión de los valores obtenidos por cada sistema distribuido muestran que la mayor cantidad de éstos se distribuyen en el intervalo entre 100 y 130 ms aproximadamente, conservándose este comportamiento en todos los sistemas con número de nodos diferentes, lo cual se observa mejor superponiendo los valores de todos los sistemas como lo muestra la Gráfica 2.

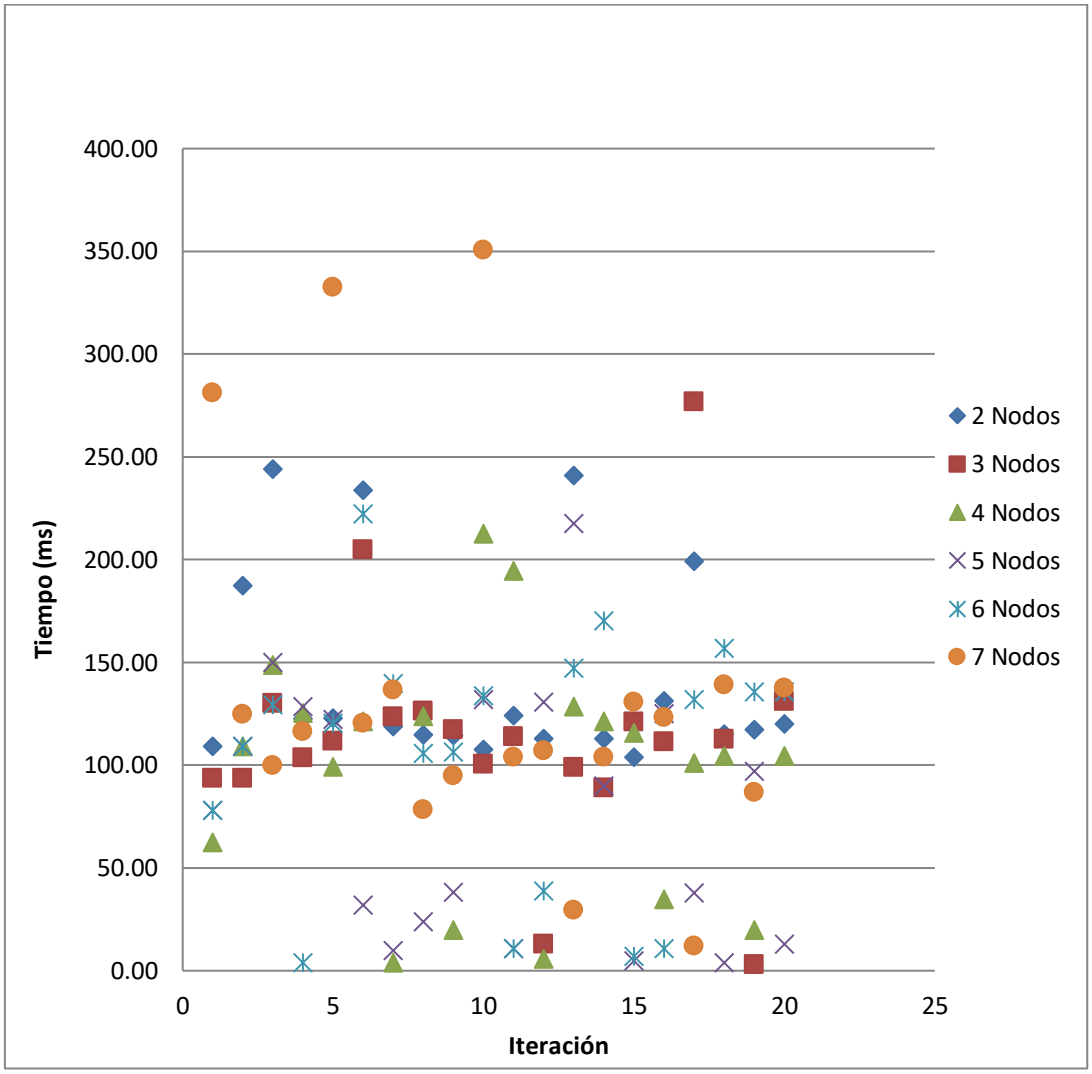
Tabla 1: Intervalo de tiempo entre envío de mensaje **BCKCHGPRY** desde el primario y selección de nuevo primario en el backup

	Intervalos de tiempo (ms)																				Total (ms)			
Nro Nodos	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**Media**	**Mediana**	**Percentil 25**	**Percentil 75**
2	109.37	187.49	244.11	122.55	123.14	233.78	119.07	115.05	114.57	107.89	124.36	113.03	241.21	113.18	103.94	131.27	199.30	115.34	117.44	120.29	142.82	119.68	113.53	176.32
3	93.79	93.72	130.26	103.63	111.69	204.93	123.75	126.33	117.31	100.64	114.10	12.98	99.01	89.17	121.28	111.32	276.77	112.67	3.00	131.20	113.88	112.18	95.09	125.68
4	62.44	109.31	148.70	125.51	99.19	121.44	4.01	123.93	19.99	212.77	194.63	5.97	128.74	121.31	115.99	34.96	101.12	104.67	20.00	104.60	97.96	106.99	41.83	125.11
5	78.12	109.31	150.08	128.57	122.53	32.00	9.99	23.99	38.20	132.03	10.97	130.79	217.80	90.06	5.00	125.13	37.98	4.00	97.08	12.99	77.83	84.09	15.74	127.71
6	78.07	109.33	129.58	4.00	120.61	222.33	139.82	105.78	106.61	133.99	10.97	38.98	147.34	170.25	7.00	10.98	132.10	156.83	135.74	135.70	104.80	125.10	48.75	138.80
7	281.22	124.93	99.92	116.33	332.80	120.51	136.59	78.53	94.95	350.71	103.93	107.08	29.45	103.94	130.84	123.33	11.99	139.24	86.92	137.72	135.55	118.42	96.19	136.33

Gráfica 1: Tendencia de los valores de Media, Mediana, Percentil 25 y Percentil 75



Gráfica 2: Dispersión de valores para los sistemas con diferentes números de nodos



19. Conclusiones

Un requerimiento de los sistemas distribuidos altamente disponibles es la capacidad de evolucionar y mantenerse sin detener los servicios que contienen y que ofrecen a los clientes, de modo que toda esa actividad sea transparente para ellos. El diseño e implementación de estos sistemas requiere del uso de patrones de diseño concurrentes y de los modelos de actualización dinámica que mejor se ajusten al uso del sistema distribuido en cuestión.

Para la implementación del sistema distribuido según el modelo de replicación pasiva se requieren patrones concurrentes que permitan la ejecución segmentada y paralela de los procesos que gestionan su continuidad y actualización dinámica. Así también, la interacción de estos procesos requiere el uso de variables compartidas, por lo que son de utilidad aquellas versiones de listados que permiten su uso concurrente. Este diseño permite el uso de procesos asíncronos, como la recepción de requerimientos y envío de respuestas entre clientes y servidores, y de procesos síncronos o parcialmente síncronos como los relacionados con el envío y recepción de mensajes asociados a los mecanismos previamente mencionados.

El modelo de actualización desarrollado toma como base el modelo planteado por Solarski, con la diferencia de que los procesos de adición y eliminación de nodos backup están gestionados por hilos diferentes. Esto permite eliminar la secuencialidad en el proceso de actualización dinámica de los nodos backup y una gestión con procesos parcialmente síncronos de los mecanismos de actualización de las variables de estado.

Los valores obtenidos para el coste del tiempo entre la notificación del primario antiguo y la elección de líder de los nuevos backups muestran un comportamiento constante para los diferentes sistemas con números de nodos variables como lo muestra la mediana, situándose en un rango entre 80 y 130 ms. Este tiempo corresponde principalmente al tiempo de transmisión de los mensajes y su recepción por parte del nodo backup, puesto que el proceso de elección de líder toma un tiempo muy corto, lo cual es evidenciable por lo tiempos muy cortos obtenidos de alrededor de 4 ms.

El factor que permite minimizar el coste de tiempo para el proceso de actualización dinámica son los pasos de mensajes entre el nodo primario y las réplicas backup, mediante los cuales el primario notifica a las réplicas backup la ejecución de procesos de manera inmediata según lo requiera el estado del sistema distribuido. Por tanto, la optimización del sistema distribuido va de la mano con la mejor gestión de los mensajes entre los diferentes nodos. Es posible decir entonces que una mayor comunicación entre los nodos de modo que les permita tener un mayor conocimiento del estatus del sistema puede favorecer la resolución de las etapas del proceso de actualización dinámica. Si bien un mayor grado de concurrencia es favorable para lograr estos objetivos, la disgregación de funciones implica un proceso de coordinación más complejo por la mayor cantidad de variables y estados a presentarse.

En el desarrollo del presente TFM puedo destacar dos aspectos importantes, un aspecto conceptual relacionado principalmente al modelo de replicación pasiva y el modelo de actualización dinámica, y un aspecto técnico relacionado con la programación concurrente, patrones de diseño, programación orientada a objetos y mecanismos de comunicación interprocesos. Durante mi labor profesional como desarrollador pude evaluar situaciones que eran factibles de resolver mediante mecanismos concurrentes y sistemas distribuidos. Particularmente a partir de la elaboración de un proyecto de migración de un sistema de Transferencia de Archivos segura pude bosquejar un patrón de diseño concurrente que me permitió resolver de manera rápida la implementación del sistema distribuido sobre el cual desarrollaría el mecanismo de actualización dinámica. Por otro lado, la revisión del modelo de actualización dinámica desarrollado por Solarski en un sistema de replicación pasiva me permitió comprender y proyectar los algoritmos a implementar. Es así que para el presente TFM requerí aproximadamente 3 meses de revisión bibliográfica, 6 meses de diseño, implementación y pruebas unitarias de las aplicaciones, 4 meses de trabajo de revisión, actualización del sistema, evaluación del modelo de actualización dinámica y de elaboración del presente documento.

Bibliografía

1. Andrews, G. and Schneider F. "**Concepts and Notations for Concurrent Programming**", Computing Surveys, Vol. 15, No.1, 1983, 1 - 43.
2. Nienaltowski, P. Arslan V. and Meyer B., "**Concurrent object-oriented programming on .NET**", IEEE Proc.-Softw., Vol. 150, No. 5, 2003, 308-314
3. Muñoz Escoí, F.D. et al, "**Concurrencia y sistemas distribuidos**", 2012, Valencia : Universidad Politécnica.
4. Lee, E. "**The Problem with Threads**", Computer, Vol. 39, 2006, 33-42.
5. Lea, D. "**Concurrent Programming in Java: Design Principles and Patterns**", Addison Wesley, Second Edition, 1999.
6. Meyer, B. "**Systematic Concurrent Object-Oriented Programming**", Communications of the ACM, Vol. 36, No. 9, 1993, 56-80.
7. Baskerville, R. et al. "**Extensible Architectures: The Strategic Value of Service Oriented Architecture in Banking**", ECIS 2005 Proceeding, 61, 2005, http://aisel.aisnet.org/ecis2005/61
8. Wilcox, J. et al. **"Verdi: a framework for implementing and formally verifying Distributed Systems"**, Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015, 357 - 368.
9. Shraer, A. et al. "**Dynamic Reconfiguration of Primary/Backup Clusters**", USENIX ATC 2012.
10. Solarski, M. "**Dynamic Upgrade of Distributed Software Components**". PhD Thesis, Faculty of Electronic and Informatic, Technical University of Berlin, Berlin, 2004.
11. Brewer, E. "**Lessons from giant-scale services**". IEEE Internet Computing, Vol. 5, No. 4, 2001, 46-55.
12. Wolski, A., Laiho, K. "**Rolling Upgrades for Continuous Services**". In: Malek, M. et al (eds) ISAS 2004, Lecture Notes in Computer Science, Vol. 3335, 175 - 189, 2005.
13. Ajmani, S. "**Automatic Software Upgrades for Distributed Systems**". PhD Thesis, Department of Electrical Engineering and Computer Science, Massachusets Institute of Technology, 2004.
14. Ajmani, S. et al. "**Modular Software Upgrades for Distributed Systems**". In: Thomas d. (eds) ECOOP 2006 - Object-Oriented Programming, Lecture Notes in Computer Science, vol 4067, 452 - 476, 2006.
15. Peterson, G. "**Myths about the Mutual Exclusion Problem**", Inform. Process. Lett. Vol 12, N° 3, 1981, 115 - 116.
16. Dijkstra, E. "**The Structure of the 'THE' Multiprogramming System**", Commun. ACM, Vol 11, N° 5, 1968, 341 - 346.
17. Hoare, C. "**Towards a Theory of Parallel Programming**". In: C. A. R. Hoare and R. H. Perrott (eds), Operating Systems Techniques, Academic Press, 61 - 71, 1972.
18. Dijkstra, E. "**Cooperating Sequential Processes**". In: F. Genuys (ed) Programming Languages, Academic Press, 1968.
19. Hoare, C. "**Monitors: An Operating System Structuring Concept**". Commun. ACM, Vol 17, N° 10, 549 - 557, 1974.
20. Alsberg, P., Day, J. "**A Principle for Resilient Sharing of Distributed Resources**". Proceeding of the Second International Conference on Software Engineering. 627 - 644, 1976.

21. Budhiraja, N. et al. "**The Primary-Backup Approach**". In: Distributed Systems, ACM Press/Addison-Wesley Publishing Co., 2nd Ed. 199 - 216, 1993.
22. Barlett, J. "**A Nonstop Kernel**". In: Proceeding of the Eighth ACM Symposium on Operating System Principles, SIGOPS Operating System Review. Vol. 15, 22 - 29, 1981.
23. **"Message Queuing (MSMQ)"**: In https://docs.microsoft.com/en-us/previous-versions/windows/desktop/legacy/ms711472(v=vs.85)
24. **"PCI(industria de tarjetas de pago) Normas de seguridad de datos":** In https://es.pcisecuritystandards.org/_onelink_/pcisecurity/en2es/minisite/en/docs/PCI_DSS_v3.pdf
25. Edward G. Coffman Jr. et al. "**Deadlock Problems in Computer Systems**". ARCS, 311-325, 1970
26. Schlichting, R., Schneider, F. **"Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems".** ACM Trans. Comput. Syst., Vol1, N°3, 222-238, 1983.
27. Lamport, L. et al. **"The Byzantine Generals Problem"**. ACM Trans. Program. Lang. Syst. Vol 4, N° 3, 382-401, 1982.