UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Departamento de Sistemas Informáticos y Computación

Máster Universitario en Computación Paralela y Distribuida

Icarus: Replicación de aplicaciones sobre Paxos

Luis Villazón Esteban

Director:
Jose Manuel Bernabéu Aubán

Valencia, Septiembre de 2013

Agradecimientos

Agradecer en primer lugar a mi director, **Jose Manuel**, por haberme ofrecido la oportunidad de realizar este proyecto, porque sin esa oportunidad seguramente nunca me hubiese animado a llevarlo a cabo.

A mis padres **Nieves** y **Luis Enrique**, porque sin ellos nunca hubiera sido posible el embarcarme en esta aventura.

A **Soraya**, porque a pesar de la distancia y de todos estos años aún sigue ahí, apoyando, escuchando todas las tonterías que se me ocurren, y consiguiendo que saque una sonrisa de donde no puede haberla con simplemente decir una palabra.

A **Ismael** por aguantar durante todo este tiempo mis comeduras de cabeza y estar ahí cuando lo necesitaba.

A **Leticia** por ser simplemente ella, por soportar y compartir tanto los buenos como los malos momentos, así como mostrarme todas aquellas cosas que creía ya olvidadas y enseñarme todo lo que me ha enseñado.

A mi mismo, por conseguir seguir adelante, creer y ser capaz de alcanzar las metas que me he propuesto a lo largo de este tiempo.

Si asumes que no existe esperanza,
entonces garantizas que no habrá esperanza.
Si asumes que existe un instinto hacia la libertad,
entonces existen oportunidades de cambiar las cosas.
Noam Chomsky

Índice general

Capítulo 1

Introducción

En este capítulo se pretende sintetizar de manera muy resumida cual es el objetivo del presente proyecto, así como un resumen a muy alto nivel de cuales son las bases sobre las que se sustenta.

A lo largo del documento se realiza una explicación de las características que posee el sistema, la implementación que se ha realizado del mismo y la forma en la cual puede ser utilizado.

1.1. Objetivo

El objetivo del actual proyecto es la implementación de un sistema, gracias al cual se pueda conseguir que una aplicación desarrollada sobre el mismo sea altamente disponible. Con el fin de conseguir esta alta disponibilidad el sistema se basa en la implementación del algoritmo de Paxos[6]. Más concretamente, se ha decidido implementar el sistema a partir del algoritmo descrito por Robber van Renesse[10].

Añadidamente, en la implementación se ha propuesto la modificación de dicho algoritmo para **soportar sistemas dinámicos**, es decir, permitir al sistema crecer y disminuir de manera dinámica, de forma que se puedan añadir nuevos elementos al sistema en tiempo de ejecución y estos no alteren la corrección de la ejecución del sistema.

Debido a este soporte se hace necesaria la implementación de un **sistema de recuperación**, con el fin de que los nuevos elementos que se añadan al sistema alcance el estado actualizado del mismo.

Finalmente también se ha implementado un **recolector de basura**, con el fin de eliminar aquellos datos y operaciones que ya no sean necesarios por los procesos, con el fin de mejorar el rendimiento del sistema y las necesidades de almacenamiento del mismo.

La implementación del proyecto se ha realizado utilizando el lenguaje de programación **CoffeeScript**[1]. Dicho lenguaje permite la compilación directa a Javascript, más concretamente en dicho proyecto se compila a **Node.js**. Las razones para realizar el proyecto sobre CoffeScript y Node.js son las siguientes:

- Node.js permite construir aplicaciones de red escalables de forma rápida.
- El consumo de recursos del mismo y el rendimiento aportado es notablemente superior a otras plataformas de desarrollo.
- CoffeeScript añade soporte a orientación de objetos, lo que mejora la legibilidad del código escrito.

1.2. Explicación

Dado que la base del proyecto se centra en la implementación del algoritmo de Paxos, en esta sección se realiza una explicación a muy alto nivel de cual es el funcionamiento y el objetivo de dicho algoritmo. La explicación se intenta realizar de forma que pueda ser entendida de forma «sencilla» por cualquier lector, por lo que no se entra grandes detalles del mismo.

El objetivo es crear un sistema que proporcione alta disponibilidad a las aplicaciones que hagan uso de del mismo. Uno de los requerimientos para proporcionar alta disponibilidad es la necesidad de mantener un número indeterminado de réplicas de las aplicaciones en funcionamiento, [1] asegurando que todas las réplicas ejecutan las mismas operaciones en el mismo orden y todas mantienen el mismo estado.

Una vez hecha esta descripción se puede definir Paxos como: **Un algoritmo de replicación y consenso, donde se asegura que todas las réplicas ejecutan todas las operaciones en un mismo orden, proporcionando replicación activa**.

Para entender como funciona dicho algoritmo se presentan varios ejemplos, los cuales intentarán facilitar la comprensión del mismo:

Algoritmo de Consenso Nos encontramos en una civilización en la cual todos las personas pueden proponer leyes ante un parlamento. Cada una de las personas es libre de proponer cuantas leyes quiera y en el orden que quiera, estas leyes van llegado al parlamento y se entregan al **presidente** del mismo. El **presidente** según recibe las leyes elige de manera arbitraria una ley y la propone a votación ante los **diputados**. Cada uno de los **diputados** elige si vota o no vota para dicha ley. Si la mayoría de los diputados votan afirmativamente , dicha ley queda aceptada. Si la mayoría de los diputados votan negativamente o no votan, dicha ley no será aceptada. Sucesivamente el **presidente** va proponiendo nuevas leyes, cada una de ellas con un identificador único ordenado, y los **diputados** van votando cada una de ellas. De esta forma, si se desea conocer el orden en el que han sido aceptadas las leyes, simplemente hay que observar el identificador, quedando de esta forma resuelto el problema del ordenamiento de las leyes.

Algoritmo de Replicación Una vez que el **presidente** ha recibido el resultado de la votación y la ley ha sido aceptada , este ha de comunicar la ley a los ciudadanos. Como esta es una civilización moderna, el resultado de la votación

[1] Idealmente este número debería ser un número entero mayor que 0. Si es igual que 0 queda claro que el sistema sería nulamente disponible. No se han realizado pruebas en lo referente a que el número de réplicas fuesen negativas, queda al gusto del lector el realizar sus propias conjeturas y poner en conocimiento del escritor sus conclusiones.

es enviada por email a todos y cada uno de los ciudadanos de la civilización. Los ciudadanos reciben en su buzón de correo electrónico dicho resultado y aprenderán la nueva ley, sabiendo que a partir de ese momento esa es la ley que deben seguir.

Ahora bien, puede darse el caso de que los emails lleguen a los ciudadanos de forma desordenada, en cuyo caso el ciudadano simplemente ha de observar el identificador de dicha ley, obedeciendo cada una en el orden indicado por el identificador y no por el orden de llegada. De esta forma queda asegurado que todos los ciudadanos cumplirán todas las leyes aprobadas y las cumplirán en el mismo orden.

También puede ocurrir que una ley no le llegue a un ciudadano, o esta tarde mucho en llegar, pero lleguen otras posteriores. En este caso, el ciudadano al no saber que dice la ley que le falta, no seguirá las leyes recibidas hasta que no reciba la ley pendiente. Se asegura que los ciudadanos cumplen las leyes en el mismo orden, pero no se asegura que todas las leyes sean cumplidas al mismo tiempo.

Simplificando, ambos algoritmos quedarían de la siguiente forma:

1. Un ciudadano propone una nueva ley, enviando esta al parlamento donde la recibe el **presidente**.

2. El presidente propone a votación una de las leyes que ha recibido, notificando esta a todos los **diputados**.

3. Los diputados deciden si aceptan o no dicha ley. (votando afirmativamente o no votando).

4. Si la ley ha sido aprobada , el **presidente** envía un email con dicha ley a todos los ciudadanos.

El objetivo, por lo tanto, de la implementación de Paxos, es conseguir un algoritmo que sea capaz de ejecutarse en N máquinas y realizar de manera correcta el consenso y la replicación de todas las peticiones recibidas, de forma que estas sean replicadas en cada una de las máquinas que desempeñen dicho rol y en el orden que en cual hayan sido decididas.

Capítulo 2

Descripción del Algoritmo

2.1. Introducción

Una vez explicado en lenguaje coloquial cual es el objetivo del algoritmo de Paxos, es el momento de realizar una descripción más formal del mismo. Como se ha podido observar en la descripción, en el algoritmo intervienen tres roles diferentes: Ciudadanos, Presidente y Diputados.

Cada uno de estos roles se identifican con un proceso diferente dentro del sistema de Paxos, e intervienen en una o más etapas del algoritmo:

- **Réplica:** o ciudadano. Es el **endpoint** de entrada y salida de Paxos. Contiene el programa que se desea replicar y recibe las entradas que se desean ejecutar dentro del sistema. Proporciona la alta disponibilidad de cara al cliente; se realiza una replicación activa en la cual cada proceso ejecuta las operaciones siguiendo un orden causal.
- **Líder:** o presidente. Es el encargado de inicializar el algoritmo de consenso. Recibe las peticiones de las **réplicas** e inicia una votación para cada una de las operaciones recibidas.
- **Aceptor:** o diputado. Es el encargado de votar y decidir si acepta o no acepta una operación propuesta por el **líder**. Actúa como la memoria del sistema almacenando cada una de las operaciones que se aceptan; de forma que posteriormente ante una caída, fallo o conexión de un nuevo **líder**, este pueda recuperar el estado anterior preguntando únicamente a los **aceptores** por todas las operaciones aceptadas.

Cada una de las **réplicas** forman parte del sistema de replicación del algoritmo. El algoritmo utiliza **replicación activa**, es decir, todas las réplicas pueden recibir y procesar las mismas operaciones; además todas ellas han de ejecutar las mismas operaciones siguiendo un mismo orden, de forma que **eventualmente todas las réplicas contengan el mismo estado**.

Para conseguir que todas las **réplicas** realicen la ejecución de las operaciones en el mismo orden utilizando replicación activa, se necesita como mínimo orden causal. Este orden es conseguido gracias al algoritmo de consenso en el cual intervienen tanto el **líder** como los **aceptores**.

A continuación se realiza una explicación del algoritmo de consenso y el algoritmo utilizado para el aprendizaje de la operación:

2.2. Algoritmo básico

A continuación se muestra el algoritmo básico de Paxos, en el cual se encuentra diferenciado en dos fases. En una primera fase se realiza la aceptación de la operación que se desea ejecutar. En la segunda fase se realiza la notificación

de la operación aceptada y se ejecuta o **aprende** dicho valor por parte del sistema.[5].

Dado que la descripción que se va a relatar es una versión simplificada del algoritmo, se han simplificado también los roles utilizados, de forma que estos se adapten mejor al los nombres dados a cada una de las fases del algoritmo. De esta forma nos encontramos con los siguientes roles:

- **Proposer**: Es el rol encargado de proponer valores dentro de sistema. Se puede considerar como el punto de entrada de la aplicación y tendría su equivalencia con el rol de **réplica** y **líder**.
- **Learner**: Es el rol encargado de aprender las operaciones que han sido aceptadas dentro del sistema. Se puede considerar como la aplicación que se va a replicar y tendría su equivalencia con el rol de **réplica** y **líder**.
- **Aceptor**: Es el rol encargado de realizar las votaciones, decidiendo si se acepta o no una operación. Tiene su equivalencia con el rol de **aceptor**.

2.2.1. Aceptación de operación

En la fase de «aceptación de la operación» se pretende aceptar un valor de entre todos los valores recibidos por los **proposers**. Dado que cada uno de los **proposers** puede proponer diferentes valores, es necesario que en el sistema se llegue a un acuerdo para elegir cual de todos esos valores será el elegido, aceptado y aprendido.

Para realizar la aceptación de un valor se produce una comunicación entre un **proposer** y los diferentes **acceptores** pertenecientes al sistema. En los mensajes utilizados en la comunicación se utilizan los siguientes campos:

- **n**: Se define como el ballot sobre el cual se desea aceptar un valor. Es utilizado por el sistema para asegurar orden causal. Si se reciben dos mensajes de **proposers** diferentes, con ballots m y n respectivamente y $m > n$, entonces se sabe que el mensaje m es posterior a n y este ha sido consecuencia de la promesa o aceptación de un mensaje previo.
- **v**: El valor que se desea aceptar en el algoritmo.

El algoritmo para realizar la **aceptación de un valor** es el siguiente:

1. Un **proposer** envía un mensaje llamado *prepare* a todos los **acceptores** que forman parte del sistema. Este mensaje contiene únicamente el ballot actual del **proposer** .
2. Cada uno de los **aceptores** recibe el mensaje y realiza las siguientes acciones:

   a) Compara el ballot recibido en el mensaje con el ballot que contiene almacenado. Si el **ballot del mensaje es mayor que el ballot almacenado y no se ha aceptado un valor**, entonces envía un mensaje llamado *promise* al **proposer**. Este mensaje indica que el **aceptor** no aceptará ningún nuevo mensaje con un ballot inferior al actual.

   b) Compara el ballot recibido en el mensaje con el ballot que contiene almacenado. Si el **ballot del mensaje es inferior que el ballot almacenado y/o se ha aceptado un valor**, entonces envía un mensaje al **proposer** conteniendo el ballot almacenado y el valor que ha sido aceptado con dicho ballot.

3. El **proposer** recibe los mensajes de los **aceptores** pudiendo suceder dos casos:

   - Se ha recibido un mensaje *promise* de una mayoría de los **aceptores**. En este caso envía un mensaje *accept* conteniendo el ballot y el valor que desea proponer a todos los **aceptores** que han respondido a la petición.
   - Se ha recibido un mensaje con un ballot superior al propio. En este caso el **proposer** ha de actualizar su ballot, siendo el nuevo ballot superior al recibido y adoptar el valor recibido en el mensaje como propio. A continuación volverá a realizar el paso 1 del algoritmo.

4. El **aceptor** recibe un mensaje *accept*. Si el ballot es mayor o igual que el almacenado entonces da por aceptado dicho valor.

Como se puede observar en este paso se produce una votación para decidir que valor, de todos los recibidos, será aceptado . Un líder, en este caso el **proposer**, envía una petición a todos los aceptores para decidir un valor. Cada uno de los **aceptores** decide unilateralmente si acepta o no dicho valor y se lo comunica al **proposer**. Si la mayoría de los **aceptores** le responde afirmativamente, sabrá que él y solo él puede proponer valores, que su valor será propuesto y podrá enviar su valor.

La respuesta en forma de promesas evita el bloqueo del sistema. Si al llegar un mensaje *prepare* el **aceptor** únicamente acepta el valor recibido, y el **proposer** no envía nunca la petición *accept*, el sistema se quedaría bloqueado y no avanzaría. Con la promesa se asegura que si aparece una nueva petición con un valor superior esta petición será la nueva prometida.

A su vez, utilizando los valores de ballot como ordenamiento temporal, se asegura que todo **proposer** reciba un valor aceptado anteriormente (en caso de que alguno haya sido escogido). Si un valor ha sido aceptado anteriormente todo **proposer** recibirá este valor y no podrá ser modificable, manteniendo la

consistencia del sistema. Se asegura de esta forma que una vez que un valor ha sido aceptado no va a poder ser modificado.

2.2.2. Aprender la operación

Una vez que el valor ha sido aceptado por una mayoría de los **aceptores** se procede a aprender dicho valor. El valor será aprendido por todos los **proposers** que forman parte del sistema, de manera que todos ellos una vez hayan aprendido dicha operación, contendrán el mismo estado:

1. Cuando un **aceptor** ha aceptado una proposición envía un mensaje a todos los **learners** del sistema indicando que dicha proposición ha sido aceptada.

2. Si un **learner** recibe el mensaje de aceptación de una mayoría de los **acceptors** del sistema entonces aprende dicha proposición.

En este paso se produce la ejecución de la proposición aceptada. Dado que ha existido una votación y la mayoría de los **aceptores** han aceptado la misma proposición, todos los **learners** aprenderán la misma proposición.

Tal como está descrito el algoritmo, se puede observar que se sigue una política de **replicación activa**, se envía un mensaje a todos los nodos y todos los nodos ejecutan la operación. Esto puede producir un alto número de intercambio de mensajes , del orden de $NA * NL$ [1] , así como una limitación en la escalabilidad del sistema. Para solventar este problema se proponen dos soluciones:

- Nombrar un **learner** líder. Todos los **aceptores** enviarán sus mensajes al líder, una vez que el líder recibe un mensaje de la mayoría de los **aceptores** propaga el mensaje al resto de los líderes. Se utiliza una política de **replicación pasiva**. Se consigue reducir el número de mensajes a $NA + NL$, pero se aumenta la carga de trabajo de un único nodo. Si este nodo se cae entonces habrá que realizar una nueva elección de líder, con la penalización que esto pueda acarrear.

- Se nombra un conjunto de líderes. Los **aceptores** enviarán sus mensajes a dicho conjunto, cuando un líder recibe un mensaje de la mayoría de los **aceptores** , propaga el mensaje al resto de los **learners**. Utiliza una replicación híbrida entre pasiva y activa, mejorando la escalabilidad y la disponibilidad del sistema. Se aumenta ligeramente el número de mensajes necesarios para el aprendizaje del valor, pero si se cae algún líder el sistema seguirá en funcionamiento.

[1] NA: Número de aceptores. NL : Número de learners

2.2.3. Aceptar y aprender múltiples valores (Multi-Paxos)

Hasta ahora se ha descrito el algoritmo en referencia a aceptar y aprender un único valor, si se desean aprender varios valores hay que realizar una pequeña modificación sobre el algoritmo, de modo que soporte la aceptación y el aprendizaje de n operaciones.

La solución propuesta consiste en ejecutar varias instancias del algoritmo, de forma que cada una de las instancias elija un único valor de entre todos los dados. El funcionamiento simplificado es el siguiente:

- Se ejecutan n instancias del algoritmo, todas ellas con el mismo número de procesos y todos los procesos ejecutando los mismos roles.
- La numeración de la instancia indica el orden de ejecución de las operaciones, la instancia i se ejecutará antes que la instancia $i + 1$.
- La elección de la operación a ejecutar no viene restringida por la numeración de la instancia, se puede elegir un valor para la instancia $i + 1$ y no haber elegido aún un valor para la instancia i.
- Si un **learner** recibe un valor para la instancia $i + 1$ pero no ha recibido un valor para la instancia i , el **learner** tiene que ejecutar la primera fase del algoritmo de consenso de Paxos para dicha instancia. De esta forma recibirá el valor asociado a la instancia i y podrá continuar aprendiendo valores posteriores. Este es el mecanismo utilizado para actualizar/recuperar procesos que se han caído del sistema o han sufrido pérdida de mensajes.

2.3. Algoritmo Avanzado

La implementación realizada de Paxos para dicho algoritmo se basa en el artículo escrito por Robbert van Renesse [10], el cual muestra una implementación del algoritmo de Multi-Paxos [2]. La implementación realizada en dicho artículo se realiza de una forma simple y sin ningún tipo de optimización; mostrando cómo debería ser la base a utilizar para implementar un algoritmo de este tipo. A su vez se introducen varias modificaciones sobre el algoritmo original de Paxos, mejorando la eficiencia del mismo.

A continuación se describen las características de la implementación realizadas en [10]:

2.3.1. Roles del sistema

En la implementación de dicho algoritmo se crean nuevos roles con el fin de que cada rol tenga un objetivo único, los roles definidos son los siguientes:

[2] Multi-Paxos se basa en la múltiple ejecución de varias instancias de Paxos para la elección de varios valores

- **Cliente**: Proceso que **envía peticiones al sistema para que sean procesadas**. Dado que pueden existir múltiples clientes, cada cliente ha de poseer un identificador único y ha de implementar un contador de operaciones internas con el fin de poder diferenciar y ordenar sus operaciones.

  El cliente, por defecto envía las peticiones a todas las réplicas y acepta únicamente la primera respuesta, soportando **fallos de caída-parada**. Para soportar fallos bizantinos se deberían recibir $(2f + 1)/2$ respuestas idénticas, donde f sea el número de réplicas que se permiten fallar.

- **Réplica**: Proceso que aúna los roles de **proposer** y **learner** del artículo *Paxos Made Simple*[5] descrito anteriormente. Es el encargado de proponer las operaciones enviadas por el cliente y de ejecutar las operaciones decididas.

- **Aceptor**: Proceso que realiza la votación de las peticiones propuestas, decidiendo si acepta o no la operación propuesta. También actúa como memoria principal del sistema, almacenando todas las operaciones aceptadas.

- **Líder**: Proceso líder del sistema, es el encargado de iniciar las votaciones y de decidir cuando una operación ha sido aceptada. En [5] el líder no existía como cuál, era el **proposer** el encargado de realizar este rol.

- **Scout** Proceso encargado de la primera fase del *Quorum*, promesa de reserva de valor.

- **Commander** Proceso encargado de la segunda fase del *Quorum*, aceptación de valor.

2.3.2. Implementación de Multi-Paxos

- La **adopción de un ballot se realiza anteriormente a la petición del cliente**, exactamente en el momento de inicio del **líder**. De esta forma, siempre que no exista un conflicto entre **líderes**, el número de mensajes necesarios para aceptar un valor se ve reducido en $(3 * num\_aceptores)/2$ mensajes.

- El **cliente** envía un mensaje a todas las réplicas del sistema , al contrario que en [5] en el cual se envía a una única **réplica**. En este punto se utiliza **replicación activa**, todas las **réplicas** realizarán la misma acción de forma determinista y responderán al cliente.

- El **cliente** únicamente acepta la primera respuesta recibida, ignorando el resto. Se consigue soportar fallos por **caída-parada**, si bien es sencillo modificar este punto para que se soporten fallos bizantinos.

- El mensaje de adopción, segunda fase del consenso, se envía a todos los **aceptores** del sistema. En el algoritmo básico únicamente se envía a los **aceptores** que respondieron en la primera fase.

- Al permitirse la existencia de varios **líderes**, puede ocurrir que entren en conflicto para la elección de valores. Cada vez que se entra en conflicto han de enviar una nueva petición de adopción de un nuevo ballot. Este es el único momento en el cual se ha de realizar de forma completa el algoritmo de **Aceptación de un valor**.

- Un alto número de **líderes** puede influir negativamente en la eficiencia del sistema. Cada **líder** puede proponer un valor, este es desechado al existir un ballot superior aceptado, y tiene que volver a adoptar un nuevo ballot. Para solventar este problema se propone dejar en estado suspendido a todos los **líderes** menos a uno, que será el encargado de soportar todas las peticiones. Cada uno de los **líderes** dormidos monitorizará al **líder** activo, de forma que cuando este caiga uno de ellos tomará el control. Se utiliza en un principio **replicación activa**, pero se tiende a que el sistema funcione con **replicación pasiva** con el fin de diminuir los conflictos y mejorar el rendimiento.

- Los **aceptores** mantienen un control sobre todas las peticiones aceptadas, de forma que cuando un nuevo **líder** se conecte al sistema, automáticamente al realizar una petición de adopción reciba el estado actual del sistema. Dado que siempre se necesita la aceptación de la mayoría de los **aceptores** para aceptar una petición, un **líder** después de una petición tendrá el estado completo de todas las operaciones realizadas.

- Las **réplicas** ordenan las peticiones por orden de llegada y las asignan a la primera instancia libre que no ha sido decidida. Una vez que una instancia ha sido decidida propone las peticiones pendientes de esa misma instancia a la siguiente libre. De esta forma eventualmente se deciden todas las peticiones.

- Si las **réplicas** deciden una operación en una instancia i, pero no ha sido decidida una instancia $i-1$, no se realizará la ejecución de dicha instancia hasta que no lleguen todas las anteriores a la misma. Se asegura que todas las **réplicas** ejecuten las operaciones en el mismo orden y mantengan el mismo estado.

2.4. Algoritmo Implementado

El algoritmo implementado para este trabajo se encuentra basado en el algoritmo avanzado descrito anteriormente, realizando sobre el mismo algunas modificaciones con el fin de conseguir una mayor eficiencia.

A continuación se describen cada uno de los pasos que sigue el algoritmo, tanto para conseguir el consenso como para la replicación de cada una de las réplicas que formarán parte del sistema. En dicha descripción, dado que se describen en capítulos posteriores, se han obviado tanto el **algoritmo de recolección de basura** como el **algoritmo de recuperación**.

Paso 1: Adoptando un ballot Cuando un **líder** se conecta al sistema la primera acción que intenta realizar es conseguir adoptar un ballot. Cada uno de los ballots se puede definir como una vista diferente, de modo que cuando un **líder** adopta un nuevo ballot se está produciendo un cambio de vista [9].

Los ballots se encuentran formados por dos componentes:

- Un número entero secuencial, el cual indica la posición temporal del ballot, de forma que si existen dos ballots con números n y $n+1$, se sabrá que el ballot con valor $n+1$ ha sido creado con posterioridad al ballot con valor n.
- Un identificador del **líder**. El ballot contiene el identificador del líder que lo ha adoptado. Este líder será el único que puede decidir operaciones dentro de la vista.

  Dado que el ballot define un cambio de vista, es necesario que sea posible ordenar los mismos de forma que se sepa cuando un ballot es posterior a otro. Esta ordenación se realizada utilizando las dos componentes que forman parte del ballot en cuestión.

  $$\forall b(i,l), \exists b(i',l') / b(i,l) > b(i',l') \leftrightarrow (b(i) = b(i') \wedge b(l) > b(l')) \vee b(i) > b(i')$$

  Un ballot es mayor a otro dado si y solo si, los identificadores de los ballots son iguales y el identificador de un líder es mayor, o el identificador del ballot es mayor.

Para realizar esta adopción se realizan las siguientes acciones:

- El **líder** crea un **scout** con su ballot actual, por defecto [0,*ident_líder*]. El **scout** envía dicho ballot, dentro de un mensaje llamado *P1A*, a todos los **aceptores** que forman parte del sistema y se mantiene a la espera de su respuesta. A su vez el líder envía la posición, o slot, hasta la cual ha decidido operaciones, de esta forma disminuye la carga de procesamiento de los aceptores y el tamaño del mensaje de respuesta.
- El **aceptor** recibe el mensaje *P1A* enviado por el **líder** a través del **scout** y realiza las siguientes comprobaciones:
  - Compara el ballot enviado con el ballot que actualmente se encuentra procesando. Si el ballot es mayor, es indicio de que se ha producido un cambio de vista y actualiza su propio ballot.
  - Genera un mensaje de respuesta *P1B*, este mensaje contiene una lista con todas las operaciones decididas desde el slot indicando en el mensaje hasta la última posición decidida del cual tiene constancia el **aceptor**. También contiene el ballot actual del **aceptor** y la posición o slot hasta la cual se ha ejecutado el recolector de basura.
- El **scout** recibe las respuestas enviadas por los **aceptores** y realiza las siguientes acciones:

  - Compara el valor del ballot recibido con el suyo propio. Si el ballot coincide, introduce las operaciones enviadas en el mensaje en una lista. Si la operación recibida ya se encuentra en la lista, únicamente almacena aquella operación con el ballot más alto.
  - Compara la posición hasta la cual se ha ejecutado el recolector de basura propio con el valor recibido en el mensaje, adoptando el valor superior.
  - Si ha recibido una respuesta por parte de una mayoría de los **aceptores**, es decir de $(n/2) + 1$, el **scout** da por adoptado el ballot y lo notifica al **líder**.

- El **líder**, una vez tiene constancia de la adopción del ballot:
  - Actualiza su lista de operaciones decididas, la lista donde contiene todos los slots que han sido decididos y la posición del recolector de basura.
  - Por cada una de las operaciones decididas envía un mensaje *DECISION* a todas **réplicas** conocidas. Este paso es una optimización respecto al algoritmo de [10]. En dicho algoritmo, una vez que se ha adoptado un ballot, se reenvían las operaciones recibidas a los **aceptores** para que sean decididas por el nuevo ballot, lo que conlleva un nuevo envío, procesamiento y recepción. Al eliminar este paso se ahorra el tiempo invertido en estos pasos y no se ve alterada la corrección del algoritmo.
  - Crea un **commander** para cada una de las operaciones recibidas cuando se encontraba en un estado **inactivo** y que no se encuentran en la lista de operaciones aceptadas. Únicamente se proponen a votación aquellas operaciones que aún no han sido aceptadas.
  - Se ejecuta el recolector de basura, eliminando todas aquellas operaciones que se encuentran en la lista de operaciones decididas, y con un slot inferior al indicando por la posición del recolector de basura.

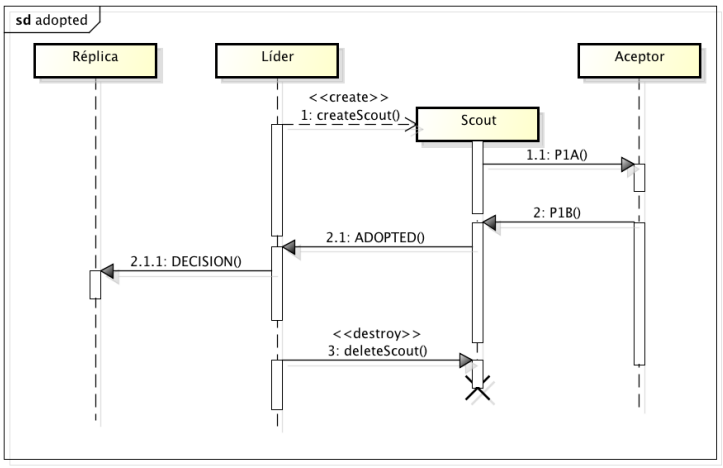


Figura 2.1: Adopción de ballot

Paso 1.1: Recibiendo Preempted Cuando el **scout** o el **commader** se encuentran en el proceso de espera de respuestas por parte de los **aceptores**, y reciben un mensaje con un ballot diferente al que ellos poseen, han de finalizar su ejecución y avisar al líder con un mensaje *PREEMPTED*. Este mensaje indica que se ha realizado un cambio de vista, y por lo tanto el **líder** no podrá adoptar nuevos valores hasta que actualice su vista y se cree una nueva en la que él sea el dueño.

Los pasos que se siguen son los siguientes:

- Cuando el **scout** o el **commander** reciben un mensaje de los **aceptores** con un ballot diferente al que ellos poseen, finalizan su ejecución y envían un mensaje **Preempted** al **líder**, conteniendo el ballot que han recibido.

- El **líder** recibe el mensaje y cambia su estado de **activo** a **inactivo**. A partir de este momento el **líder** realiza una monitorización del **líder** que es dueño del ballot actual. Cuando el **líder** detecta que el **líder** dueño del ballot ha abandonado el sistema, o ha pasado un intervalo de tiempo t, aumenta el identificador de su propio ballot en una unidad por encima del ballot recibido y procede a adoptar este nuevo ballot. De esta forma se asegura que:

  - En todo momento existe un **líder** proponiendo y decidiendo operaciones, y en el caso de que dicho **líder** abandone el sistema, un nuevo **líder** recogerá su legado en el menor tiempo posible.
  - Se realiza un balanceo de la carga en las decisiones de las operaciones. Al intentar adoptar un nuevo ballot después de transcurrido un

tiempo t, se asegura que todos los líderes del sistema en algún momento se habrán encontrado activos y no se encontrarán siempre en un estado de inactividad o espera.

Paso 2: Proponiendo una petición Cuando una **réplica** recibe un mensaje *REQUEST* con los datos del cliente y la operación a realizar, comprueba en primer lugar si la operación a realizar es de tipo escritura o lectura.

Si la **operación es de tipo lectura** no existe una necesidad de ejecutar el algoritmo de consenso para ordenar la petición, esta es ejecutada inmediatamente.

Si la **operación es de tipo escritura** se ha de comprobar si la operación ya ha sido decidida o procesada dentro de la **réplica**. Si ha sido decidida o procesada entonces se envía al cliente un mensaje *DECIDED*, indicando que la operación se encuentra duplicada y ya se ha procesado anteriormente.

En caso de que la operación no se encuentre decidida o procesada, se procede a introducir a la operación en la lista de operaciones a proponer.

Para proponer una operación, en primer lugar se debe decidir un slot al cual se debe proponer, dicho slot es elegido escogiendo el menor slot libre entre los slots de operaciones propuestas y operaciones decididas, así como del slot de la ejecución actual; de forma que una operación se propondrá al menor slot libre siempre y cuando este sea mayor que el slot de la ejecución.

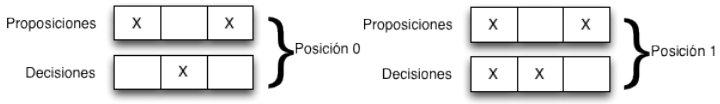


Figura 2.2: Elección Slot

Una vez elegido el slot, se marca la operación como propuesta para dicho slot y se envía un mensaje *PROPOSE* a todos los **líderes** que forman parte del sistema, contenido la operación propuesta y el slot al cual se ha propuesto.

Hay que tener en cuenta que una misma operación puede ser propuesta a distintos slots, y en un slot pueden existir propuestas múltiples operaciones, esto es así debido a que de todas las operaciones propuestas para un slot, únicamente una será decidida, y el resto de las operaciones serán *re-propuestas* a un nuevo slot con el fin de que eventualmente todas ellas se encuentren decididas.

También se ha de tener en cuenta que tanto en la lista de proposiciones como en decisiones pueden existir slots intermedios sin ninguna operación, esto es debido a lo siguiente:

- En **proposiciones** puede existir un slot intermedio libre si, por alguna razón, ha llegado una operación decidida a dicho slot y ha sido ejecutada

antes de que se hubiera propuesto una operación en dicha **réplica**. En este caso las operaciones enviadas por el cliente para proponer serán enviadas a un slot superior, quedando este sin candidatos a proponer. Dicho comportamiento adquiere sentido debido a que la proposición de operaciones a un slot ya decidido produciría la *re-proposición* de las operaciones, conllevando a un mayor intercambio de mensajes y a un mayor coste temporal para la decisión de la operación.

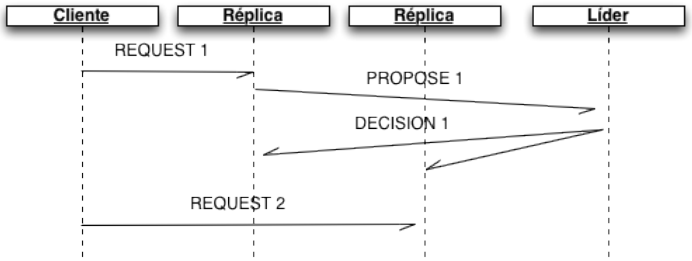


Figura 2.3: Decisión antes de proposición

- En **decisiones** ha llegado la notificación de una operación i pero no ha llegado la notificación de una operación $i-1$. En este caso las operaciones enviadas por el cliente serían propuestas al slot $i-1$, y si ya ha sido decidida la operación, el **líder** retornará la operación que ha sido decidida permitiendo continuar con el algoritmo de decisión y procesamiento.

Paso 3: Decidiendo una petición Cuando un **líder** recibe un mensaje *PROPOSE* por parte de una **réplica**, comprueba si tanto el slot como la operación ya han sido decididos. Si la operación ya ha sido decidida el mensaje es ignorado. Si el slot ya ha sido decidido, entonces envía un mensaje *DECISION* a todas las **réplicas** del sistema, conteniendo el slot decidido y la operación asociada, la cual puede no ser la misma que la enviada en el mensaje *PROPOSE*. De esta forma se asegura que un slot que ha sido decidido con una operación no sea modificado y siempre se mantenga con dicha operación. [3]

[3] Debido a que se ejecuta el recolector de basura, esto solamente sucede si no se ha ejecutado el recolector de basura para dicho slot. En caso de que se hubiera ejecutado se iniciará el protocolo de recuperación.

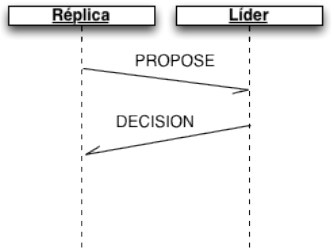


Figura 2.4: Respuesta PROPOSE a un slot ya decidido

En caso de que ni el slot ni la operación hayan sido decididos anteriormente, se introduce la operación en la lista de operaciones propuestas, y si el **líder** es el dueño del ballot actual se procede a inicializar la votación para decidir la operación.

- El **líder** crea una nueva instancia de la clase **commander**, al cual se pasa como parámetros el ballot del **líder** y la operación a votar.
- El **commander** genera un mensaje *P2A* y lo envía a todos los **aceptores** conocidos del sistema. Dicho mensaje contiene el ballot y la operación a votar.
- El **aceptor** recibe el mensaje *P2A*.
  - Comprueba el ballot recibido con el propio, si el recibido es igual o mayor al propio actualiza el ballot y añade la operación a la lista de operaciones aceptadas.
  - Envía un mensaje de respuesta *P2B* al **líder** conteniendo el ballot actual del **aceptor**.
- El **commander** recibe los mensajes *P2B* enviados por los **aceptores**, si alguno de los mensajes contiene un ballot diferente al propio, se supone que se ha producido un cambio de vista y se envía un mensaje *PREEMPTED* al líder.

  En caso de que el **commander** reciba un mensaje *P2B* de $(n+1)/2$ **aceptores** con el mismo ballot que el propio; el **commander** tendrá la certeza de que la mayoría de los **aceptores** han aceptado y conocen la operación votada, y por lo tanto se puede dar la operación por **decidida**. En este momento se envía un mensaje *DECISION* a todas las réplicas conocidas del sistema, conteniendo el slot decidido y la operación vinculada al mismo.
- El **líder** elimina la operación de la lista de operaciones propuestas y se marca el slot como **decidido**.

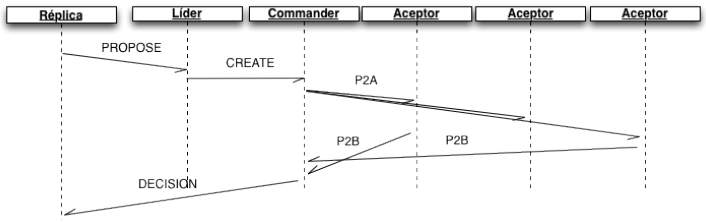


Figura 2.5: Decisión de una operación

Paso 4: Procesando una petición Cuando una **réplica** recibe un mensaje *DECISION* procede a iniciar el algoritmo de procesamiento de operaciones, para ello realiza lo siguiente:

- Comprueba que el slot para el cual se ha recibido la operación no ha sido ya decidido ni procesado, en dicho caso introduce la operación en la lista de operaciones decidas y marca dicho slot como decidido. A su vez si el slot recibido se corresponde con el siguiente slot candidato para decisiones, este se actualiza hasta que apunte a un slot que no contenga ninguna decisión. Este paso es importante para asegurar que las nuevas operaciones se propongan para el slot apropiado.

- Comienza a procesar todas las operaciones que se encuentran decididas a partir del slot actual de procesamiento. Dicho slot no tiene porque coincidir con el slot recibido en el mensaje de decisión, sino que es el último slot para el cual se ha procesado una operación dentro de la **réplica**.

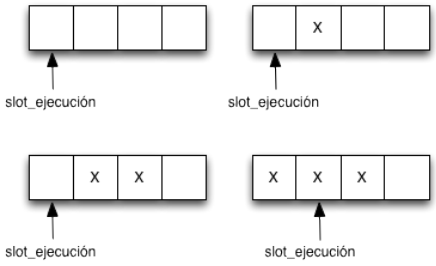


Figura 2.6: Progresión del slot de procesamiento

Por cada operación que se encuentra decidida:

- Si existen operaciones propuestas para el slot decidido, se *re-proponen* para un nuevo slot y se eliminan todas las operaciones vinculadas a dicho slot. Si no existe una operación decidida para dicho slot se finaliza la ejecución.

- Si por algún motivo, la operación que se ha decidido ya se encuentra procesada, se aumenta el valor del slot para el cual se ha de procesar la operación y se continua el algoritmo con el siguiente slot.
- Se aumenta en una unidad el valor del slot para el cual se ha de procesar la operación, se procesa la operación enviando la misma al programa vinculado a la réplica.
- Una vez que se ha recibido la confirmación del procesamiento, se elimina la operación de la lista de decisiones y se añade a la lista de operaciones procesadas, finalmente se envía una respuesta al cliente con el resultado de la operación.

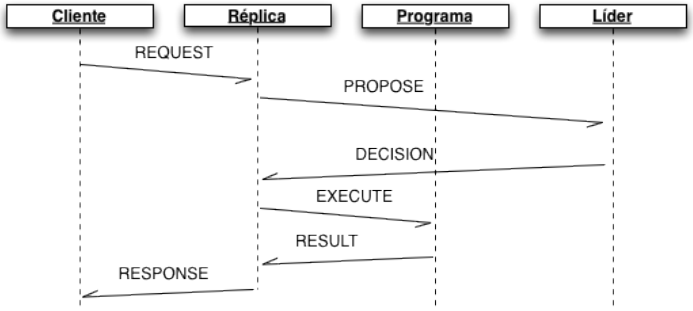


Figura 2.7: Procesamiento de una operación

Capítulo 3

Características del Sistema

3.1. Características

- El sistema se encuentra implementado sobre **CoffeeScript**[1], y utiliza la librería **ZeroMQ**[2] para realizar la comunicación entre todos los procesos del sistema.
- La comunicación del sistema se encuentra optimizada para su ejecución dentro de clusters, si bien es totalmente factible el despliegue del sistema en otro tipo de entornos.
- Las aplicaciones a replicar se pueden encontrar implementadas en cualquier lenguaje de programación. Dado que la comunicación entre la aplicación y la **réplica** se puede realizar a partir de **ZeroMQ**[2], únicamente es necesario que exista un *wrapper* de la librería para dicho lenguaje.
- El sistema se encuentra formado por tres roles diferentes, implementando cada uno de ellos un comportamiento específico. Para el correcto funcionamiento del sistema y que este ofrezca alta disponibilidad, es necesario que al menos un número f de instancias de cada proceso se encuentre en funcionamiento, donde f es el número de instancias que se permiten fallar. El valor de f es dependiente de cada uno de roles, dependiendo su valor del tipo de fallos soportado en cada uno de ellos. El tipo de fallo soportado en función de cada uno de los roles es:
  - **Réplicas**, **fallos caída-parada**. El mínimo número de **réplicas** que se deben encontrar en el sistema debe de ser un $n \geq f + 1$.
  - **Líderes** , **fallos caída-parada**. El mínimo número de **líderes** que se deben encontrar en el sistema debe de ser un $n \geq f + 1$.
  - **Aceptores**, **fallos bizantinos**[7]. Debido a que se ha de asegurar que siempre se encuentre en funcionamiento una mayoría de los mismos, tanto para conseguir el consenso como para asegurar la progresión del algoritmo, el mínimo número de **aceptores** que se deben encontrar en el sistema debe de ser un $n \geq 2f + 1$.

  Por lo tanto, suponiendo que se soportan para todos los roles un $f = 1$, el mínimo número total de procesos necesarios para que el sistema se pueda ejecutar, ofreciendo alta disponibilidad es de 5 procesos , siendo 2 procesos los referentes a las **réplicas** y los **líderes**; y 3 procesos para los **aceptores**.
- La decisión de una operación se realiza en término medio utilizando aproximadamente $(3 * A/2) + R(1 + L)$ mensajes , donde A es el número de **aceptores**, L el número de **líderes** y R el número de **réplicas**. Suponiendo que $A = L = R = 1$ el número medio de mensajes necesarios para decidir una operación sería **4**.
- La decisión de una operación necesita de $(3 * A) + 2 + R(1 + L)$ mensajes en el peor de los casos. Este caso se produce cuando el **líder** ha de adoptar un nuevo ballot.

- El sistema es capaz de realizar el consenso de una operación y decidir cual es su posición de ejecución en un intervalo de tiempo mínimo comprendido entre 1 milisegundo y 5 milisegundos. A estos valores hay que añadir posteriormente el tamaño del mensaje y el coste temporal en transferir dicho mensajes a través de la red. Tal como se muestra a continuación los tiempos en realizar la aceptación de una operación son despreciables, siendo los tiempos que marcan la cota mínima las operaciones realizadas por el **líder** y la **réplica**.

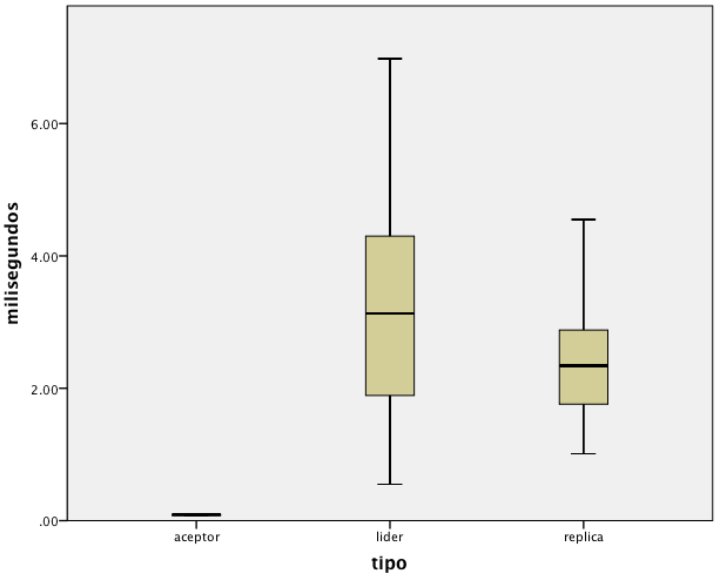
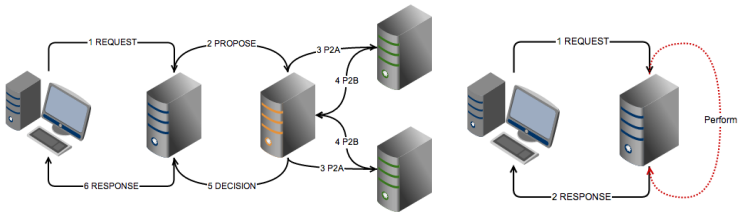


Figura 3.1: Tiempo de resolución de las peticiones

- El sistema puede **crear nuevas instancias de manera dinámica**, ya sea con el fin de aumentar la capacidad de procesamiento de peticiones recibidas, o de asegurar la continuidad del algoritmo. Si el sistema detecta que el número de instancias es inferior al necesario para asegurar la alta disponibilidad, este puede crear nuevas instancias de los roles necesarios para que el sistema pueda seguir funcionando correctamente.

- En el sistema se encuentran implementadas **operaciones de lectura y escritura**. Las operaciones indicadas para escritura han de ejecutar completamente el algoritmo de consenso. Las operaciones de lectura son ejecutadas directamente en las réplicas, disminuyendo notablemente el tiempo de respuesta de las mismas. El uso de estos dos tipos de operaciones hace que el sistema tenga una **consistencia secuencial**[4]. Si se realizan todas las operaciones en modo escritura, obviando las operaciones de lectura, se consigue una **consistencia estricta**.



(a) (b)

Figura 3.2: (a)Mensaje de escritura.(b) Mensaje de lectura

- El sistema soporta particiones dentro de los roles **réplicas** y **líderes**. El sistema puede alcanzar distintas particiones de ambos roles, siempre y cuando se cumplan alguna de las siguientes condiciones:

  - Todos los **líderes** conocen a todos los **aceptores** del sistema, de esta forma los líderes recuperaran las operaciones no recibidas cuando adopten un ballot.

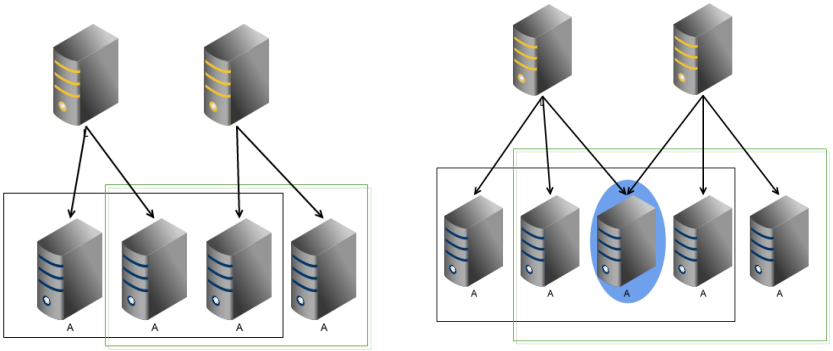
  - Los **líderes** conocen a un subconjunto de los **aceptores**, siempre y cuando se cumplan las siguientes condiciones.

    - Los **líderes** conocen a un subconjunto de los **aceptores** tal que dicho conjunto es igual o mayor que la mayoría del conjunto.

    - El número de **aceptores** total en el sistema es impar.

    La razón para cumplir estas condiciones son las siguientes: Si cada **líder** conoce a una mayoría de los **aceptores** la intersección de dichos conjuntos no podrá ser el conjunto vacío, por lo que en teoría al menos un subconjunto de los **aceptores** habrán aprendido todas las operaciones del sistema.

    Aún así, si el conjunto total es par puede darse el caso de que exista algún **aceptor** perteneciente a la intersección que no reciba todas las operaciones enviadas por el conjunto de los **líderes**, dificultando de esta forma el mantenimiento del estado global del sistema y la recuperación del mismo, si se utiliza un conjunto impar de **aceptores** este problema queda resuelto.



(a) (b)

Figura 3.3: (a)Conjunto aceptores par.(b) Conjunto aceptores impar

- Aunque existan particiones dentro del sistema, eventualmente todas las **réplicas** alcanzarán el mismo estado, hayan o no hayan recibido todas las operaciones enviadas por los clientes. Con que una réplica haya recibido y procesado una operación, el resto de réplicas procesarán eventualmente dicha operación.
- Se implementa un algoritmo de **recolección de basura** en los roles de **líder** y **aceptor**. De esta forma se realiza una mejor gestión de la memoria, eliminando operaciones que ya no sean utilizadas por el sistema, mejorando el rendimiento de los procesos y disminuyendo las necesidades computacionales.
  - Una vez que todos los **líderes** han recibido una operación, esta es eliminada del **aceptor**.
  - Una vez que un **líder** envía un mensaje a las **réplicas** indicando que una operación ha sido decidida, elimina dicha operación de su memoria.
- Se implementa un **algoritmo de recuperación** en todos los roles. De esta manera cuando una nueva instancia de un rol se conecta al sistema, este puede eventualmente alcanzar el estado del resto de los procesos y continuar de manera normal su ejecución.
  - Un **aceptor** recibe el estado de $(n/2) + 1$ **aceptores** del sistema, realiza una unión de todos los estados recibidos y hace del mismo su estado propio.
  - Un **líder** actualiza su estado recibiendo el mensaje de adopción de ballot por parte de los **aceptores**. En cada adopción de ballot, el **aceptor** envía todas las operaciones que contiene, el **líder** realiza la unión de estas operaciones y actualiza su estado.

- Una **réplica** realiza la recuperación de dos formas diferentes:
  - Si los **líderes** contienen las operaciones que solicita la **réplica**, estos envían las operaciones a la **réplica** para que actualice su estado.
  - En caso contrario la **réplica** recibe el estado del resto de las **réplicas**, una vez que ha recibido al menos una operación, actualiza su estado y finaliza el algoritmo.

  La **réplica** actualiza su estado recibiendo y ejecutando cada una de las operaciones que han sido decididas y ejecutadas en el resto de las **réplicas** que forman parte del sistema. Aún así, sería posible enviar el estado final de las **réplicas** con el fin de disminuir tanto el tiempo de recuperación como el tamaño y cantidad de los mensajes enviados en el algoritmo.

- Actualmente no se encuentra implementada ninguna medida de seguridad, tanto para asegurar la integridad y confidencialidad de los mensajes, como para asegurar que los procesos que se encuentran ejecutando dentro del sistema son procesos conocidos, y válidos para encontrarse ejecutando dentro del mismo.

Capítulo 4

Comunicación

En este capítulo se describe cual es la comunicación utilizada dentro del sistema, realizando una descripción de la implementación realizada y de los mensajes utilizados en la misma.

En primer lugar se realiza un resumen del flujo de comunicación entre los procesos, es decir, el intercambio o flujo de mensajes que se producen en la comunicación entre los diferentes roles para realizar una determinada acción. A continuación se describen las características propias de la comunicación del sistema y la implementación realizada de la misma. Por último se realiza una descripción de los tipos de mensajes existentes.

4.1. Flujo de Mensajes

En esta sección se muestra el flujo de mensajes, utilizando diagramas de actividad, entre los diferentes procesos.

4.1.1. Adopción de un ballot

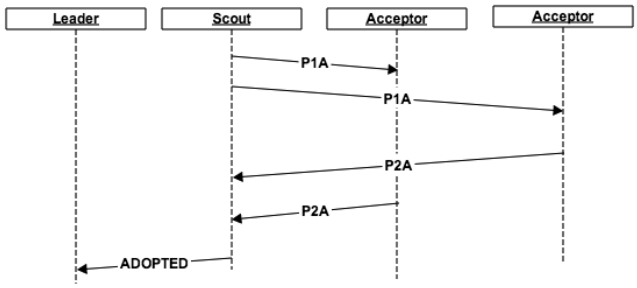


Figura 4.1: Adopción de un ballot

4.1.2. Preempted en adopción de un ballot

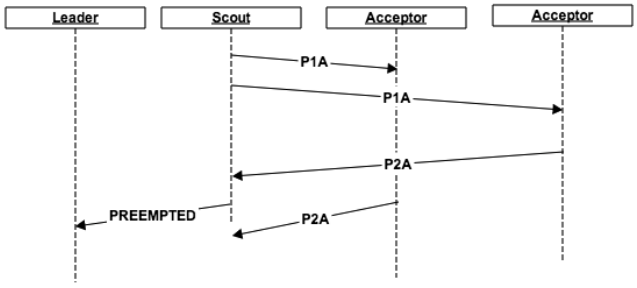


Figura 4.2: Preemted de una adopción

4.1.3. Decisión de una operación

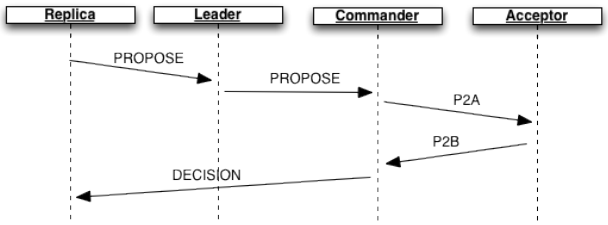


Figura 4.3: Decisión de una operación

4.1.4. Preempted en la decisión de una operación

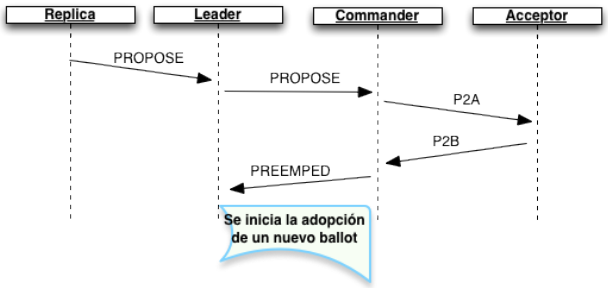


Figura 4.4: Preemted de una decisión

4.1.5. Recuperación aceptor

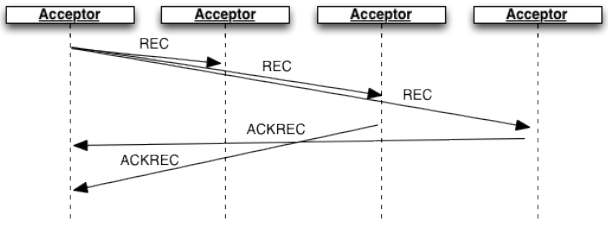


Figura 4.5: Recuperación de un aceptor

4.1.6. Recuperación réplica

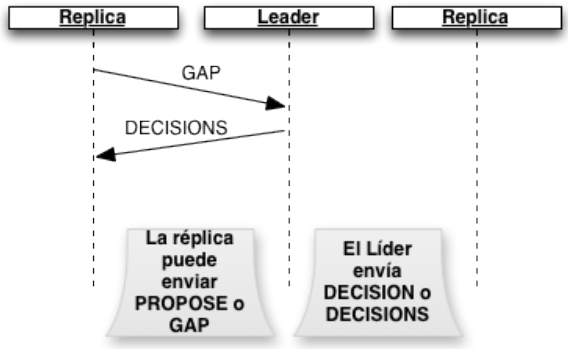


Figura 4.6: Recuperación de una réplica mediante el líder

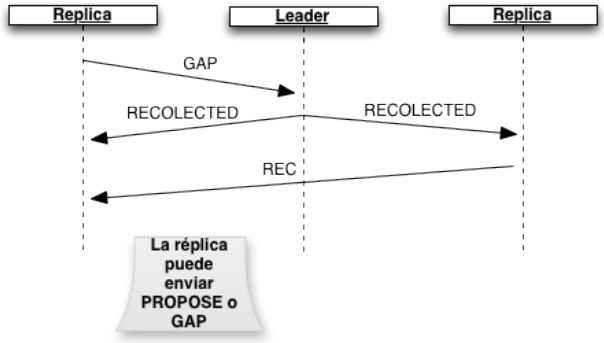


Figura 4.7: Recuperación de una réplica mediante otra réplica

4.2. Características

La comunicación entre los diferentes nodos que conforman el sistema contiene las siguientes características:

- Comunicación asíncrona, no existe una cota superior temporal en la cual el mensaje deba ser entregado [1].

[1] En la implementación final esto no es del todo cierto, se utiliza un timeout para impedir que el sistema se encuentre indefinidamente esperando por una respuesta.

- La comunicación es segura, es decir, un mensaje enviado es eventualmente recibido por el proceso receptor. Aún así, no se asegura que los mensajes lleguen al receptor en el orden de emisión.

- Los mensajes se pueden enviar y recibir de manera duplicada.

- Los mensajes no se corrompen.

La comunicación más común dentro del sistema es la **comunicación uno a muchos** y muchos a uno. Dada la naturaleza de los roles que forman parte del sistema, no es necesario que dicha comunicación entre los procesos sea fiable, es decir, no es necesario el asegurar que un mensaje llegue a todos los procesos a los cuales ha sido enviado.

Las comunicaciones realizadas entre los diferentes procesos del sistema son las siguientes:

- Un cliente se comunica con un conjunto ($1 \leq n \geq total\_réplicas$) de **réplicas**, enviando a las mismas una petición. El cliente espera como máximo una única respuesta, aceptado la primera que recibe y desechando el resto. Esto permite que el sistema se mantenga funcionando con al menos una **réplica** funcional, soportando **fallos caida-parada**. En caso de que se decida dar soporte a **fallos bizantinos** por parte de las réplicas, el cliente esperaría a recibir $(n/2) + 1$ respuestas idénticas para dar por resuelta la petición.

- Cada **réplica** envía un mensaje a un conjunto de n **líderes**, tal que $1 \leq n \geq número\_líderes$. La **réplica** espera como máximo una respuesta por cada petición, desechando el resto de peticiones. Esto permite que el sistema se mantenga funcionando con al menos un **líder** funcional.

- Cada **líder** envía un mensaje a un conjunto de **aceptores**. El **líder** necesita recibir confirmación de la mayoría de los **aceptores** para continuar, de esta forma el sistema se mantendrá funcionando mientras se encuentren en ejecución un número de **aceptores** superior a $(n/2) + 1$.

4.3. Implementación

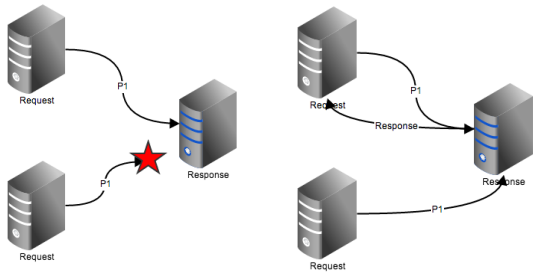
Con el fin de cumplir las características descritas anteriormente, para la implementación de la comunicación se ha utilizado una librería orientada a comunicación mediante mensajes llamada **ZeroMQ**[2]. Dicha librería ofrece un conjunto de patrones de comunicación , los cuales facilitan la creación de los protocolos de comunicación utilizados en el sistema.

Los patrones utilizados según la comunicación necesaria entre los diferentes roles son los siguientes:

- **Patrón Request-Response**: Es el patrón utilizado por el algoritmo de recuperación entre las **réplicas**, y también para la comunicación de la **réplica** con el programa encargado de procesar las operaciones [2]. Este patrón ofrece una **comunicación síncrona y unidireccional**.

  La parte **Request** únicamente puede enviar peticiones, no pudiendo enviar una nueva petición hasta que no ha recibido una respuesta a la petición anterior.

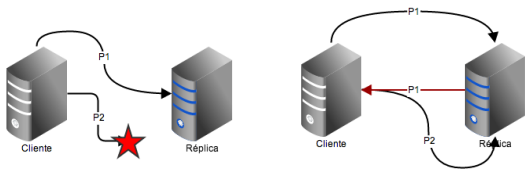
  La parte **Response** únicamente recibe peticiones, no pudiendo recibir una nueva petición hasta que no ha respondido a la anterior.



(a) (b)

Figura 4.8: (a)No procesa la segunda petición.(b) Envía respuesta y procesa la nueva petición

- **Patrón Request-Router**: Es el patrón utilizado para la comunicación entre los **clientes** y las **réplicas**. Este patrón ofrece una **comunicación parcialmente asíncrona**. El cliente implementa la parte **Request**, envía un mensaje al **router** y suspende su ejecución hasta recibir una respuesta a su petición, de esta forma se asegura que el cliente no procese peticiones fuera del orden preestablecido.



(a) (b)

Figura 4.9: (a)Se impide enviar sin recibir respuesta previa.(b) Envío correcto

[2]Únicamente en el caso de que se utilice la comunicación por mensajes y no mediante código

La **réplica** implementa el **router**. Un **router** puede recibir n operaciones y no es necesario responder a las mismas en el orden recibido o responder a las mismas; de esta forma una **réplica** puede recibir varias peticiones simultáneamente y responder a las mismas según se decidan dentro del sistema.

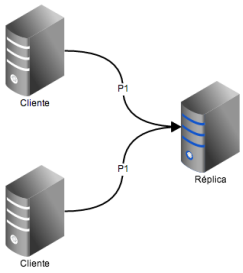


Figura 4.10: Réplica puede recibir múltiples peticiones

- **Patrón Dealer-Router**[11]: Es el patrón utilizado para realizar el algoritmo de recuperación entre los **aceptores**, ofrece una **comunicación asíncrona**. El **dealer** es el encargado de enviar mensajes al **router**, una vez enviado un mensaje puede seguir su ejecución sin esperar a recibir una respuesta. El **router** tal como se ha dicho anteriormente, puede recibir múltiples peticiones sin tener que responder anteriormente a ninguna de ellas para poder recibir la siguiente.

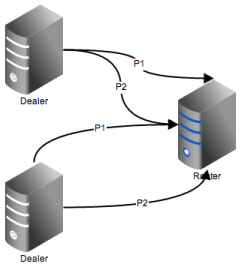


Figura 4.11: Dealer y router pueden enviar y recibir múltiples mensajes

- **Patrón Productor-Consumidor**[12]: Es el patrón utilizado para realizar las comunicaciones entre todos los nodos, ofreciendo una comunicación uno a muchos, segura y asíncrona. Los consumidores se conectan a los productores y se subscriben a un tipo de mensaje en concreto, el productor envía un mensaje de un tipo y es recibido por todos los consumidores que se han subscrito al mismo.

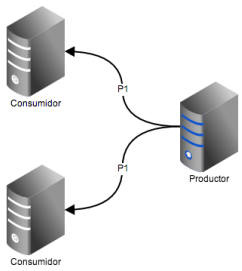


Figura 4.12: Comunicación del publicador con los consumidores

Tal como se ha dicho anteriormente, el patrón utilizado para la mayoría de las comunicaciones realizadas dentro del sistema es el patrón **productor-consumidor**. Los beneficios de usar dicho patrón, en contra de otro patrón asíncrono y que permite comunicación uno-a-muchos como el **dealer-router**, son los siguientes:

- El servidor o productor no ha de conocer en todo momento que procesos se encuentran en el sistema para enviar mensajes a los mismos. Los consumidores se conectan al servidor, y realizan una subscripción a un tipo de mensaje específico. El productor envía un mensaje y este será enviado de forma transparente a todos los consumidores que se encuentran subscritos.

- Un cliente se puede conectar y subscribir a un líder incluso cuando este no se encuentre disponible. En el momento que el productor se encuentre disponible el consumidor empezará a recibir los mensajes automáticamente. De esta forma se evita que el consumidor tenga que realizar una monitorización para conocer cuando el productor se encuentra disponible y realizar la conexión al mismo.

Además de las ventajas descritas, en el patrón **productor-consumidor** existe otra ventaja, el consumo del ancho de banda en la comunicación. Debido a la implementación del patrón realizada en **ZeroMQ**, la comunicación utilizando dicho patrón es realmente eficiente en comparación con el resto de patrones implementados por dicha librería.

En la gráfica que se muestra a continuación, se ha realizado el envío de 5000 mensajes con una cadena de texto, enviando los mismos de manera síncrona. En la gráfica se muestran el número total de bytes transmitidos en cada uno de los patrones utilizados dentro del sistema.

Como se puede observar, el número de bytes transmitidos utilizando el patrón **productor-consumidor** es notablemente inferior al utilizado por el resto de patrones analizados.

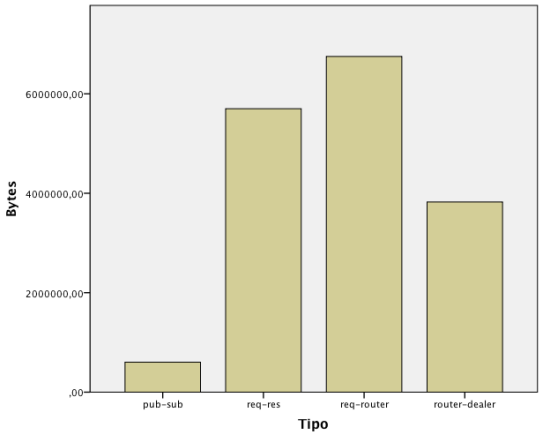


Figura 4.13: Consumo de bytes diferentes patrones de comunicación

4.4. Tipos de Mensaje

Todos los mensajes utilizados en Paxos contienen la siguiente estructura:

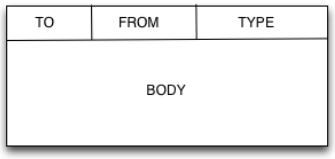


Figura 4.14: Formato Mensaje

Donde:

- **To**: Dirección del nodo al cual va destinado el mensaje.
- **From**: Dirección del nodo que envía el mensaje.
- **Type**: Tipo de mensaje enviado.
- **Body**: El cuerpo del mensaje, es diferente según el tipo del mensaje.

A continuación se realiza una descripción de todos los tipos de mensajes utilizados en el sistema y los campos que contienen.

4.4.1. Request

Mensaje enviado por el cliente a las **réplicas** del sistema. Es el mensaje inicial del algoritmo, conteniendo la acción que desea realizar el cliente. El cuerpo del mensaje contiene los siguientes campos:

- **client_id**: Identificador del cliente. Puede ser un nombre único, una IP, etc.
- **operation_id**: Identificador de la operación local del cliente. Sigue el orden secuencial de operaciones ejecutadas por el cliente.
- **type**: Tipo de la operación. Indica si la operación es de solo lectura **READ** o de escritura **WRITE**.
- **operation**: La operación que va a ser procesada en el sistema. El formato de la misma depende del programa que se esté ejecutando, siendo independiente del sistema de replicación.

4.4.2. Response

Mensaje enviado por las **réplicas** a los clientes como respuesta a los mensajes *REQUEST* enviados por estos. El mensaje es enviado al cliente una vez que la operación ha sido decidida y procesada por la réplica. El cuerpo del mensaje contiene los siguientes campos:

- **client_id**: Identificador del cliente.
- **operation_id**: Identificador de la operación.
- **result**: Resultado de la ejecución de la operación.

4.4.3. Propose

Mensaje enviado por las **réplicas** del sistema a los **líderes**. Envía un mensaje de proposición de una operación sobre un slot dado.

- **slot**: Slot sobre el que se desea realizar la proposición.
- **operation**: Operación que se desea proponer.
- **recPort**: Puerto de escucha de la réplica mediante el cual se lleva a cabo el algoritmo de recuperación.

4.4.4. Recolected

Mensaje enviado por un **líder** al conjunto de **réplicas** cuando, al recibir un mensaje *GAP*, no es capaz de resolver las peticiones solicitadas.

- **slot_min**: Slot mínimo a partir del cual la réplica solicita la recuperación.
- **slot_max**: Slot máximo hasta el cual la réplica solicita la recuperación.
- **replica**: identificador de la **réplica** que ha enviado el mensaje *GAP*.
- **replicaRec**: dirección en la cual la **réplica** se encuentra escuchando para recibir los mensajes de recuperación.

4.4.5. GAP

Mensaje enviado por la **réplica** a los **líderes** para recibir posibles operaciones pendientes que aún no ha recibido.

- **slot_min**: Slot a partir del cual realiza la petición de operaciones.
- **slot_max**: Slot máximo hasta el cual se desea recibir operaciones.
- **rec**: Puerto de escucha utilizado por la **réplica** para el algoritmo de recuperación.

4.4.6. Rec

Mensaje enviado tanto por **réplicas** como **aceptores**. En cada uno de ellos el mensaje tiene un significado diferente, mientras que en los **aceptores** el mensaje indica la petición de un **aceptor** para iniciar el algoritmo de recuperación; en las **réplicas** se trata de la respuesta a un mensaje *GAP*, el cual ha sido reenviado del **líder** a las **réplicas** para que estas respondan a la réplica solicitante.

Réplica

- **performed**: Lista de todas las operaciones ejecutadas en la **réplica**.
- **slotInProposals**: Slot que se encuentra indicado en la variable de dicho nombre.
- **slotInDecisions**: Slot que se encuentra indicado en la variable de dicho nombre.

Aceptor

- **slot**: Slot hasta el cual el aceptor ha ejecutado su recolector de basura y a partir del cual desea recibir las operaciones pendientes.

4.4.7. AckRec

Mensaje enviado como respuesta a un mensaje *Rec* en el algoritmo de recuperación de los **aceptores**. Utilizado para enviar el estado de los diferentes roles en el algoritmo de recuperación.

- **slot**: Valor del slot hasta el cual se ha ejecutado el recolector de basura.
- **ballot**: Ballot actual.
- **pvals**: Lista de todas las operaciones aceptadas.

4.4.8. P1A

Mensaje enviado por el **líder** a los **aceptores**. Se envía para realizar la adopción de un ballot.

- **leader**: Identificador del líder que realiza la petición.
- **ballot**: Ballot que contiene el líder y el cual se desea adoptar.
- **slot**: Slot hasta el cual el líder que envía el mensaje ha ejecutado su recolector de basura.

4.4.9. P1B

Mensaje enviado por los **aceptores** a los **líderes**. Contiene el ballot con el cual se ha realizado la adopción, así como un conjunto de operaciones aceptadas anteriormente.

- **ballot**: Ballot con el cual se realiza la adopción.
- **accepted**: Lista de operaciones aceptadas anteriormente por el aceptor.
- **garbageCollected**: Valor del slot hasta el cual el **aceptor** ha eliminado las operaciones aceptadas.

4.4.10. P2A

Mensaje enviado por el **líder** a los **aceptores**. Contiene la operación que se desea aceptar.

- **operation**: Operación que el líder envía para que sea aceptada.

4.4.11. P2B

Mensaje enviado por el **aceptor** al **líder**. Contiene el ballot con el cual se ha aceptado la operación. Es la respuesta al mensaje P2A.

- **ballot**: Ballot con el cual ha sido aceptada la operación.
- **garbageCollected**: Valor del slot hasta el cual el **aceptor** ha eliminado las operaciones aceptadas.

4.4.12. Adopted

Mensaje enviado por el **scout** al **líder** que lo ha creado. Se produce a continuación de recibir la adopción de un ballot. Contiene el ballot con el cual se ha realizado la adopción y una lista con todos los valores aceptados anteriormente.

- **ballot**: Ballot con el cual se ha realizado la adopción.

- **pvalues**: Lista del conjunto de las operaciones aceptadas por todos los **aceptores** que han respondido al **líder**.
- **garbageCollected**: Valor del slot más alto de todos los recibidos en los mensajes *P2B*.

4.4.13. Decision

Mensaje enviado por el **commander** a todas las **réplicas**. Indica la operación que ha sido decidida para un slot en concreto.

- **slot**: Slot para el cual se ha realizado la decisión.
- **operation**: Operación decidida.

4.4.14. Decisions

Mensaje enviado por un **líder** a una **réplica** como respuesta a un mensaje *GAP* cuando no ha ejecutado su recolector de basura sobre las operaciones solicitadas.

- **mapOfOperations**: Mapa de las operaciones almacenadas en el **líder**, como índice se encuentra el slot y como valor la operación asociada al mismo.

4.4.15. Duplicated

Mensaje enviado por la **réplica** al **cliente** cuando la petición *REQUEST* enviada ya ha sido procesada anteriormente.

- **operacion**: La operación que se encuentra duplicada.

4.4.16. Preempted

Mensaje enviado por el **scout** o el **commander** al detectar que el ballot recibido por los **aceptores** es más moderno al ballot del **líder**.

- **ballot**: Ballot actual utilizado por los aceptores.

Capítulo 5

Descripción de las Clases

En el siguiente capítulo se realiza una descripción de las clases implementadas para la realización del sistema. A lo largo del capítulo se muestran los diagramas UML representativos de las clases, así como una descripción de la funcionalidad de cada una de ellas y los métodos que las componen.

5.1. Server y GCS

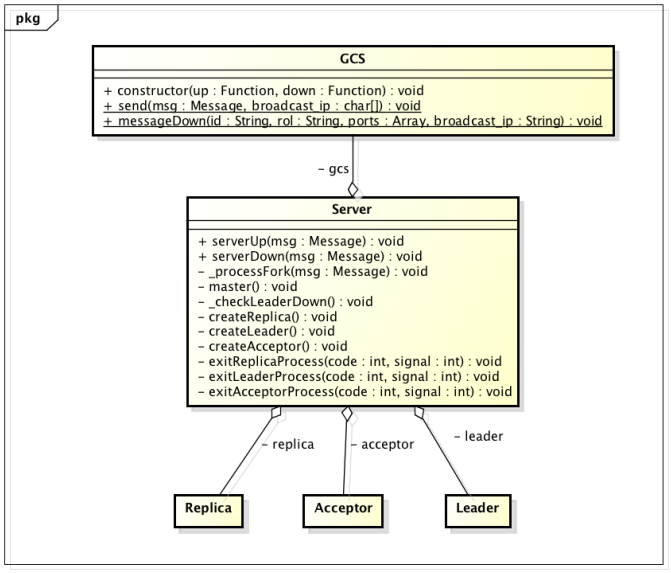


Figura 5.1: Diagrama Clases: Clases GCS y Server

5.1.1. Server

La clase server es la encargada de inicializar y gestionar los diferentes roles que debe ejecutar una instancia del sistema, a su vez la clase server se encarga de monitorizar todos los procesos que se encuentran dentro del sistema, descubriendo de esta forma cuando un nuevo proceso es añadido o eliminado del mismo y actuar en consecuencia.

operaciones

- **constructor** : Inicializa el servidor. Si el proceso servidor es un proceso padre entonces inicializa su funcionamiento por defecto, monitorizando los roles que ejecuta la instancia y los procesos pertenecientes al sistema. Si no es el proceso padre entonces crea un proceso con un rol específico.
- **processFork**: Procesa los mensajes enviados por cada uno de los roles que gestiona el servidor.

- **master** : Crea todos los roles que se encuentran definidos para la instancia y comienza a monitorizar la red y los procesos creados.
- **checkServers**: Comprueba el número de instancias de cada rol que existen dentro del sistema. Si el número de instancia de un rol es inferior al necesario para que el sistema se pueda ejecutar de manera correcta, y la instancia actual no se encuentra ejecutando dicho rol entonces se crea dentro de la instancia.
- **checkLeaderDown**: Comprueba si algún rol **líder** ha abandonado el sistema. En caso de que en un tiempo t no se haya recibido información de alguno de los líderes, se supone que dicho **líder** ha abandonado el sistema. En este momento si la instancia ejecuta el rol de **líder** se le comunica dicho hecho para que proceda a realizar las operaciones oportunas.
- **serverUp**: Se ejecuta cuando el GCS recibe un mensaje de un proceso indicando que se encuentra vivo. Comprueba si el proceso indicando en el mensaje ya se encuentra descubierto, y se notifica a cada uno de los sub-procesos que forman parte de la instancia de que se ha añadido un nuevo proceso al sistema.
- **serverDown**: Se ejecuta cuando el GCS recibe un mensaje de un proceso indicando que dicho proceso ha abandonado el sistema. Se elimina al proceso de la lista de procesos pertenecientes al sistema, y se notifica a cada uno de los sub-procesos que forman parte de la instancia de que se ha eliminado un proceso del sistema.
- **createReplica**: Si no se encuentra creado un proceso **réplica**, crea un nuevo proceso de **réplica**.
- **createAcceptor**: Si no se encuentra creado un proceso **aceptor**, crea un nuevo proceso de **aceptor**.
- **createLeader**: Si no se encuentra creado un proceso **líder**, crea un nuevo proceso de **líder**.
- **exitReplicaProcess**: Envía un mensaje al GCS indicando que el proceso **réplica** ha finalizado y ya no se encuentra disponible en el sistema.

  Si la instancia se encuentra ejecutando en modo *stealh* y se cumplen las condiciones necesarias, entonces se vuelve a inicializar el proceso **réplica**.
- **exitAcceptorProcess**:Envía un mensaje al GCS indicando que el proceso *acceptor* ha finalizado y ya no se encuentra disponible en el sistema.

  Si la instancia se encuentra ejecutando en modo *stealh* y se cumplen las condiciones necesarias, entonces se vuelve a inicializar el proceso **aceptor**.
- **exitLeaderProcess**:Envía un mensaje al GCS indicando que el proceso **líder** ha finalizado y ya no se encuentra disponible en el sistema.

  Si la instancia se encuentra ejecutando en modo *stealh* y se cumplen las condiciones necesarias, entonces se vuelve a inicializar el proceso **líder**.

5.1.2. GCS

GCS es la clase encargada de difundir mediante **Broadcast** los mensajes que indican el estado de cada uno de los procesos que forman parte del sistema. Cada vez que GCS recibe un mensaje, este se lo comunica a la clase Server para que realice las operaciones definidas para dicha acción.

operaciones

- **constructor**: Inicializa el servidor GCS para recibir mensajes utilizando el protocolo UDP. A su vez vincula las operaciones *UP* y *DOWN*, pasadas como parámetros, para ser ejecutadas en el momento que se recibe algún mensaje de este tipo.
- **send**: Envía un mensaje a la dirección de **Broadcast** indicada.
- **messageDown**: Envía un mensaje de tipo *DOWN* a la dirección de Broadcast indicada.

5.2. Rol, Réplica, Aceptor y Líder

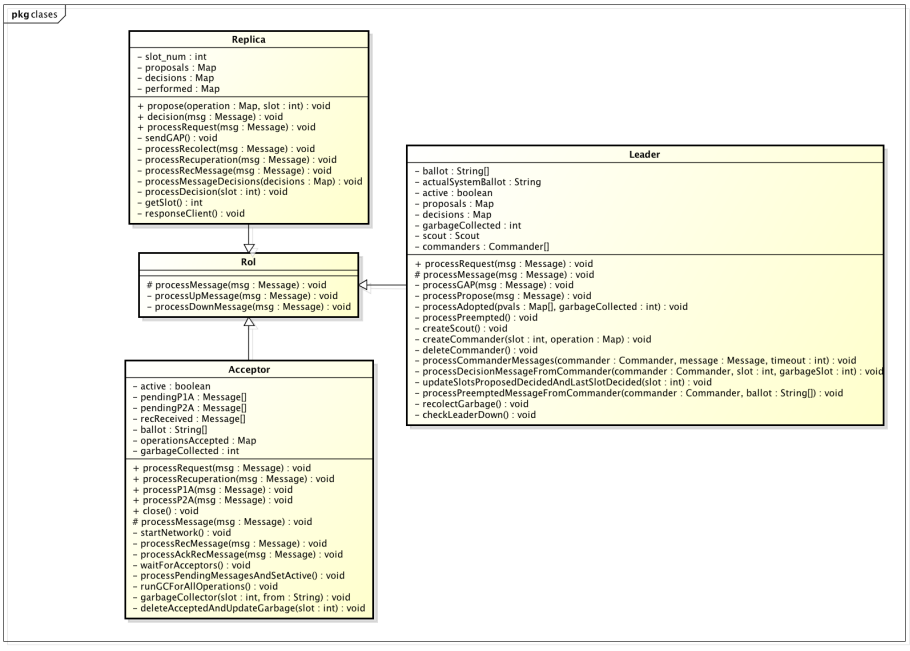


Figura 5.2: Diagrama Clases: Leader, Aceptor y réplica

5.2.1. Rol

Clase base de la cual heredan todos los roles que forman parte del sistema, es decir **réplica**, **aceptor** y **líder**. Contiene el procesamiento de los mensajes *UP* y *DOWN*, utilizados para conocer todas las instancias de los diferentes roles que existen dentro del sistema. Las operaciones que contiene implementadas son las siguientes:

operaciones

- **constuctor**: Inicializa la clase a partir de un fichero de configuración e indica la red a utilizar y a partir de la cual enviará y recibirá mensajes.
- **processMessage**: Procesa los mensajes *UP* y *DOWN* enviados por la clase **Server**. Esta operación será sobrescrita por cada uno de los roles para añadir el procesamiento de los mensajes que cada uno de ellos va a procesar.
- **processUPMessage**: Actualiza la lista de las instancias de cada uno de los roles que forman parte del sistema. Si la instancia que se indica como *UP* no se encuentra en la lista, entonces es añadida a la misma.
- **processDownMessage**:Actualiza la lista de las instancias de cada uno de los roles que forman parte del sistema. Si la instancia que se indica como *DOWN* se encuentra en la lista, entonces es eliminada de la misma.

5.2.2. Réplica

Las **réplicas** son el punto de entrada del sistema. Son los encargados de recibir las peticiones del cliente , proponer dichas peticiones ante los **líderes** y ejecutar las mismas una vez son decididas.

variables

- La réplica contiene tres tablas Hash: **decisions** , **proposals** y **performed**.

  En **decisions** se almacenan aquellas operaciones que han sido decididas, en **proposals** aquellas peticiones que han sido propuestas y en **performed** aquellas operaciones que han sido ejecutadas.

  La clave de las tablas es la instancia sobre la cual se van a decidir y proponer las operaciones (slot). Dado que en un principio se pueden proponer múltiples operaciones para una misma instancia, el valor de **proposals** es una lista de todas las operaciones que han sido propuestas dicha instancia. Tanto en **decisions** como en **performed** el valor será una única operación, la cual no podrá ser modificada.

  Cuando una operación ha sido decidida para una instancia, esta es guardada dentro de la tabla **decisions** y se eliminan todas las operaciones que se encuentran en la tabla **proposals** para dicha instancia. Cuando una

operación ha sido procesada, dicha operación es guardada dentro de la tabla **performed** y es eliminada de la tabla **decisions**.

Una operación puede encontrarse en **decisions** y no encontrarse en **proposals**. Esto es posible debido a que una petición del cliente por algún motivo no pudo llegar a la **réplica**, pero si ha sido procesada por otras **réplicas**, siendo propagado el estado de decisión cuando esta ha sido aceptada por el líder.

operaciones

- **constructor**: Inicializa la instancia de **réplica**. En primer lugar realiza una petición a la clase *server* para que le proporcione aquellos **líderes** conocidos que se encuentran conectados al sistema. A su vez la **réplica** ha de esperar a que la aplicación encargada de procesar las operaciones decididas se encuentre conectada a la **réplica**. Una vez que se tiene constancia de la existencia de la aplicación y se ha recibido el mensaje por parte de la clase *server*, se procede a la inicialización de la red , se modifica el estado de la réplica a **RUNNING** y se envía un mensaje *GAP* a todos los **líderes** conocidos para que, en el caso de que existan, envíen una lista de todas las operaciones ya decididas o inicialicen el protocolo de recuperación.

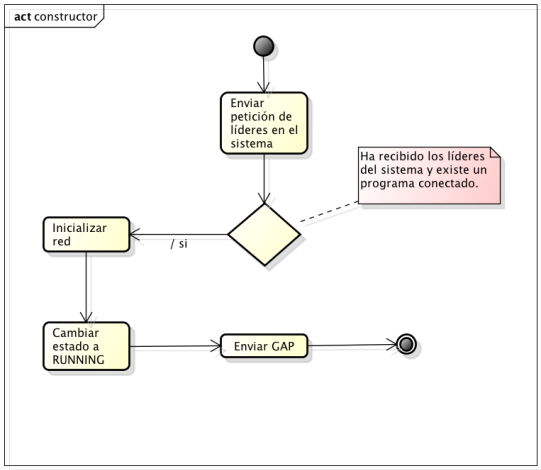


Figura 5.3: Diagrama Actividad: Constructor Réplica

- **sendGAP**: Envía un mensaje *GAP* a todos los **líderes** conocidos en el sistema. Dicho mensaje se envía al inicializar la **réplica** y en un intervalo de tiempo t, tal que $t > d$ donde d es el tiempo transcurrido desde la última recepción de un mensaje *DECISION* o *REC*.

- **processRequest**: Procesa los mensajes recibidos por la **réplica**. Los tipos de mensaje que procesa son: *REQUEST*, *DECISION*, *DECISIONS*, *RECOLECTED* y *RESULT*
- **processRequestMessage**: Procesa un mensaje *REQUEST*. Si el mensaje es de tipo lectura entonces se envía directamente a la aplicación vinculada a la **réplica**, en caso contrario se envía a proponer la petición.
- **processRecolect**: Procesa un mensaje *RECOLECTED*, generando un mensaje de respuesta *REC* hacía la **réplica** que ha realizado la petición de recuperación.

  Una vez que la **réplica** recibe el mensaje comprueba si el mensaje no proviene de si mismo. Si el mensaje proviene de si mismo entonces indica que se encuentra en estado de recuperación y modifica su estado, en caso de no encontrarse modificado anteriormente. A continuación finaliza la ejecución del método.

  Se actualiza el «slot de inicio» de la recuperación indicado por la **réplica**, de esta forma se puede saber si el mensaje de recuperación se refiere a un mensaje antiguo o duplicado y se puede ignorar.

  Se calcula el «slot mínimo» de la recuperación y el «slot máximo». El «slot mínimo» por defecto será el indicado en el mensaje, en caso de que no haya sido indicado se establece a cero. El «slot máximo» será el indicado en el mensaje a excepción del caso de que no sea indicado o sea superior al último slot procesado dentro de la **réplica**, en dichos casos será actualizado al último slot procesado.

  Se genera un array con todas las peticiones procesadas que se encuentran entre el «slot mínimo» y el «slot máximo» y se envía en un mensaje *REC* a la **réplica**.

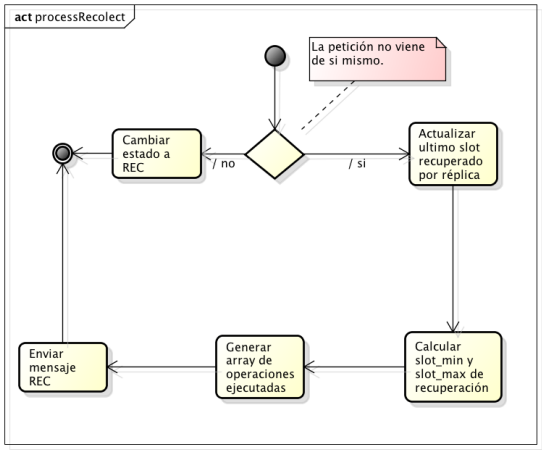


Figura 5.4: Diagrama Actividad: ProcessRecolect

- **processRecuperation**: Procesa un mensaje *REC*. Si se recibe un mensaje de dicho tipo y el estado de la **réplica** es **RUNNING** o **REC**, se procesa el mensaje llamando al método **processRecMessage**, en caso contrario se ignora.

- **processRecMessage**: Procesa un mensaje *REC*.

  Actualiza el estado de la réplica a **REC2** para indicar que se encuentra en proceso de actualización. En este estado, si recibe algún otro mensaje *REC* o *DECISIONS* serán ignorados.

  Actualiza los valores de *slotInProposals* y *slotInDecisions* con los valores máximos entre los actuales y los recibidos en el mensaje.

  Recorre cada una de las operaciones que se han enviado dentro del mensaje. Si el slot al que pertenece la operación ya ha sido procesado entonces se ignora la operación, en caso contrario se ejecuta la operación y se actualiza el valor de *slot_num*.

  Una vez finalizado el procesamiento de todas las operaciones se cambia el estado de la **réplica** a **RUNNING**.

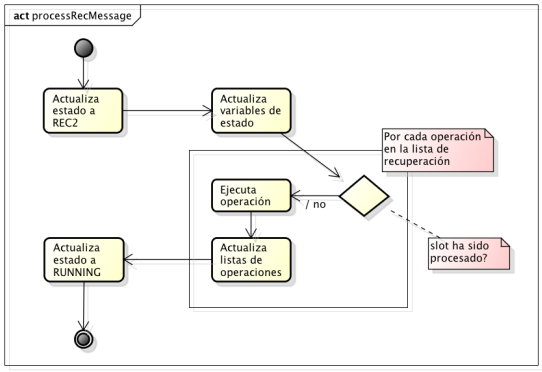


Figura 5.5: Diagrama Actividad: ProcessRecMessage

- **propose**: Propone en el sistema la petición recibida por el cliente.

  Si la petición enviada se encuentra entre las peticiones decididas o procesadas, se envía al cliente un mensaje *DECIDED* indicando que la operación que desea proponer ya ha sido propuesta anteriormente.

  Si no se cumple dicha condición se busca un slot candidato, y se introduce la operación en la lista de *proposals* con dicho slot.

  Se envía un mensaje *PROPOSE* a todos los **líderes** conocidos del sistema, con la operación y el slot propuesto.

- **processMessageDecisions**: Procesa un mensaje de tipo *DECISIONS* enviando por los **líderes** en el proceso re recuperación de la **réplica**.

Si el estado de la réplica es **RUNNING** y la lista de decisiones del mensaje es superior a cero se procede a la recuperación, se modifica el estado de la réplica a *REC*.

Por cada operación indicada dentro del mensaje se llama al método *DECISION*, las operaciones serán procesadas siguiendo el algoritmo de decisión.

Una vez finalizado el procesamiento de todas las operaciones se modifica el estado de la **réplica** a **RUNNING**.

- **decision**: Procesa un mensaje de tipo *DECISION*.

  Comprueba si la operación recibida existe en la lista de operaciones decididas o si ya ha sido procesada, también comprueba si el slot para el cual ha sido decida la operación ya ha sido decidido anteriormente. Si estas condiciones no se cumplen, se introduce la operación en la lista de operaciones decididas y se marca el slot indicado en el mensaje como decidido.

  Si el slot indicado en el mensaje es igual al *slotInDecisions* de la **réplica**, se aumenta en una unidad para indicar cual es el nuevo slot esperado para recibir una decisión.

  Se llama al método **processDecision**.

- **processDecision**: Procesa una decisión perteneciente a un slot indicado.

  Se comprueba si existe una operación en el slot indicado, de no ser así finaliza la ejecución del método.

  Si existen operaciones propuestas para el slot que se encuenta en decisión, entonces se realiza una nueva proposición de estas operaciones para que sean decididas en un nuevo slot.

  Si la operación ya se ha ejecutado anteriormente se aumenta el valor de *slot_num*. Si la operación no se ha ejecutado se añade la operación a la lista de operaciones ejecutadas. A continuación se elimina de la lista de operaciones en decisiones y se eliminan todas las operaciones propuestas para dicho slot, re-proponiendo dichas operaciones a un nuevo slot.

  Se ejecuta la operación en la aplicación vinculada a la **réplica**, se aumenta el valor de *slot_num* y se envía la respuesta al cliente.

  Se procede a realizar una nueva ejecución del método **processDecision** indicando como slot el *slot_num* actual.

- **getSlot**: Retorna el valor del slot para el cual una operación debe de ser propuesta. El valor de dicho slot será el mínimo entre los valores de *slotInProposals* y *slotInDecisions*

- **responseClient**: Crea un mensaje *RESPONSE* y lo envía al cliente.

5.2.3. Acceptor

Los **aceptores** son los roles encargados de votar si una operación es aceptada o no dentro de una instancia de Paxos (slot). Para ello se realiza una votación entre todos los aceptores del sistema, coordinada por un **líder**.

El propósito del **aceptor** es decidir si una operación es aceptada o no, y almacenar dichas operaciones como un histórico y facilitar la recuperación del sistema.

Cada **aceptor** tiene como estado el ballot a partir del cual acepta operaciones, y una lista de todas las operaciones que ha aceptado.

operaciones

- **constructor**: Inicializa la instancia de **aceptor**. En la inicialización de la instancia se llamada al método **startNetwork** y se inicia el recolector de basura de forma que se ejecute automáticamente en intervalos de tiempo t.

- **startNetwork**: Inicializa la red del **aceptor** de modo que pueda recibir y enviar mensajes a los **líderes**, además de permitir la comunicación con el resto de los **aceptores** para realizar el algoritmo de recuperación.

  Antes de inicializar la red se envía un mensaje al proceso **server** para que envíe una lista de todos los **líderes** conocidos dentro del sistema. Una vez que se ha recibido dicho mensaje se procede al inicio de la red y se llama al método **waitForAcceptors**.

- **waitForAcceptors**: Dicho método es llamado en la inicialización del **aceptor**, es utilizado para inicializar la recuperación de su estado.

  El **aceptor** se mantiene a la espera durante un tiempo t para descubrir al resto de los **aceptores** que forman parte del sistema. Si durante ese intervalo de tiempo no se descubren nuevos **aceptores**, se supone que se encuentra solo en el sistema y comienza una ejecución normal.

  En caso de que se descubran otros **aceptores** se procede a inicializar el algoritmo de recuperación.

  Una vez finalizado el algoritmo de descubrimiento se procede a procesar todas las operaciones recibidas y que se encuentran almacenadas en las listas de operaciones pendientes.

- **close**: Elimina los *listeners* vinculados a la recepción de mensajes.

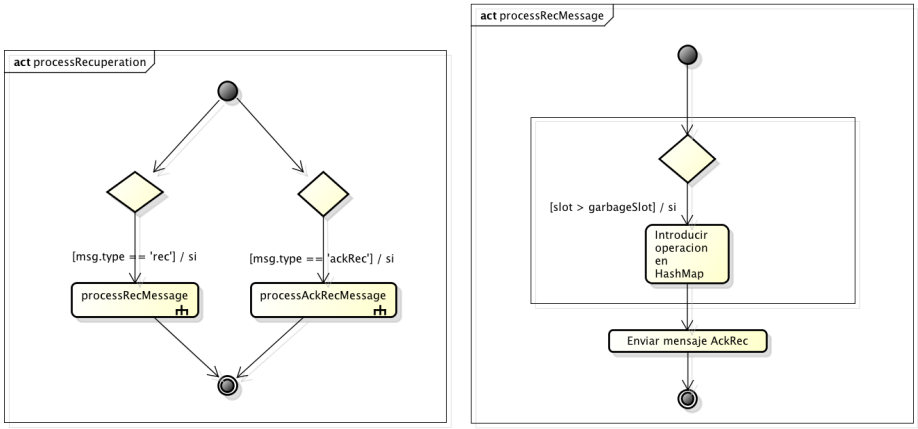
- **processMessage**: Recibe los mensajes enviados por el proceso **server**.

- **processRequest**: Procesa los mensajes *P1A* y *P2A*. En caso de que el aceptor se encuentre en estado activo se procesan los mensajes de manera normal, en caso contrario los mensajes son introducidos en una lista de mensajes pendientes, de forma que serán procesados en el momento que el **aceptor** cambie su estado a *activo*.

- **processRecuperation**: Procesa los mensajes pertenecientes al protocolo de comunicación. Envía los mensajes *REC* al método **processRecMessage** y los mensajes *AckRec* al método **processAckRecMessage**.

- **processRecMessage**: Procesa un mensaje *REC*, genera una tabla hash cuyo indice es el slot de cada operación y el valor es la operación perteneciente a dicho slot. En la tabla hash se introducen todas aquellas operaciones que se encuentran decididas para un slot superior al valor actual del recolector de basura.

  Finalmente se crea un mensaje *AckRec* que será enviado al **aceptor** que ha iniciado la recuperación. En dicho mensaje se envía la tabla hash con las operaciones, el valor del recolector de basura y el ballot actual que contiene el **aceptor**.



(a) (b)

Figura 5.6: Diagramas Actividad: (a)ProcessRecuperation. (b)ProcessRecMessage

- **processAckRecMessage**: Procesa un mensaje *AckRec*, dicho mensaje contiene todas las operaciones que han sido decididas por un **aceptor**, así como su valor de recolector de basura y el valor del ballot.

  Cuando recibe un mensaje, actualiza una lista cuyo contenido son las direcciones de todos los aceptores que han respondido al mensaje *REC*.

  Actualiza el valor del recolector de basura y el ballot, manteniendo el mayor valor entre el recibido y el que contiene el **aceptor**.

  Introduce cada una de las operaciones recibidas en la tabla hash de operaciones aceptadas.

Si ha recibido respuestas del mensaje *REC* de la mayoría de los **aceptores** del sistema, ejecuta el recolector de basura para eliminar aquellas operaciones que se encuentren con un valor de slot inferior al actual valor del recolector de basura, y se procesan aquellas peticiones que se han recibido mientras se ejecutaba el protocolo de recuperación.

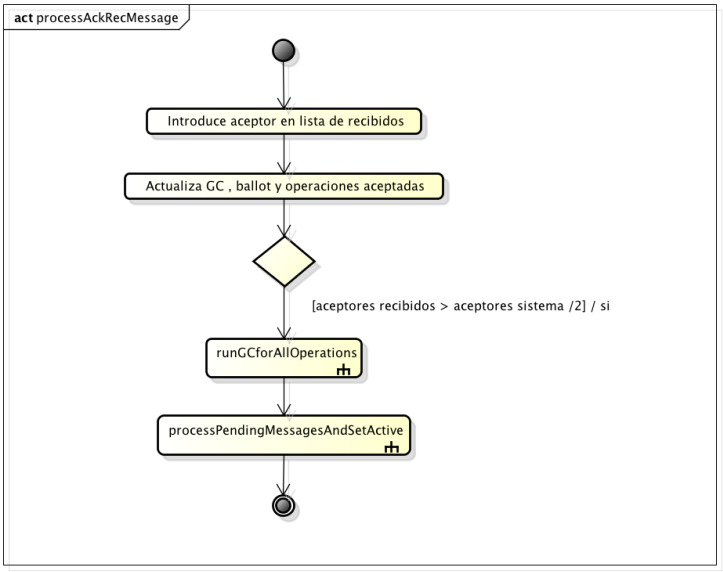


Figura 5.7: Diagrama Actividad: ProcessAckRecMessage

- **processPendingMessagesAndSetActive**: Recorre las listas de los mensajes pendientes *P1A* y *P2A* recibidos mientras se ejecutaba el algoritmo de recuperación, y llama a los métodos **processP1A** y **processP2A** para cada uno de ellos.

- **processP1A**: Procesa un mensaje *P1A* y responde al líder que ha enviado el mensaje con un mensaje *P1B*.

  Si el ballot que contiene el **aceptor** es inferior al ballot recibido en el mensaje *P1A*, el **aceptor** actualiza su ballot con el valor del recibido.

  Se crea un mensaje *P1B* conteniendo una lista de todas las operaciones aceptadas, el valor actual del ballot, así como el valor del recolector de basura y se envía al **líder**.

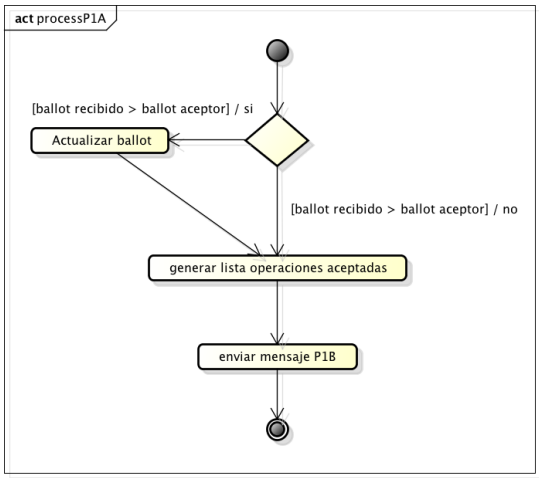


Figura 5.8: Diagrama Actividad: ProcessP1A

- **processP2A**: Procesa un mensaje *P2A* y responde al líder que ha enviado el mensaje con un mensaje *P2B*.

  Si el ballot enviado en el mensaje es igual o superior al ballot actual que contiene el **aceptor**, se actualiza el valor del ballot con el valor recibido. A continuación se añade la operación enviada en la lista de operaciones aceptadas, y se ejecuta el recolector de basura para marcar que todas las operaciones con un slot inferior al indicado en el mensaje, han sido procesadas por el **líder** que ha enviado dicho mensaje.

  Finalmente se envía un mensaje *P2B* al **líder** con el valor del ballot actual y el valor del recolector de basura.

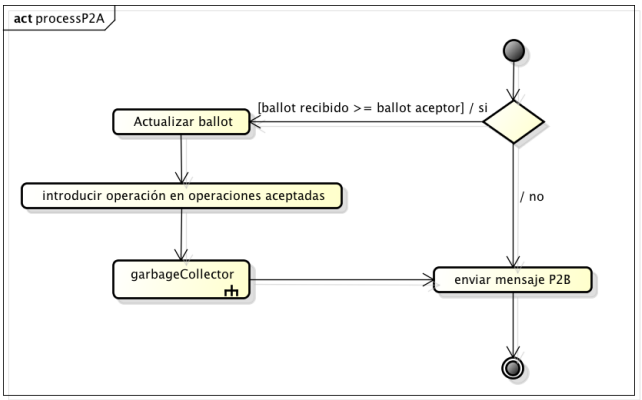


Figura 5.9: Diagrama Actividad: ProcessP2A

- **runGCForAllOperations**: Ejecuta el método **deleteAcceptedAndUpdateGarbage** para cada una de las operaciones aceptadas con un slot inferior al valor actual del recolector de basura.

- **garbageCollector**: Ejecuta el recolector de basura recibiendo como parámetros un slot y la dirección de un **líder**.

  Se añade la dirección del *lider* a una tabla Hash en la cual la clave es el valor del slot, y los valores de las claves una lista de todos los **líderes** que han aceptado operaciones hasta al menos dicho slot. Si el número de **líderes** que han aceptado operaciones hasta el slot indicado es igual o mayor que el número total de **líderes** en el sistema, se llama al método **deleteAcceptedAndUpdateGarbage** para que proceda a la eliminación de las operaciones.

- **deleteAcceptedAndUpdateGarbage**: Elimina las operaciones aceptadas para el slot indicado y actualiza el valor del recolector de basura con el valor del slot.

5.2.4. Leader

Clase encargada de inicializar el **Quorum** en el cual se elegirá una petición de entre todas las propuestas por las **réplicas**.

operaciones

- **constructor**: Inicializa la instancia del **líder** y crea una nueva instancia de *Scout* para adoptar un nuevo **ballot**. En el momento anterior de la inicialización de la instancia realiza una petición a **server**, para comprobar si existen **aceptores** en el sistema. Si este es el caso se procede a su inicialización, en caso contrario se mantiene a la espera hasta encontrar al menos un **aceptor**.

- **processMessage**: Procesa cada una de las peticiones enviadas por la clase *Server*. En especial procesa la petición *replicasMapAndAcceptorsMap* , la cual contiene los puertos y las direcciones de las **réplicas** y **aceptores** que existen en el sistema, y que han sido descubiertos antes de que el **líder** hubiera sido inicializado.

- **processRequest**: Procesa cada una de las peticiones enviadas a través de la red, y llama a la función específica para cada una de ellas. Acepta las peticiones **propose**, *P1B* , *P2B* y *GAP*.

- **processGAP**: Procesa un mensaje de tipo *GAP* enviado por las **réplicas**.

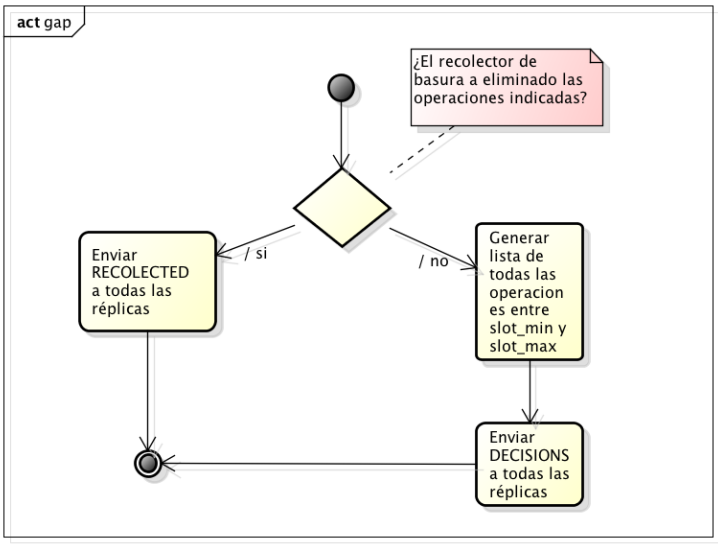


Figura 5.10: Diagrama Actividades: Procesar mensaje GAP

- **processPropose**: Procesa cada una de las peticiones de tipo *propose* . Si el **líder** se encuentra activo crea un nuevo *commander* con la operación propuesta.

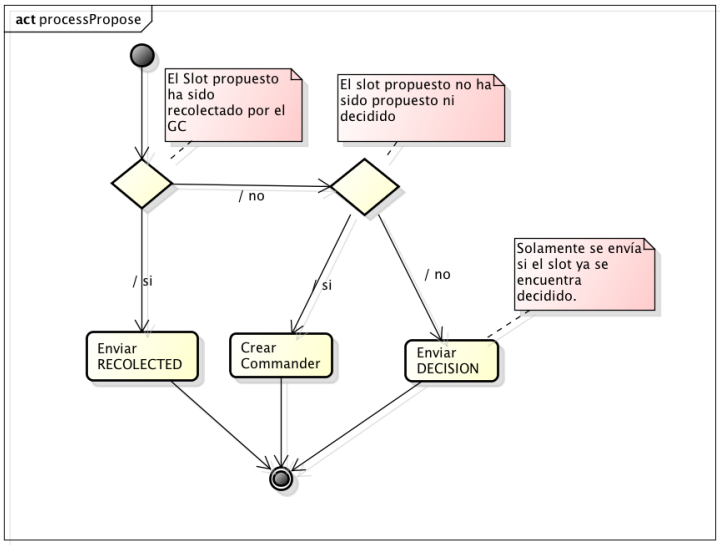


Figura 5.11: Diagrama Actividades: Procesar mensaje Propose

- **createScout**: crea un nuevo *Scout* con el **ballot** actual.
- **createCommander**: crea un nuevo *Commander*.
- **processAdopted**: Procesa cada una de las peticiones de tipo *adopted*. Por cada uno de las propuestas se crea un nuevo *commander*.

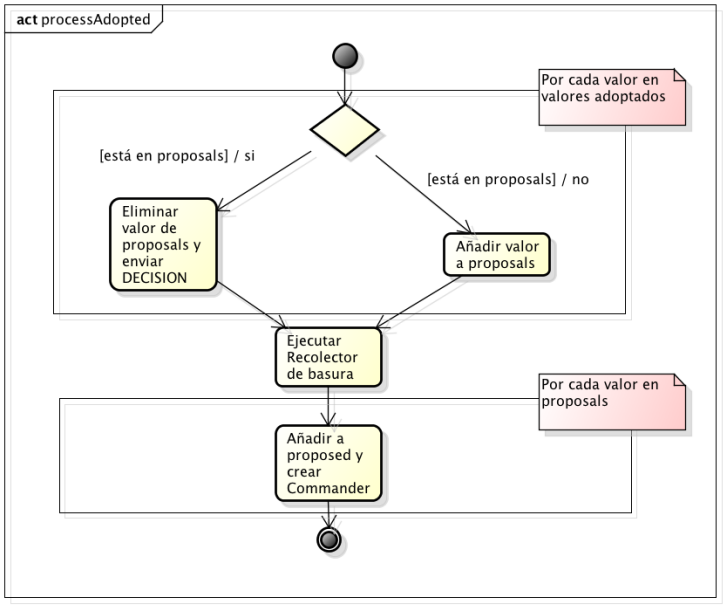


Figura 5.12: Diagrama Actividades: Procesar mensaje Adopted

- **processPreempted**: Procesa cada una de las peticiones de tipo *PREEMPTED*. Si el **ballot** recibido es mayor que el actual, se actualiza este y se crea un nuevo *Scout*.
- **processCommanderMessages**: Procesa los mensajes enviados por el *commander* al **líder**. Acepta los mensajes de tipo *DECISION* y *PREEMPTED*.
- **processDecisionMessageFromCommander**: Procesa los mensajes de tipo *DECISION* enviados por el *commander*. Actualiza las listas que mantienen los **slots** propuestos y decididos así como actualiza el valor del último **slot** decidido. A su vez ejecuta el recolector de basura y elimina el *commander* que ha enviado el mensaje de *decision*.
- **updateSlotsProposedDecidedAndLastSlotDecided**: Actualiza las listas que mantienen los **slots** propuestos y decididos, actualiza a su vez el valor del último **slot** decidido.

- **processPreemptedMessageFromCommander**: Procesa los mensajes de tipo *PREEMPTED* enviados por el *commander*. Cambia el estado del **líder** a inactivo, actualiza el **ballot** actual del sistema al recibido y elimina el *commander* que ha enviado el mensaje.

- **deleteCommander**: Elimina a un *commander* de la lista de *commanders* creados por el **líder**.

- **recolectGarbage**: Elimina todas aquellas operaciones que se encuentran en la lista de **proposals** y ya han sido decididas.

- **checkLeaderDown**: Comprueba si el líder que contiene adoptado el **ballot** del sistema se mantiene activo, si no es así llama al método **preempted** para inicializar la adopción de un nuevo **ballot**.

5.3. Scout y Commander

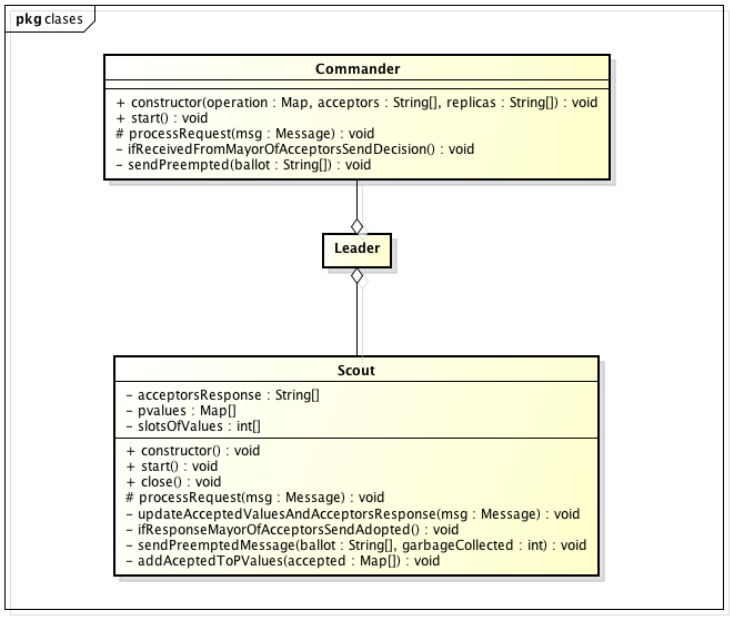


Figura 5.13: Diagrama Clases: Scout y Commander

5.3.1. Scout

operaciones

- **constructor**: Inicializa la instancia del **scout**.

- **start**: Comienza la ejecución del **scout**, enviando un mensaje de tipo *P1A* a todos los **aceptores** conocidos dentro del sistema.
- **close**: Finaliza el **scout**, elimina el listener que captura los mensajes destinados a la instancia.
- **processRequest**: Procesa los mensaje enviados al **scout**. En especial procesa los mensajes *P1B*. El procesamiento de dicho mensaje es como se describe a continuación:

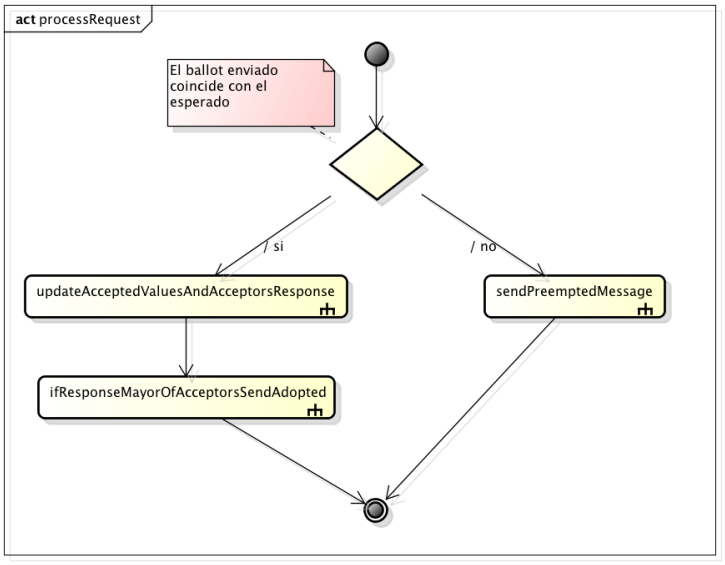


Figura 5.14: Diagrama Actividad: Procesamiento mensaje P1B

- **updateAcceptedValuesAndAcceptorsResponse**: Añade las operaciones enviadas en el mensaje de tipo *P1B* a la lista de operaciones aceptadas, llamando al método **addAcceptedToPValues**. A continuación actualiza la lista de los aceptores que han enviado el mensaje de tipo *P1B* y actualiza el valor del recolector de basura al mayor valor recibido.
- **ifResponseMayorOfAcceptorsSendAdopted**: Comprueba si el número de **aceptores** que han enviado los mensajes es superior a $(n/2) + 1$, donde n es el número de **aceptores** del sistema. De ser así envía un mensaje *ADOPTED* al **líder**.
- **sendPreemptedMessage**: Genera un mensaje *PREEMPTED* con el valor del **ballot** recibido y lo envía al **líder**.
- **addAcceptedToPValues**: Añade las operaciones recibidas en los mensajes *P1B* a la lista de operaciones aceptadas. Comprueba que las operaciones no se encuentran duplicadas dentro de la lista.

5.3.2. Commander

operaciones

- **constructor**: Inicializa la instancia del **commander**.
- **start**: Comienza la ejecución del **commander**, enviando un mensaje de tipo *P2A* a todos los **aceptores** conocidos dentro del sistema.
- **processRequest**: Procesa los mensajes enviados al **commander**. En especial procesa los mensajes *P2B*. El procesamiento de dicho mensaje es como se describe a continuación:

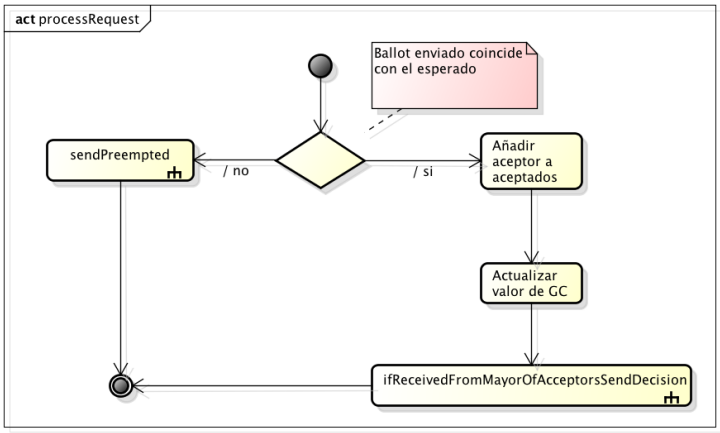


Figura 5.15: Diagrama Actividad: Procesamiento mensaje P2B

- **ifReceivedFromMayorOfAcceptorsSendDecision**: Comprueba si el número de **aceptores** que han enviado los mensajes es superior a $(n/2) + 1$, donde n es el número de **aceptores** del sistema. De ser así envía un mensaje *DECISION* a todas las **réplicas** conocidas del sistema y al **líder**.
- **sendPreempted**: Genera un mensaje *PREEMPTED* con el valor del **ballot** recibido y lo envía al **líder**.

5.4. Net, AcceptorNet, LeaderNet y réplicaNet

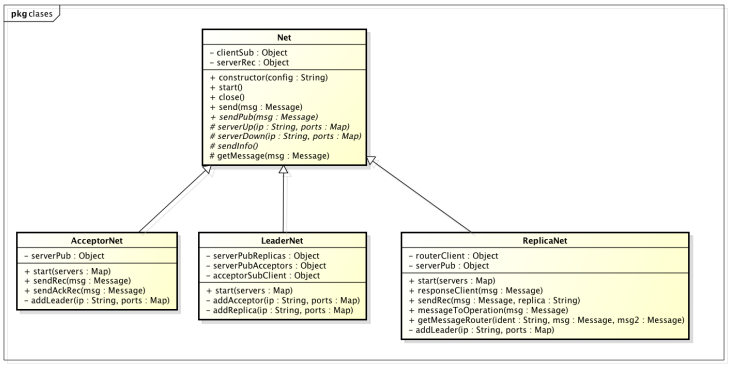


Figura 5.16: Diagrama Clases: Net, AcceptorNet, LeaderNet, réplicaNet

5.4.1. Net

La clase Net es la clase base encargada de realizar todas las comunicaciones de cada uno de los roles implementados dentro del sistema.

operaciones

- **constructor**: Inicializa la clase **Net**, creando el *socket* cliente para las subscripciones.
- **start**: Inicializa la escucha del cliente. Ante cada mensaje recibido llama al método **getMessage**. Inicializa el envío a intervalos regulares de los mensajes de información, utilizados por el sistema para conocer cual es el estado de cada uno de los procesos. Para enviar este mensaje de información se llama al método **sendInfo**.
- **close**: Finaliza la escucha del *socket* cliente y finaliza el envío de los mensajes de información.
- **serverUp**: Clase abstracta. Se ejecuta cuando se recibe un mensaje de tipo *UP*. Realiza acciones específicas según el tipo de implementación de **Net**
- **serverDown**: Clase abstracta. Se ejecuta cuando se recibe un mensaje de tipo *DOWN*. Realiza acciones específicas según el tipo de implementación de **Net**.
- **getMessage**: Des-encripta el mensaje recibido y envía un evento indicando que existe un nuevo mensaje.
- **send**: Envía un mensaje a la dirección indicada en el mismo.

- **sendPub**: Clase abstracta. Publica un mensaje a todos los subscriptores del sistema.
- **sendInfo**: Clase abstracta. Envía un mensaje de tipo información mediante *Broadcast*.

5.4.2. AcceptorNet

Clase encargada de realizar las comunicaciones referentes al rol **aceptor**.

operaciones

- **constructor**: Inicializa la clase **AcceptorNet**, creando el *socket* cliente subscrito a los mensajes *P1A* y *P2A*, un servidor de publicación y crea un servidor para la recuperación.
- **start**: Inicializa el servidor de publicación y el servidor de recuperación. Si existe algún servidor enviado como parámetro perteneciente a los **líderes**, se conecta a ellos para recibir los mensajes que publiquen. Si existe algún servidor enviado como parámetro perteneciente a **aceptores**, se introducen en una lista de aceptores conocidos para poder proceder posteriormente a la recuperación.
- **close**: Finaliza el servidor de publicación y recuperación.
- **sendRec**: Envía un mensaje *REC* a todos los **aceptores** conocidos dentro del sistema. Cuando se recibe una respuesta se envía un evento indicando que existe un nuevo mensaje.
- **sendAckRec**: Envía un mensaje *AckRec* al proceso que ha iniciado la petición de recuperación.
- **serverUp**: Si recibe un mensaje *UP* de un **líder**, se subscribe al mismo. Si recibe un mensaje *UP* de un **aceptor**, lo introduce en una lista de **aceptores** conocidos.
- **serverDown**: Si recibe un mensaje *DOWN* de un **líder**, se elimina la subscripción al mismo. Si recibe un mensaje *DOWN* de un **aceptor**, se elimina de la lista de **aceptores** conocidos.
- **addLeader**: Añade un **líder** a una lista de **líderes** conocidos y se subscribe al mismo para recibir sus publicaciones.
- **sendPub**: Publica mensajes *P1B* y *P2B* hacia todos los procesos subscritos.
- **sendInfo**: Envía información de la dirección IP del proceso, el tipo del mismo y de los puertos de sus servidores.

5.4.3. LeaderNet

Clase encargada de realizar las operaciones referentes al rol **líder**.

operaciones

- **constructor**: Inicializa la clase **LeaderNet**. Crea los servidores de publicación para **réplicas** y **aceptores**, así como los clientes para subscribirse a los mensajes *P1B* y *P2B* enviados por los **aceptores**, y a los mensajes *PROPOSE* y *GAP* enviados por las **réplicas**.
- **start**: Inicializa la escucha de los servidores de publicación para **réplicas** y **aceptores**. Si existe algún servidor enviado como parámetro perteneciente a **aceptores** o **réplicas** se conecta a ellos para recibir los mensajes que publiquen.
- **close**: Finaliza los servidores de publicación para **réplicas** y **aceptores**.
- **serverUp**: Si recibe un mensaje *UP* de un **aceptor** o una **réplica** se subscribe al mismo.
- **serverDown**: Si recibe un mensaje *DOWN* de un **aceptor** o una **réplica**, elimina la subscripción del mismo y elimina su dirección de la lista de procesos conocidos.
- **addAcceptor**: Añade un **aceptor** a una lista de **aceptores** conocidos y se subscribe al mismo para recibir sus publicaciones.
- **addréplica**: Añade una **réplica** a una lista de **réplicas** conocidas y se subscribe al mismo para recibir sus publicaciones.
- **sendPub**: Publica mensajes *P1A* y *P2A* a todos los **aceptores** subscriptos. Publica mensajes *DECISION*,*DECISIONS* y *RECOLECTED* a todas las **réplicas** subscriptas.
- **sendInfo**: Envía información de la dirección IP del proceso, el tipo del mismo y de los puertos de sus servidores.

5.4.4. réplicaNet

Clase encargada de realizar las operaciones referentes al rol **réplica**.

operaciones

- **constructor**: Inicializa la clase **ReplicaNet**. Crea el servidor de publicación encargado de publicar los mensajes destinados a los **líderes**. Se subscribe a los mensajes *DECISION* , *DECISIONS* y *RECOLECTED* enviados por los **líderes**.

- **start**: Inicializa el servidor de **router** encargado de recibir las peticiones de los clientes. Inicializa el servidor de recuperación. Si existe algún servidor enviado como parámetro perteneciente a **líderes** se conecta a ellos para recibir los mensajes que publiquen.
- **close**: Finaliza la escucha del *socket* cliente y finaliza el envío de los mensajes de información.
- **responseClient**: Envía un mensaje de respuesta al cliente una vez que su petición ha sido procesada.
- **sendRec**:Envía un mensaje *REC* a una réplica concreta del sistema.
- **serverUp**: Si recibe un mensaje *UP* de un **líder** se subscribe al mismo.
- **serverDown**: Si recibe un mensaje *DOWN* de un **líder**, elimina la subscripción del mismo y elimina su dirección de la lista de procesos conocidos
- **addLeader**: Añade un **líder** a una lista de **líderes** conocidos y se subscribe al mismo para recibir sus publicaciones.
- **getMessageRouter**: Método utilizado para procesar los mensajes recibidos a partir de un *socket* tipo *router*. Cuando recibe el mensaje envía un evento indicando que existe un nuevo mensaje.
- **messageToOperation**: Envía un mensaje al programa perteneciente a la réplica.
- **sendPub**: Publica mensajes *PROPOSE* y *GAP* a todos los **aceptores** subscriptos.
- **sendInfo**: Envía información de la dirección IP del proceso, el tipo del mismo y de los puertos de sus servidores.

Capítulo 6

Arquitectura del Sistema

Anteriormente se ha descrito tanto la estructura e implementación del sistema como los protocolos de comunicación utilizados en el mismo. A continuación se describe la arquitectura sobre la cual el sistema puede ser ejecutado y desplegado.

6.1. Arquitectura en Cluster

Todos los roles o procesos del sistema, **réplicas**, **aceptores** y **líderes** son ejecutados dentro de un clúster de ordenadores, o en un conjunto de ordenadores pertenecientes a una misma red. El cliente debe de conocer los puntos de entrada, en este caso las **réplicas**, de la aplicación para poder utilizar los servicios ofrecidos.

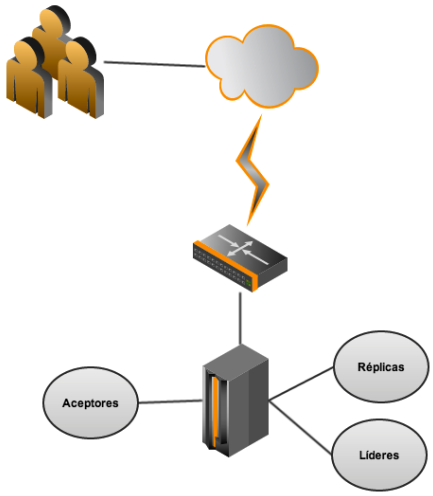


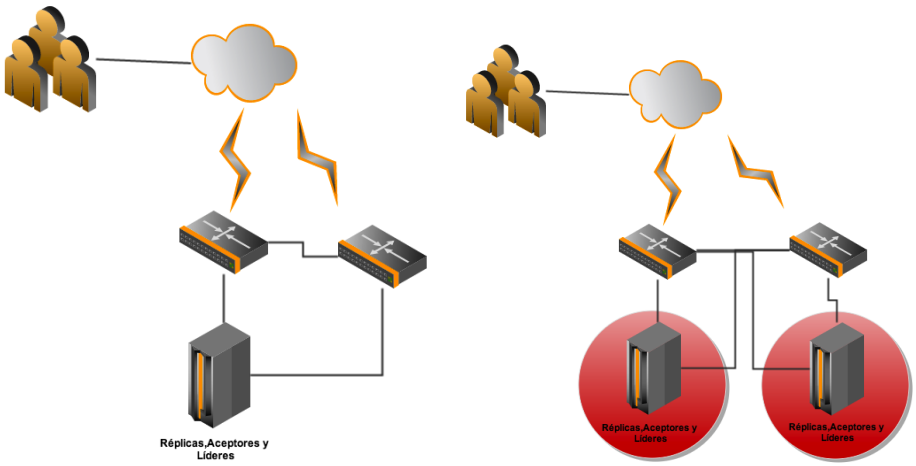
Figura 6.1: Arquitectura Clúster

Esta arquitectura conlleva un problema básico respecto a la disponibilidad del sistema, si la red de comunicación con el exterior falla el sistema se encontrará en un estado inaccesible, y por lo tanto no disponible. Para solucionar este problema se propone realizar una duplicidad de la red de comunicación con el exterior, de esta forma con $f + 1$ duplicidades de la red se podrían soportar f fallos de la misma.

Con dicha aproximación aún sigue existiendo un problema importante, si el clúster cae o falla, se perderán todas las operaciones almacenadas en los **aceptores** y el sistema se podría encontrar gravemente comprometido [1].

[1] Recordemos que los aceptores actuan como la memoria del sistema, gracias a ellos es posible conocer todas las operaciones realizadas y mediante estas, actualizar y recuperar el sistema

Una posible solución a dicho problema consiste en realizar una repartición de todos los procesos, y en especial de los **aceptores**, en diferentes dominios de fallos, asegurando en mayor o menor medida que el fallo de un dominio no imposibilite la continuidad del sistema.



(a) (b)

Figura 6.2: (a)Arquitectura Clúster con duplicidad de la red. (b)Arquitectura Clúster con dominios de fallos

Bajo esta configuración hay que tener en cuenta dos restricciones sin las cuales no se podría asegurar la disponibilidad:

- Los **aceptores** deben de encontrarse distribuidos de forma más o menos equitativa entre los dominios de fallos. De esta forma si un dominio de fallos no se encuentra accesible , el número de **aceptores** que se mantienen en el resto de dominios han de ser suficientes para permitir la continuación del sistema.

  A su vez, sería posible definir un número mínimo de **aceptores** dentro del sistema, de forma que cuando se detecte que el número de **aceptores** disponibles se encuentra cerca o por debajo de dicho umbral, se desplieguen nuevos **aceptores** dinámicamente.

- Al menos ha de existir un **líder** en cada uno de los dominios de fallos existentes, asegurando que aún con la caída de un dominio siga existiendo al menos un **líder** que pueda inicializar las votaciones.

  De la misma forma en cada dominio de fallos , aunque con un único **líder** se asegure la continuidad del sistema, sería necesario el mantener desplegados un número superior de los mismos, con el fin de soportar para dicho rol fallos de tipo **caida-parada**.

6.1.1. Descubrimiento de procesos

Con el fin de conseguir que el sistema se comporte de manera dinámica dentro del clúster, es necesario implementar un método por el cual los procesos puedan crearse y añadirse al sistema. Así como conseguir que los procesos que ya se encuentran en el sistema sean consciente de los procesos existentes, de aquellos que desean añadirse y de aquellos que abandonan el mismo.

Cuando un proceso descubre que un nuevo proceso se ha conectado, procede a realizar una conexión con él, añadiendo este a todas las comunicaciones futuras que realice. De la misma forma, cuando un proceso descubra que un proceso ha abandonado el sistema, procederá a eliminar todas las conexiones que se encontraban abiertas entre ellos.

La forma de implementar dicho descubrimiento es el siguiente:

- Cada uno de los procesos envía en un intervalo de tiempo t un mensaje a la dirección de **broadcast** de la red. En dicho mensaje se indica:
  - El proceso se encuentra vivo.
  - La dirección IP del proceso.
  - Los roles que implementa y por lo tanto que esta ejecutando.
  - Los puertos de entrada mediante los cuales el resto de los procesos se pueden comunicar con él. Los puertos de entrada pueden ser los siguientes:
    - **LTA**: Puerto utilizado para realizar la comunicación entre el **líder** y los **aceptores**.
    - **LTR**: Puerto utilizado para realizar la comunicación entre el **líder** y las **réplicas**.
    - **Replica**: Puerto utilizado para realizar la comunicación con la **réplica**.
    - **Aceptor**: Puerto utilizado para realizar la comunicación con el **aceptor**.
  - Un ejemplo del fichero JSON enviado con la información del nodo sería como sigue:

```
{
        "type":"UP",
        "body":{
                "ip":"x.x.x.x",
                "ports":{
                        "leader":{
                                "LTR":4444,
                                "LTA":3333,
                        },
                        "replica": 6666,
                        "acceptor":7777
                }
        }
}
```

- Cuando un proceso falla o se cae, envía un mensaje a la dirección de **broadcast** de la red. En dicho mensaje se indica:
  - El proceso se encuentra caído.
  - La dirección IP del proceso.
  - Los puertos de entrada que el proceso contenía.
  - Un ejemplo del fichero JSON enviado con la información del nodo sería como sigue:

```
{
        "type":"DOWN",
        "body":{
                "ip":"x.x.x.x",
                "ports":{
                        "leader":{
                                "LTR":4444,
                                "LTA":3333,
                        },
                        "replica": 6666,
                        "acceptor":7777
                }
        }
}
```

Este comportamiento se encuentra descrito en los diagramas de actividades que se muestran a continuación.

En este diagrama se muestra la actividad de envío de mensajes mediante **broadcast**. Cada t intervalos de tiempo se envía un mensaje indicando todos los datos necesarios para que los procesos puedan realizar una comunicación entre los mismos. Si el proceso falla se procede a enviar un mensaje mediante **broadcast** indicando que el proceso ha dejado de ser funcional.

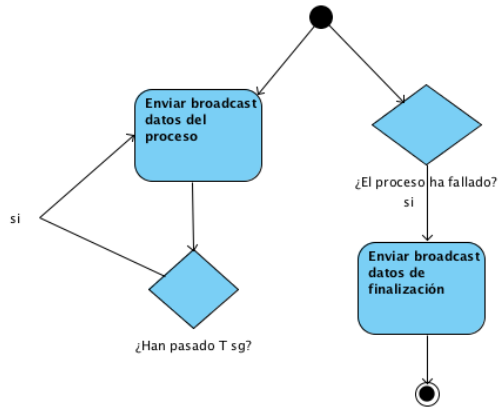


Figura 6.3: Envío de estado del proceso mediante broadcast

En el siguiente diagrama se muestra cual es el comportamiento de los procesos al recibir dichos mensajes.

Si el mensaje es de tipo *UP*, se comprueba si el proceso ya es conocido y pertenece a la misma red, si no es conocido y pertenece a la red se añade a la lista de procesos conocidos. Esta lista es usada posteriormente para conocer cual es el número de procesos de un determinado rol en ejecución dentro de una red, permitiendo desplegar nuevos procesos en caso de que el número en ejecución fuese insuficiente para proseguir con el correcto funcionamiento del sistema. A continuación el proceso comprueba si existe una conexión activa con dicho proceso, de no ser así procede a realizar la conexión con el fin de recibir los mensajes que envíe.

Si el mensaje es de tipo *DOWN* simplemente se realiza el proceso inverso al anterior. Se elimina el proceso de la lista de procesos conocidos y se procede a la desconexión de todas las conexiones realizadas a dicho proceso.

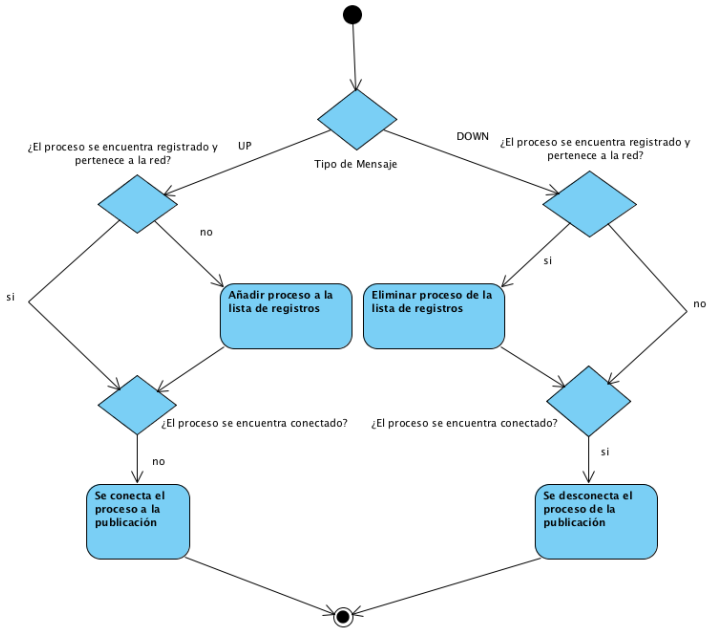


Figura 6.4: Registro de procesos

6.1.2. Procesos encubiertos

Por defecto, al ejecutar un proceso este se inicia automáticamente como un proceso activo, ejecutando los roles que han sido declarados para el mismo. Pero puede suceder que no se quiera que todos los procesos ejecuten los roles, sino que se mantengan a la espera de que ocurra algún evento que les indique

que deben ejecutar cierto rol. Para conseguir este comportamiento se utilizan procesos sigilosos o *Stealth*.

La diferencia entre un proceso normal y un proceso *Stealth* son los siguientes:

- Un proceso *Stealth* al inicializarse no ejecuta ningún rol.
- Un proceso *Stealth* realiza una monitorización del estado del sistema, comprobando cual es el número de instancias de cada uno de los roles dentro de la red.
- Si se cumple alguna condición declarada en las propiedades del sistema, por ejemplo «el número de aceptores dentro de la red es inferior a N», entonces el proceso *Stealth* ejecutará , en caso de no encontrarse ya en ejecución, este nuevo rol.

Los procesos *Stealth* permiten un mayor dinamismo del sistema dado que, ante la caída de algún nodo o proceso, un proceso *Stealth* puede inicializar automáticamente un rol, evitando de esta forma que el sistema no entre en un estado inconsistente.

También permiten una mayor elasticidad del sistema, al monitorizar los diferentes nodos, cada uno de los procesos pueden comprobar si es necesario la creación de un nuevo proceso con el fin de soportar la carga de trabajo, o de eliminar un proceso en caso de que la carga de trabajo haya disminuido.

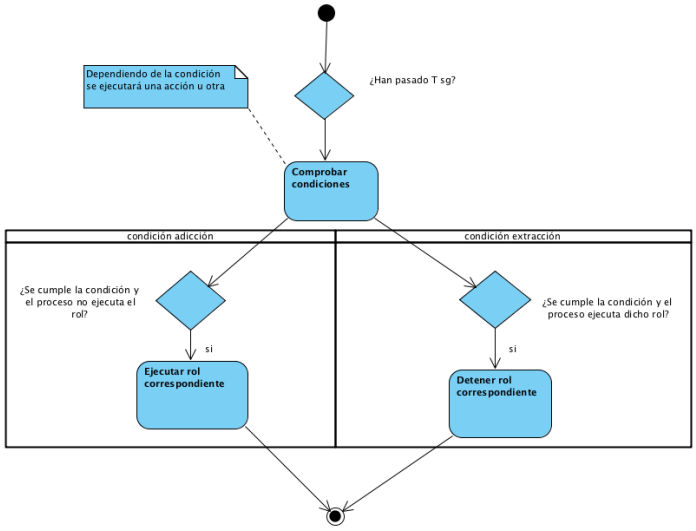


Figura 6.5: Proceso encubierto

6.1.3. Configuración

A continuación se muestra la estructura del fichero de configuración, y los campos que son necesarios completar para conseguir configurar de manera correcta el sistema sobre la arquitectura en clúster:

```
{
        "id":"",
        "roles":[],
        "stealth":false,
        "settings":{

        },
        "network":{

        }
}
```

- **id**: Identificador único del proceso. Debe de ser un identificador único dentro del sistema, por defecto se considera que el **id** es la IP del equipo donde se ejecuta el proceso.
- **roles**: Array que contiene los roles que podrán ser ejecutados por el proceso. Los valores posibles son: **réplica**,**líder**,**aceptor**.
- **stealth**: Valor que indica si el proceso se ejecutará de forma normal o en forma de proceso encubierto. Los valores posibles son *true* y *false*. Si se indica *true* el proceso se ejecutará como un proceso encubierto, ejecutandose de manera normal en caso contrario.
- **settings**: Contiene la configuración de cada uno de los roles que pueden ser ejecutados por el sistema, indicando cada uno de los puntos de entrada de los mismos.
- **network**: Contiene la configuración de la red en la cual se encuentra ejecutando el proceso.

Settings

El apartado settings contiene la configuración de cada uno de los puntos de entrada de los roles que se van a desplegar dentro del proceso. En la aplicación final no es necesario definir cada uno de los roles, sino simplemente aquellos que se van a desplegar. Por ejemplo, si el proceso únicamente va a actuar como un **aceptor** entonces únicamente se rellenarán los datos vinculantes al **aceptor**.

```
{
        "acceptor":{
                "publisher":{
                        "protocol":"",
                        "address":"",
                        "port":""}
        },
        "replica":{
                "router":{
                        "protocol":"",
                        "address":"",
                        "port":""},
                "publisher":{
                        "protocol":"",
                        "address":"",
                        "port":""}
        },
        "leader":{
                "publisherToReplicas":{
                        "protocol":"",
                        "address":"",
                        "port":""},
                "publisherToAcceptors":{
                        "protocol":"",
                        "address":"",
                        "port":""}
        }
}
```

- **acceptor**: Contiene la configuración de los puntos de entrada para el rol de **aceptor**. El **aceptor** contiene un único punto de entrada, el cual es el utilizado para comunicarse con los líderes del sistema. Para dicho punto de entrada es necesario definir las siguientes propiedades:
  - **protocol**: Protocolo utilizado para la comunicación, por defecto **TCP**.
  - **address**: Direcciones aceptadas para realizar la escucha, por defecto *****.
  - **port**: Puerto de escucha.
- **replica**: Configuración de los puntos de entrada para el rol de **réplica**. Contiene dos puntos de entrada que son:
  - **router**: Punto de entrada utilizado por los clientes para comunicarse con la **réplica**.
  - **publisher**: Punto de entrada utilizado para publicar mensajes a los **líderes**.
- **leader**: Configuración de los puntos de entrada para el rol de **líder**. Contiene dos puntos de entrada que son:
  - **publisherToReplicas**: Punto de entrada utilizado para publicar mensajes a las **réplicas**.
  - **publisherToAcceptors**: Punto de entrada utilizado para publicar mensajes a los **aceptores**.

Network

```
{
        "broadcast":"x.x.x.x",
        "domains":[["x.x.x.x","x.x.x.x"]],
        "heartbeat":10000,
        "stealth_time":15000,
        "min_acceptors":1,
        "min_leaders":1,
}
```

- **broadcast**: Indica la dirección de *broadcast* de la red a la cual se enviarán los mensajes de *heartbeat*.
- **domains**: Array con cada uno de los subdominios a los cuales pertenece el proceso. El subdominio se encuentra compuesto por un array que contiene la dirección de comienzo de la red y la dirección de finalización de la misma.
- **heartbeat**: Intervalo de tiempo en el cual se envía un mensaje a la dirección de broadcast con los datos de configuración del proceso. El intervalo se encuentra definido en milisegundos.
- **stealth_time**: Intervalo de tiempo en el cual un proceso encubierto comprueba el sistema. El intervalo se encuentra definido en milisegundos.
- **min_acceptors**: Número mínimo de aceptores que pueden existir en el dominio. Únicamente es necesario si el proceso es **stealth**.
- **min_leaders**: Número mínimo de líderes que pueden existir en el dominio. Únicamente es necesario si el proceso es **stealth**.

6.2. Réplicas Distribuidas

Con la arquitectura descrita anteriormente se consigue ejecutar todos los procesos en un mismo clúster, lo que permite una comunicación más fiable y unos tiempos de comunicación inferiores a los que se podrían conseguir, por ejemplo, en una red *WAN*.

Aún así, podría suceder que se desee mantener una distribución de las **réplicas** en diferentes redes, con el fin de asegurar una mayor disponibilidad del sistema y poder mantener las diferentes **réplicas** cercanas a los usuarios que van a hacer uso de las mismas.

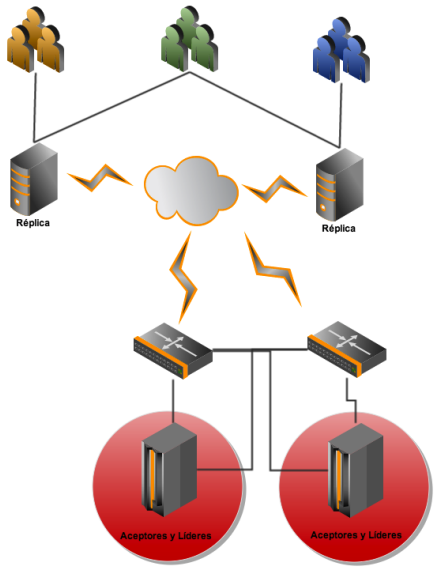


Figura 6.6: Arquitectura Réplicas distribuidas

En este caso el descubrimiento de los procesos no se puede realizar mediante *broadcast*, pues los procesos se encuentran en redes diferentes. Por ello, las **réplicas** deben conocer el punto de acceso de al menos uno de lo líderes del sistema, y realizar al mismo una petición de adhesión al sistema. El algoritmo de adhesión sería como sigue:

- Una nueva **réplica** se inicializa en el sistema, envía un mensaje a los líderes que conoce indicando sus datos de conexión.
- El **líder** recibe la petición de la **réplica** y decide si la acepta o no dentro del sistema. Si la acepta responde a la **réplica** con sus datos de conexión y se subscribe a las publicaciones de la misma.
- La **réplica** recibe la respuesta del **líder** y se subscribe a las publicaciones de la misma.

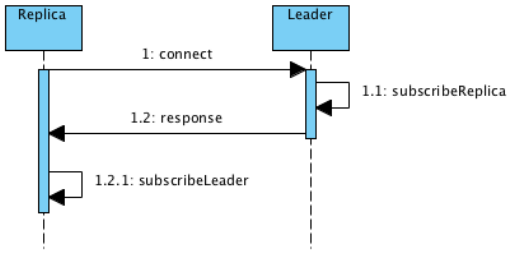


Figura 6.7: Subscripción Réplica-Lider

6.2.1. Configuración

En caso de utilizar la arquitectura de réplicas distribuidas, es necesario añadir un nuevo punto de entrada a los líderes, de forma que esta sea utilizada para procesas las peticiones de adhesión al sistema por parte de las réplicas.

```
{
        "leader":{
                "publisherToReplicas":{
                },
                "publisherToAcceptors":{
                },
                "joinReplica":{
                        "protocol":"tcp",
                        "address":"*",
                        "port":"xxxx",
                }
        }
}
```

6.3. Procesos Distribuidos

Aunque actualmente no se encuentre implementado, mediante unas ligeras modificaciones el sistema podría permitir el despliegue de cada uno de los roles en diferentes redes, de forma que tanto **réplicas** , **líderes** y **aceptores** puedan desplegarse cada uno de ellos en redes diferentes. La comunicación y el descubrimiento de cada uno de los procesos se llevaría a cabo de la misma forma que la descrita en el apartado de «Réplicas Distribuidas».

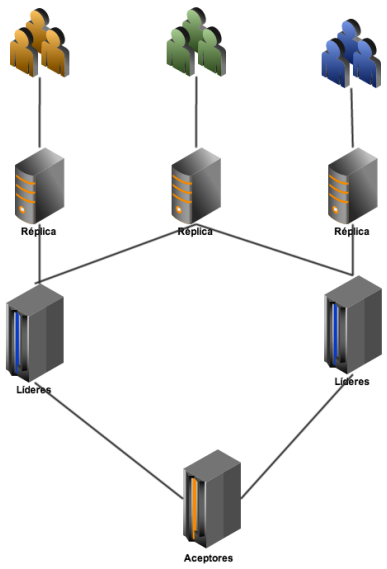


Figura 6.8: Arquitectura procesos distribuidos

No todas las réplicas tienen que conocer a todos los líderes, ni todos los líderes a todas las réplicas, pero si es necesario que todos los líderes conozcan al mismo conjunto de aceptores, o al menos a la mayoría de los mismos. De no ser así no se podría asegurar la consistencia del sistema.

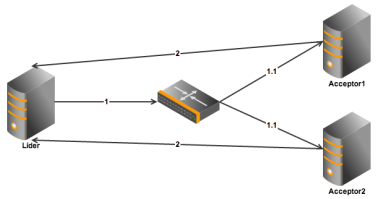


Figura 6.9: Adhesión líder con aceptores

Debido a que los **aceptores** pueden migrar y modificar sus direcciones , un **líder** puede no conocer en todo momento cual es la dirección de todos y cada uno de los **aceptores** que forman parte del sistema. Una posible solución para resolver este problema consiste en que el **líder**, en vez de enviar una petición de adhesión a cada uno de los **aceptores**, envíe una petición a la dirección de **broadcast** de la red donde se encuentran los mismos, de esta forma cada uno de los aceptores recibirá el mensaje y podría responder individualmente.

6.3.1. Configuración

Al igual que en la configuración de «Réplicas distribuidas», es necesario añadir un nuevo punto de entrada tanto en los **líderes** , como en los **aceptores**, el cual servirá para procesar las peticiones de adhesión al sistema.

```
{
        "leader":{
                "publisherToReplicas":{
                },
                "publisherToAcceptors":{
                },
                "joinReplica":{
                        "protocol":"tcp",
                        "address":"*",
                        "port":"xxxx",
                }
        },
        "acceptor":{
                "publisher":{
                },
                "joinLeader":{
                        "protocol":"tcp",
                        "address":"*",
                        "port":"xxxx",
                }
        }
}
```

Capítulo 7

Recolección de basura

Tal y como se ha visto anteriormente, los **aceptores** actúan como la memoria principal del sistema, manteniendo todas las operaciones aceptadas en el sistema. De esta forma si algún nodo pierde su estado y necesita realizar una recuperación, mediante los **aceptores** puede recuperar el estado perdido.

Desde el punto de la recuperación el mantenimiento de todas las operaciones realizadas es toda una ventaja, pero conlleva un grave problema: el tamaño de la lista de operaciones puede crecer desproporcionadamente, provocando un consumo excesivo de memoria.

Por lo tanto es necesario encontrar un método mediante el cual, sea posible eliminar todas aquellas operaciones que con mayor o menor seguridad se conozca que no volverán a ser utilizadas. De esta forma no se vería mermada la capacidad de recuperación del sistema y se disminuiría el consumo de memoria.

A lo largo del capítulo se describirán varias aproximaciones orientadas a la implementación final del recolector de basura, las aproximaciones se realizan de la forma más genérica posible, por lo que no se tienen en cuenta las características de la comunicación implementadas en la aplicación.

Al final del capítulo se realiza la descripción del recolector de basura implementado y los rendimientos obtenidos con el mismo.

7.1. Primera aproximación

Se supone en un principio que el sistema es estático, es decir, en todo momento existe un número fijo de instancias de cada uno de los roles. A su vez dicho número es conocido por todas y cada una de las instancias.

En este caso, si se sabe que todas las réplicas ha procesado hasta un slot «*S*», se podrá asegurar que todas las operaciones de los slots anteriores ya no serán utilizadas, y por lo tanto podrían ser eliminadas del sistema. La solución a la recolección de basura en este caso sería la siguiente:

- Cuando una **réplica** ha procesado una operación adoptada, envía un mensaje a todos los **líderes** del sistema indicando que ya es conocedor de dicha operación.

- Cuando un **líder** recibe la confirmación de la operación de todas las **réplicas**, actualiza su estado indicando que las **réplicas** han procesado como mínimo hasta la operación del slot «*S*», y eliminando todas las proposiciones inferiores a dicho slot.

- En cada envío de mensajes *P1A* y *P2A* el **líder** envía el valor del slot en el cual se encuentra la operación.

- El **aceptor** al recibir el mensaje actualiza su estado, actualizando el valor del slot actual de recolección de basura y eliminando todas aquellas operaciones con un valor inferior a este.

- El **aceptor** en cada envío de mensajes *P1B* envía el valor del slot actual de recolección de basura. De esta forma el **lider** es conocedor de hasta que slot se ha ejecutado el recolector de basura.

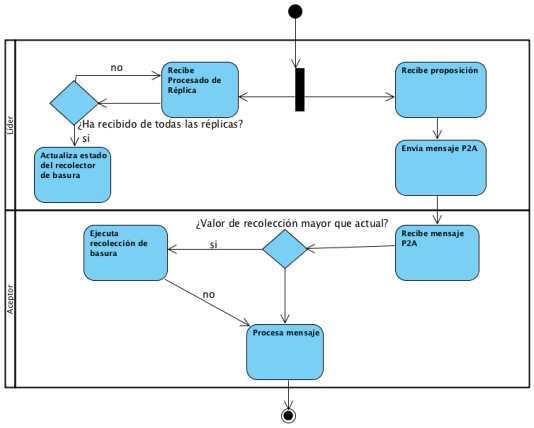


Figura 7.1: Diagrama Actividad: Recolección de basura

En la siguiente figura se muestra en que estado se encontraría el sistema después de ejecutar el recolector de basura. De las dos **réplicas** existentes, una de ellas ha procesado hasta la operación del slot 3, mientras que ambas han procesado todas las operaciones hasta el slot 2. El **líder**, al recibir la confirmación de procesamiento del slot 2 de todas las **réplicas** actualiza su estado, y en la próxima petición envía el valor del slot a todos los **aceptores**. Los **aceptores** al recibir el valor del slot 2 eliminan todos los valores anteriores, dejando únicamente en memoria los valores superiores a dicho slot.

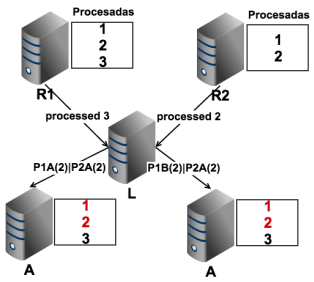


Figura 7.2: Ejemplo Recolección de basura

7.2. Segunda aproximación

La primera aproximación posee un notable problema, si al menos una de las **réplicas** se cae o se ejecuta más lentamente que el resto de las **réplicas**, el recolector de basura no funcionará correctamente.

Si vemos el ejemplo anterior, se puede observar como si cae R2, ninguna de las operaciones de los aceptores sería eliminada.

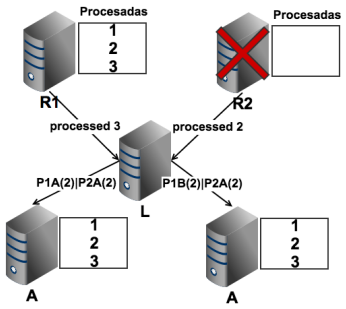


Figura 7.3: Recolección de basura caida-parada

Para solventar este problema existen dos tipos de soluciones:

- Las **réplicas** son recuperables. Una vez que una **réplica** cae, esta puede volver a levantar con su mismo identificador y manteniendo su estado. De esta forma la **réplica** simplemente debería pedir al **líder** las operaciones pendientes y se podrá proseguir el algoritmo de forma normal. Sería el equivalente a tener una **réplica** lenta.

  Aún así, si esta **réplica** siguiese cayendo a intervalos regulares, en los cuales no fuera posible conseguir avanzar su estado, el recolector de basura seguiría sin poder funcionar correctamente.

- Modificar el soporte de fallos, de modo que en vez de soportar fallos de caída-parada, se soporten fallos bizantinos. De esta forma en el sistema se ejecutarán $2f + 1$ réplicas, de modo que si el líder recibe confirmación de $(n/2) + 1$ **réplicas** continuará con el algoritmo de recolección de basura.

  En este punto, si una **réplica** no ha podido actualizar su estado, necesitará realizar una comunicación con el resto de las **réplicas** para recibir aquellos slots que no haya podido obtener.

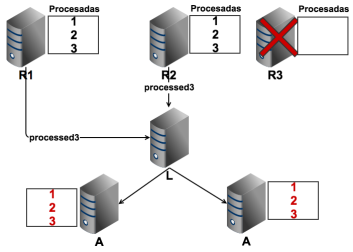


Figura 7.4: Recolección de basura fallo bizantino

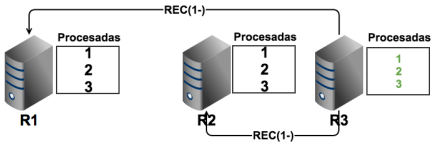


Figura 7.5: Recuperación replica

7.3. Tercera aproximación

La segunda aproximación es válida si nos encontramos ante un sistema estático en el que es conocido el número de procesos que se encuentran dentro del sistema. Si nos encontramos en un sistema en el cual no todos los líderes conocen a todas las réplicas, este algoritmo de recolección de basura tampoco sería usable. Para demostrar que este recolector de basuras no funciona en dicho entorno se muestra un ejemplo.

Supóngase que existe un sistema con 2 **líderes** y a cada uno de ellos se conectan 2 **réplicas**, el conjunto de las **réplicas** es disjunto entre sí, y cada uno de los líderes desconoce a las **réplicas** que se conectan al otro líder.

En un momento dado, un **líder** descubre que todas las **réplicas** han procesado hasta el slot «N» y envía un mensaje al conjunto de **aceptores** para que ejecuten el recolector de basura.

Un instante después el otro **líder** descubre que todas las **réplicas** han procesado hasta el slot «N-1», y envía un mensaje al conjunto de **aceptores**. En este momento, el **líder** descubre que ya ha pasado el recolector de basura, y dado que desconoce al resto de **líderes** y **réplicas** se encuentra imposibilitado para continuar.

Con el fin de encontrar solución a este problema se proponen las siguientes soluciones:

- Los **aceptores** mantienen una lista de todos los **líderes** que se encuentran conectados al mismo, así como el slot hasta el cual han realizado el procesamiento. Cuando un **líder** intenta adoptar un valor con un slot ya

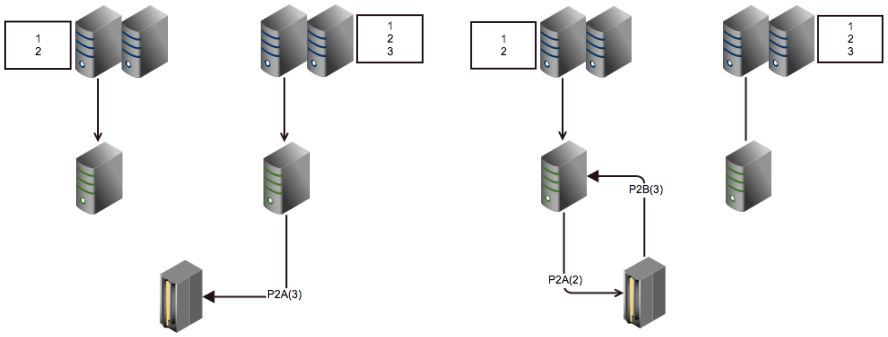


Figura 7.6: Recolección de basura incorrecto

eliminado, el **aceptor** le enviará un mensaje con los **líderes** que ya han pasado por dicho slot. El **líder** se conectará a alguno de estos **líderes** para intentar recuperar la operación eliminada.

- Se aumenta el soporte de fallos de **fallos caída-parada** a **fallos bizantinos**. Los **aceptores** no realizarán la recolección de basura hasta que hayan recibido la confirmación de más de f líderes. El resto de los **líderes** que no hayan podido actualizar su valor mediante los **aceptores**, recuperaran su valor de la misma forma que la indicada en el punto anterior.

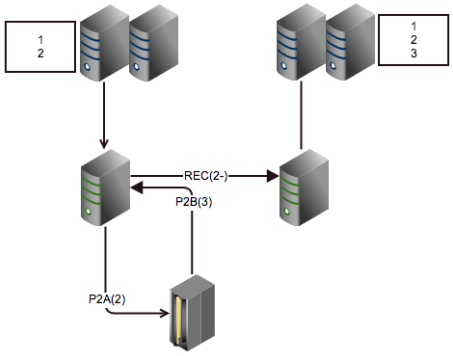


Figura 7.7: Recuperación del líder de operaciones eliminadas

En este punto también podría suceder el caso de que todos los **líderes** hayan ejecutado su propio recolector de basura, por lo tanto el **líder** no tendría opción de recuperar o actualizar su estado. La solución propuesta ante este problema es la siguiente [1].

[1] el porque de esta solución se explica con más detenimiento en el capítulo de **recuperación del sistema**

- Si el **líder** que es preguntado para retornar las operaciones ya ha ejecutado su recolector de basura, envía al **líder** una lista con las réplicas que se encuentran conectadas al mismo.
- El **líder** al recibir el mensaje, realiza una petición de conexión a dichas **réplicas** y a continuación les solicita las operaciones pendientes. El **líder** una vez ha recibido las operaciones no cierra la conexión, sino que la mantiene abierta con el fin de mantener el sistema más acoplado y evitar en mayor o menos medida futuras perdidas de información.
- El **líder** envía dichas operaciones al resto de las **réplicas** para que estas puedan proseguir la actualización.

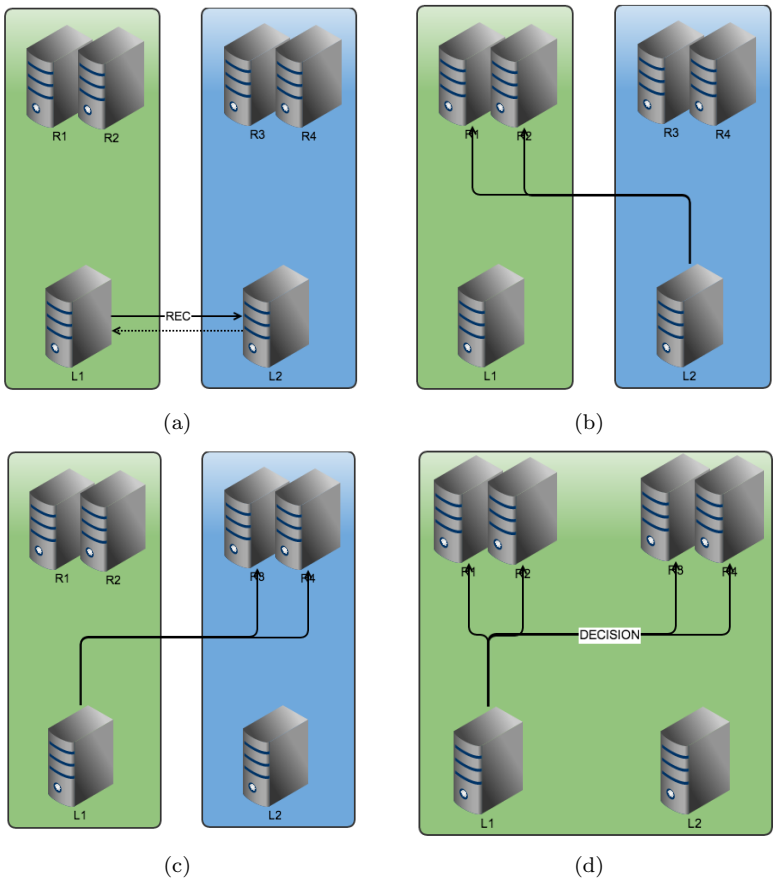


Figura 7.8: (a)Petición de un lider para recuperar operaciones, envía las réplicas conocidas. El lider receptor envía la última operación eliminada y su lista de réplicas.(b)(c) Los líderes se subscriben a las réplicas que no conocen.(c) Se actualiza el estado de operaciones del lider.(d) Se envían las nuevas operaciones a las réplicas.

7.4. Recolector Implementado

El recolector de basura realizado en la implementación actual del sistema se basa en las siguientes premisas:

- Si una operación aceptada ha sido leída por todos los **líderes** del sistema, estos no necesitarán acceder posteriormente a dicha información y por lo tanto puede ser eliminada.

- Si una operación es enviada a todos las **réplicas** del sistema, no será necesario que los **líderes** sigan manteniendo dicha información y por lo tanto dicha información puede ser eliminada.

A continuación se muestra en funcionamiento del recolector de basura tanto en los **aceptores** como en los **líderes** que forman parte del sistema. Dado que las **réplicas** han de mantener en todo momento su estado, no implementan un recolector de basura.

7.4.1. Recolector de basura en los aceptores

Como ya se ha comentado anteriormente, los **aceptores** actúan como la memoria principal del sistema, manteniendo todas las operaciones que han sido aceptadas, así como la posición en la cual se ha producido dicha aceptación.

A pesar de la importancia de mantener todas las operaciones, esto resulta inviable en términos de almacenamiento. Almacenar todas las operaciones aceptadas ocuparía gran cantidad de memoria y posiblemente la mayoría de ellas no volverían a ser utilizadas, por lo que su almacenamiento carecería de sentido.

Para paliar este problema se ha implementado un recolector de basura, el cual elimina aquellos elementos que ya han sido leídos por todos los **líderes** que en un momento dado forman parte del sistema, y a su vez dicho número es superior al número de líderes necesarios para que el sistema pueda seguir realizando sus funciones correctamente.

Algoritmo recolector de basura

La condición para ejecutar el recolector de basura es la siguiente: Si $f + 1$ **líderes** (donde f es el número de líderes que pueden fallar) han recibido una operación aceptada por los **aceptores**, entonces estos **líderes** ya conocen dicha operación y no van a necesitar aprenderla en pasos posteriores, por lo tanto dicha operación puede ser eliminada.

De esta forma, el recolector de basura se ejecuta en los **aceptores** una vez que ha recibido un mensaje *P2B*, en la cual se confirma que un **líder** ha recibido una operación y pide la aprobación para decidir la misma.

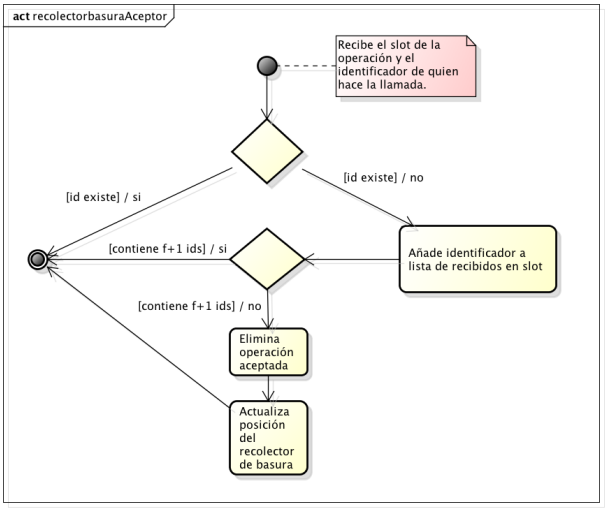


Figura 7.9: Ejecución del recolector de basura

Cuando se ejecuta el recolector de basura, en primer lugar se comprueba si la operación ya ha sido leída por el **líder** que realiza la petición. Si la petición ya ha sido leída por dicho **líder** el recolector finaliza; si no es así añade al **líder** a una lista de *líderes que han leido el slot x*. A continuación se comprueba si dicho slot ha sido leído por $f + 1$ **líderes** del sistema, si ha sido así, se procede a eliminar la operación perteneciente a dicho slot y a actualizar el valor del recolector de basura, el cual apuntará a la última operación recolectada.

Rendimiento recolector de basura

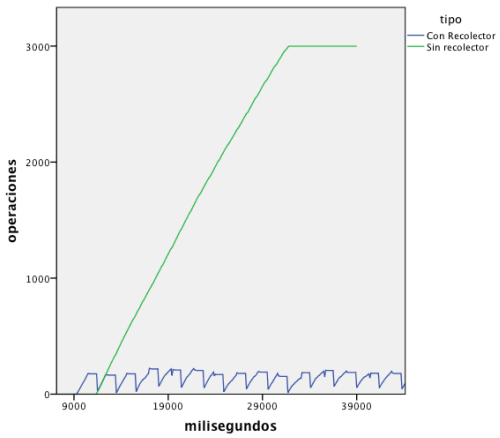


Figura 7.10: Comparación de elementos entre ejecutar y no ejecutar el GC

En la figura anterior se muestra una comparación del número de elementos que se almacenan en los **aceptores** con el recolector de basura habilitado e inhabilitado. Para realizar dicha comparación se han enviado 3000 operaciones secuenciales a dos **líderes**, los cuales van turnando su ejecución en intervalos comprendidos entre 300 y 3000 mili-segundos, asegurando que se ejecute en todo momento la operación encargada de lanzar el recolector de basura.

		tipo	
		Con Recolector	Sin recolector
operaciones	Mínimo	0	0
	Máximo	223	2999
	Media	117	1378
	Mediana	131	1280
	Rango	223	2999

Figura 7.11: Comparación de elementos entre ejecutar y no ejecutar el GC

Cuando no se ejecuta el recolector de basura, el número de operaciones almacenadas dentro del **aceptor** crece en igual proporción al número de operaciones que procesa el sistema.

Cuando se ejecuta el recolector de basura si ambos **líderes** han leído las operaciones estas son eliminadas, de esta forma se limita el número de operaciones almacenadas en los **aceptores** al número de operaciones que se almacenan entre las lecturas de todos los **líderes** , disminuyendo sustancialmente el número de operaciones almacenadas en memoria.

En las siguientes gráficas se muestra la evolución del recolector de basura, comparando su ejecución utilizando 2 o 3 líderes, como en el caso anterior, se realizan 3000 peticiones de forma secuencial, y cada uno de los líderes procede ha realizar su ejecución en intervalos comprendidos de 300 y 3000 mili-segundos.

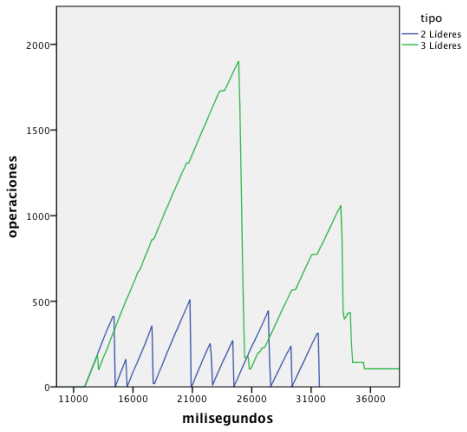


Figura 7.12: Número de operaciones en ejecución con 2 y 3 líderes

		tipo	
		2 Líderes	3 Líderes
operaciones	Mínimo	0	0
	Máximo	510	1901
	Media	115	455
	Mediana	64	168

Figura 7.13: Número de operaciones en ejecución con 2 y 3 líderes

Tal como se puede observar el número de operaciones almacenadas en el **aceptor** aumenta según aumenta el número de **líderes** que se ejecutan, pues hasta que todos ellos no han leído los valores estos no se pueden eliminar. A su vez se puede observar como el número de veces que se ejecuta el recolector de basura es inversamente proporcional al número de **líderes**, a más **líderes** menos veces se llamará al recolector.

7.4.2. Recolector de basura en los líderes

Los **líderes** únicamente necesitan almacenar las operaciones que han sido decididas hasta el momento en el cual son enviadas a las **réplicas**, una vez que han sido enviadas a las **réplicas** dichas operaciones pueden ser eliminadas por el recolector de basura. Dado que el envío del mensaje a las **réplicas** se realiza de forma atómica a todas ellas, una vez enviado dicho mensaje se puede eliminar la operación.

Algoritmo recolector de basura

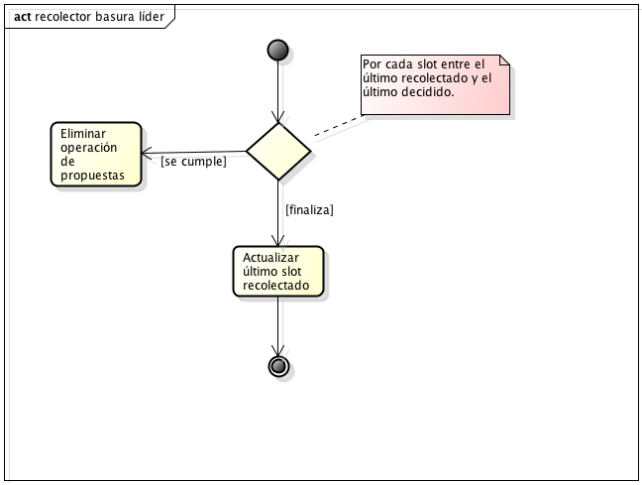


Figura 7.14: Ejecución del recolector de basura

Una vez que el mensaje de *operación decidida* ha sido enviada a todas las **réplicas** que forman parte del sistema, el recolector de basura recorre todas las posibles posiciones existentes entre la última posición inspeccionada por el recolector de basura y la última posición donde ha sido decidida una operación. Si en dicha posición existe alguna operación propuesta esta se elimina, pues ya ha sido decidida y notificada a todas las **réplicas**. Una vez que ha recorrido todas las posiciones, actualiza la posición del recolector de basura a la última posición donde se ha decidido una operación.

Rendimiento recolector de basura

A continuación se muestra una comparación del número de elementos que se almacenan en los **líderes** utilizando y no utilizando el recolector de basura. Para dicha comparación se han enviado 3000 operaciones secuenciales a dos **líderes**, los cuales van turnando su ejecución en intervalos comprendidos entre 300 y 3000 mili-segundos.

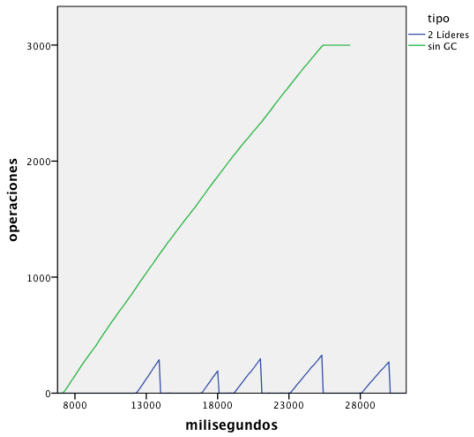


Figura 7.15: Comparación de líder con y sin recolector de basura

Cuando no se ejecuta el recolector de basura las operaciones propuestas no se eliminan del sistema, las peticiones almacenadas van creciendo al mismo ritmo de la llegada de las peticiones enviadas por las *réplicas*.

Al ejecutar el recolector de basura, mientras el **líder** se mantiene en ejecución el número de operaciones almacenadas tiende a cero , esto es debido a que decide todas las operaciones propuestas por las **réplicas** y las elimina inmediatamente.

Los picos en el número de operaciones es debida a dos motivos:

- El **líder** se encuentra dormido esperando adoptar un nuevo **ballot**. En este estado el **líder** tiene que almacenar las operaciones que reciba y no

puede proponer las mismas a los **aceptores**.

- Cuando recibe un mensaje de **adopción** por parte de los **aceptores** también recibe aquellas peticiones que han sido aprendidas por las mismas, lo que hace que el número de operaciones también se vea incrementado.

En la gráfica de a continuación se muestra la diferencia entre la ejecución del recolector de basura existiendo en el sistema 2 y 3 **líderes**. Como se puede observar el comportamiento es básicamente parecido, con la excepción de mayores picos en algunos puntos de la ejecución referente a los 3 **líderes**. Estos picos son debidos a las operaciones enviadas por los **aceptores** y que aún no han sido eliminadas por el recolector de basura de los mismos.

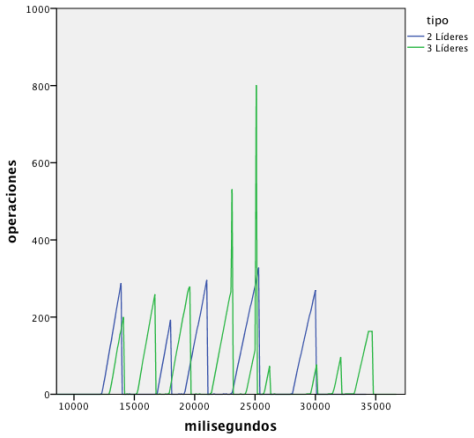


Figura 7.16: Comparación recolector de basura ejecutando 2 y 3 líderes

		tipo		
		2 Líderes	3 Líderes	sin GC
operaciones	Mínimo	0	0	0
	Máximo	328	801	2999
	Media	38	34	1249
	Mediana	0	0	1149

Figura 7.17: Valores de las operaciones almacenadas según el tipo de ejecución

Capítulo 8

Recuperación

La ejecución de la recuperación se realiza cuando un proceso se inicia dentro del sistema, o bien un proceso detecta que existe alguna operación existente en el sistema y que desconoce.

En la versión más simple de la recuperación, cuando un proceso se inicia pregunta a todos los procesos que ejecutan el mismo rol por su estado actual. Todos los procesos preguntados responderán con un mensaje que contendrá el estado de cada uno de ellos. El proceso actualizará su estado según los estados recibidos, procediendo entonces a su ejecución normal.

Según el tipo del proceso será necesario que la recuperación sea recibida por al menos uno de los procesos (como en el caso de las **réplicas**), o por la mayoría de los procesos (como en el caso de los **aceptores**).Los **líderes** ocupan un caso especial, pues reciben su estado de los mensajes de adopción enviados por los aceptores, no necesitando un protocolo de recuperación en si mismo.

En el sistema se supone que todos los mensajes se entregan siguiendo un orden total y la red es segura por lo que no se considera la perdida de mensajes, por ello en la siguiente descripción no se considera la recuperación de mensajes intermedios perdidos.

8.1. Recuperación de los aceptores

Debido a la naturaleza de los **aceptores**, es necesario que al menos existan en el sistema $(n/2) + 1$ **aceptores** con todas las operaciones decididas hasta un momento t, o en su defecto las operaciones decididas y las referencias a los slots recolectados mediante el *recolector de basura*.

De esta forma cuando un nuevo **aceptor** se conecta al sistema, es necesario que se actualice o recupere el estado que contienen el resto de los **aceptores**, de modo que en un instante t el nuevo **aceptor** haya realizado la recuperación y posea el mismo estado que la mayoría de los procesos.

8.1.1. Algoritmo de recuperación

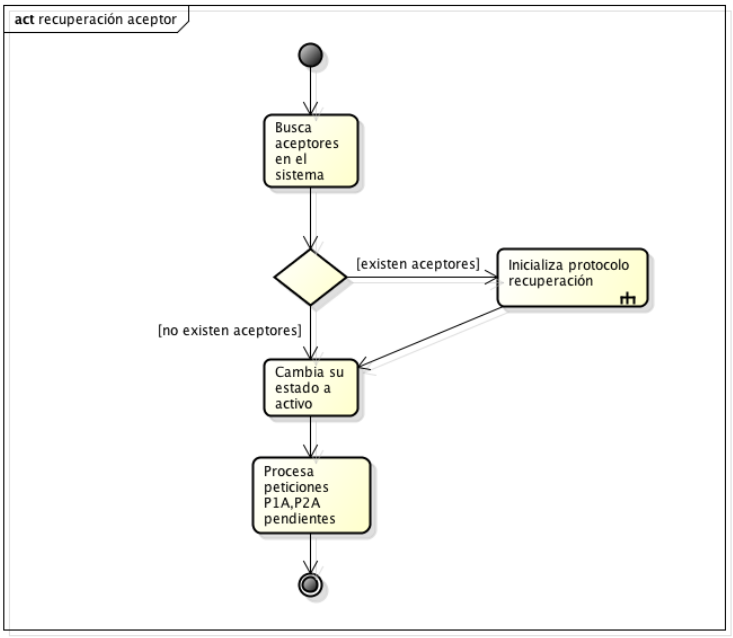


Figura 8.1: Diagrama Actividad: Recuperación de aceptor

1. Al conectarse, el **aceptor** espera a recibir la notificación de la existencia de otros **aceptores** dentro del sistema. Si no recibe notificación dentro de un intervalo de tiempo t , se supone que en ese instante existe únicamente ese aceptor y por lo tanto empieza a aceptar peticiones. Para ello cambia su estado a activo y si previamente ha recibido mensajes de algún **líder** estos son procesados.

2. Si existen más aceptores se procede a ejecutar el **protocolo de recuperación**.

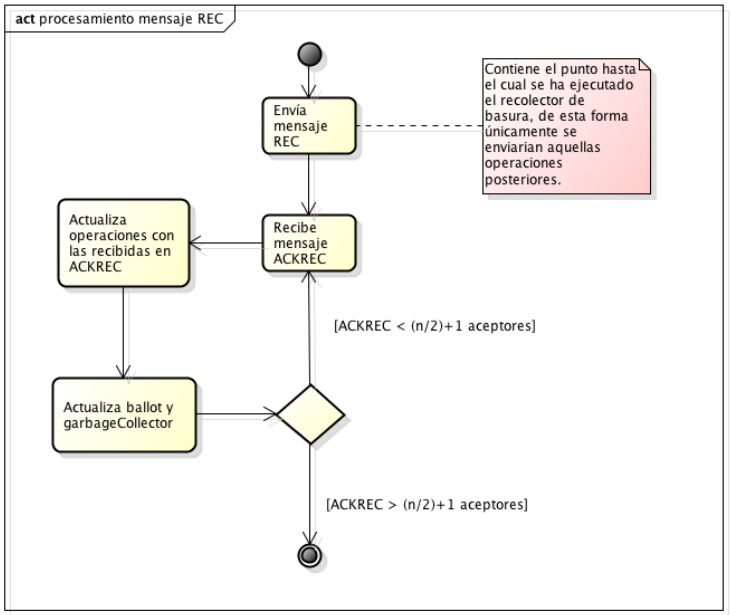


Figura 8.2: Diagrama Actividad: Recuperación de aceptor

1. Al iniciar el protocolo de recuperación el **aceptor** envía un mensaje *REC* a todos los aceptores. En este mensaje se envía el valor actual de su recolector de basura, de modo que únicamente reciba aquellas operaciones superiores al mismo. Este valor es útil debido a que el aceptor puede no haber perdido todo su estado, sino simplemente una parte del mismo, de esta forma puede pedir únicamente a partir de las operaciones pendientes.

2. Posteriormente al envío del mensaje *REC*, empezará a recibir mensajes de respuesta por parte de los aceptores, estos mensajes contendrán las operaciones almacenadas en cada uno de ellos, así como la posición de su recolector de basura. Una vez recibido el mensaje realiza las siguientes operaciones:

   a) Actualiza la lista de mensajes recibidos.

   b) Actualiza el valor del recolector de basura, eligiendo el valor más alto de todos los recibidos.

   c) Actualiza la lista de operaciones aceptadas con los valores recibidos.

3. Si ha recibido el mensaje de respuesta de la mayoría de los aceptores, entonces el algoritmo de recuperación finaliza su ejecución.

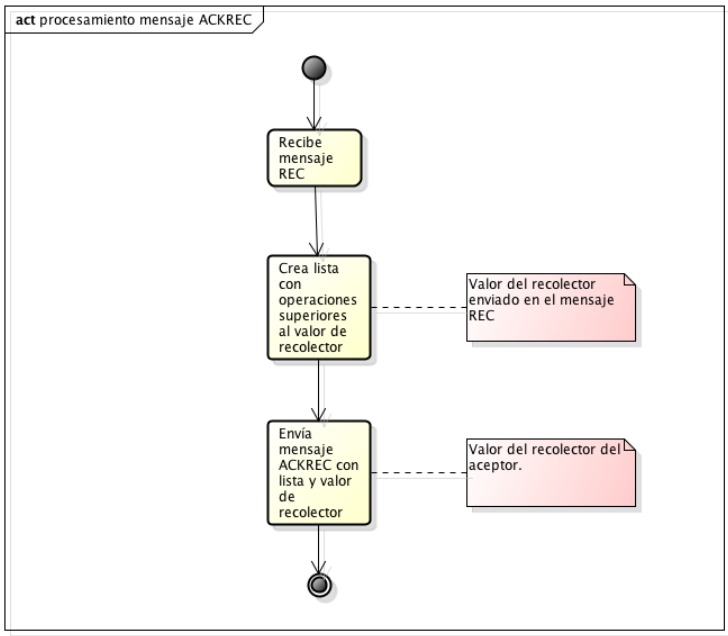


Figura 8.3: Diagrama Actividad: Recuperación de aceptor

En cualquier momento de su ejecución un **aceptor** puede recibir un mensaje de recuperación y ha de ser capaz de responder al mismo, las acciones que realiza ante su recepción son las siguientes:

1. Lee los campos enviados en el mensaje.

2. Crea una lista con todas las operaciones con un slot superior al valor del recolector de basura enviado en el mensaje.

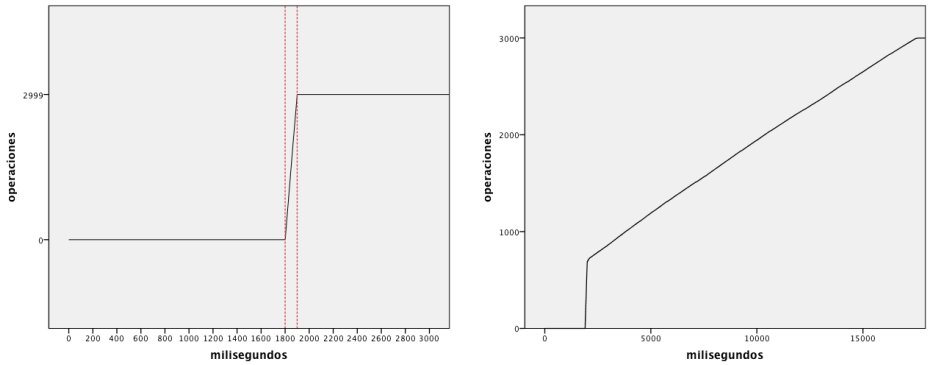
3. Envía un mensaje *ACKREC* con la lista de las operaciones y el valor de su propio recolector de basura.

8.1.2. Rendimiento

Para realizar las pruebas de rendimiento se han utilizando diferentes configuraciones del sistema, realizando variaciones entre 1 y 2 **líderes** y manteniendo siempre 2 **aceptores**. En un instante de tiempo t se introduce un nuevo **aceptor**, y este ha de recuperar el estado a partir de los **aceptores** que se encuentran en ejecución.

En la primera prueba se ejecutan 3000 operaciones dentro del sistema, compuesto por 1 **líder** y 2 **aceptores**, ambos con el **recolector de basura desactivado**. Una vez que todas las operaciones han sido procesadas se introduce

un nuevo **aceptor**, este ha de recuperar todas las operaciones y finalizar su ejecución con 3000 operaciones en su lista de operaciones aceptadas(*Figura(a)*).

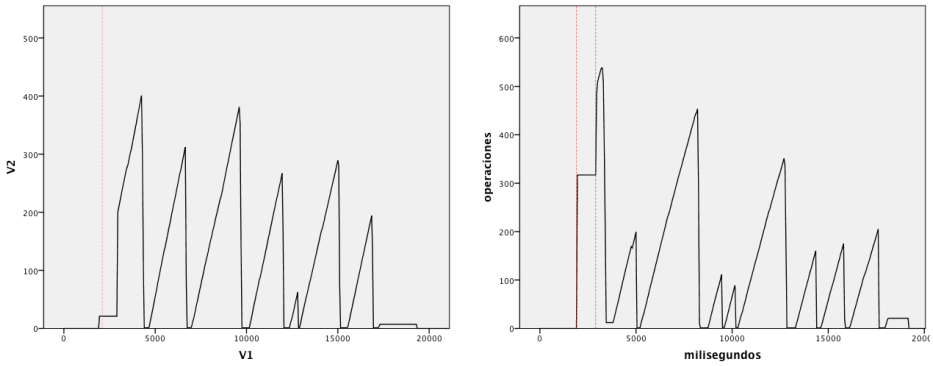


(a) (b)

Figura 8.4: Recuperación de aceptor con GC desactivado

En la *Figura(b)* se realiza la misma configuración que en el caso anterior, con la salvedad de que se introduce el **aceptor** una vez han sido procesadas aproximadamente unas 1000 operaciones, de esta forma se puede comprobar como el **aceptor** una vez recuperado puede procesar las nuevas operaciones de una forma normal.

A continuación se muestran varias gráficas, en ambas el sistema se encuentra compuesto por 2 **líderes** y 2 **aceptores**, añadiendo en un instante t un tercer **aceptor**. En la *Figura(a)* se ejecutan todos los **aceptores** con su funcionamiento normal.



(a) (b)

Figura 8.5: Recuperación de aceptor con GC desactivado

En la *Figura (b)* uno de los **aceptores no ejecuta su recolector de basura**, el **líder** siempre recibirá todas las peticiones antiguas y tendrá que volver a proponer las mismas ante los **aceptores**. Los **aceptores** que funcionan correctamente deberán ser capaces de filtrar dichas peticiones y ejecutar correctamente su recolector de basura.

En la siguiente figura uno de los **aceptores** no ejecuta su recolector de basura y los **líderes** realizan un filtrado de las operaciones inferiores al valor de su recolector.

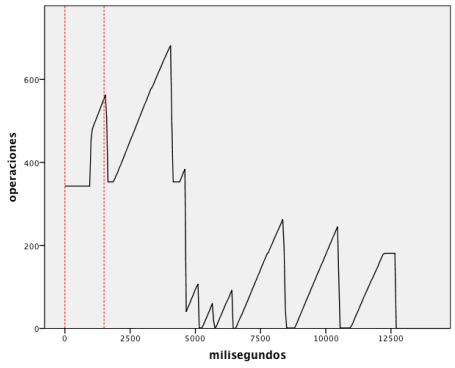


Figura 8.6: Recuperación de aceptor con GC desactivado

A continuación se muestra una comparativa de los tiempos necesarios para la recuperación de un **aceptor**, dependiendo del número de elementos que se encuentran almacenados en los **aceptores** y el número de los mismos que forman parte del sistema.

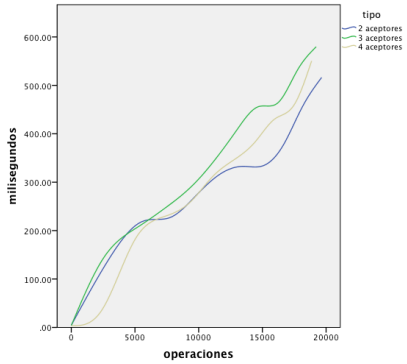


Figura 8.7: Tiempos de recuperación según el número de aceptores

Como se puede observar, cuando existen pocas peticiones almacenadas el tiempo de recuperación del **aceptor** se mantiene prácticamente idéntico en am-

bas configuraciones. Aún así a medida que aumentan el número de operaciones, las diferencias se van haciendo más palpables, pudiendo observar que a mayor número de aceptores dentro del sistema , mayor será el tiempo necesario para recuperar el estado de un **aceptor**.

8.2. Recuperación de los líderes

La recuperación de los **líderes** se encuentra intrínseca dentro del protocolo de comunicación con los **aceptores**. Cuando un líder se conecta al sistema o solicita un nuevo **ballot** para decidir operaciones, recibe por parte de los **aceptores** todas las operaciones que han recibido anteriormente, así como la posición actual de su propio recolector de basura. Los líderes gracias a estos datos son capaces de recuperar el estado actual del sistema y continuar con su funcionamiento normal.

8.2.1. Rendimiento

En la siguiente gráfica se muestra el tiempo invertido por el **líder** en realizar la recuperación, desde el momento en el cual envía la petición a los **aceptores**, hasta el momento en el que recibe la respuesta. Tal como se muestra en la gráfica, el tiempo consumido es linealmente dependiente del número de operaciones que se recuperan, lo cual puede llevar a un grave problema ya que a mayor número de operaciones a recuperar, mayor será el tiempo necesario, y por lo tanto mayor será el tiempo en el cual el sistema se encontrará sin poder decidir nuevas operaciones.

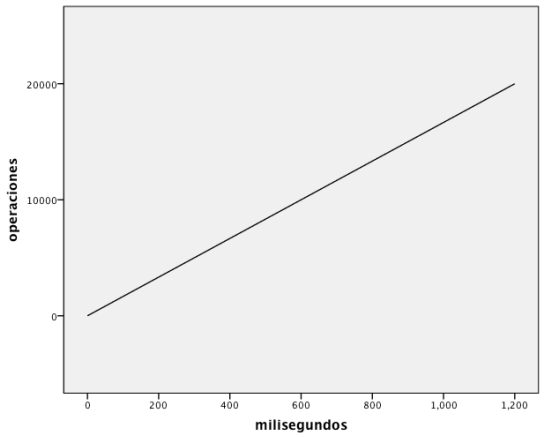


Figura 8.8: Tiempo de recuperación de líder

En la siguiente gráfica se comparan los tiempos consumidos en la recuperación del **líder** en función del número de **aceptores** que forman parte del sistema.

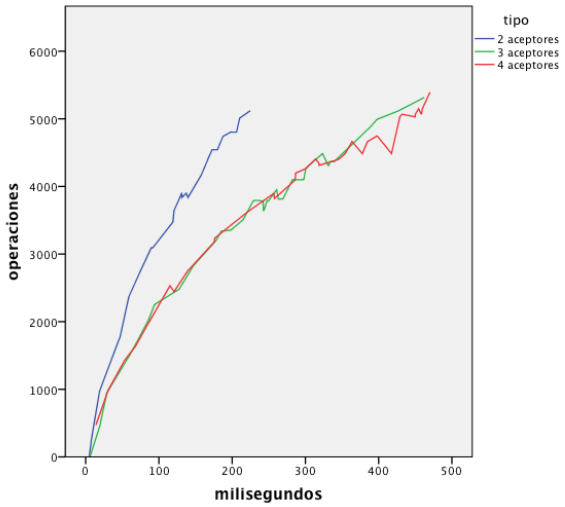


Figura 8.9: Tiempos de recuperación según el número de aceptores

A mayor número de **aceptores** mayor es el tiempo necesario para realizar la recuperación, también se puede observar como el tiempo consumido con 3 y 4 **aceptores** es prácticamente idéntico. Estos hechos son debidos a que para realizar la recuperación, el líder ha recibir un mensaje *P1B* de al menos la mayoría de los **aceptores** del sistema, por lo tanto a mayor número de **aceptores** mayor es el número de mensajes necesarios para finalizar la recuperación.

8.3. Recuperación de las réplicas

La recuperación de las **réplicas** conlleva la recuperación más complicada dentro del sistema. Todas las **réplicas** eventualmente han de contener el mismo estado, por lo que la recuperación ha de realizarse de tal forma que se consiga que una nueva **réplica**, o una **réplica** que ha perdido un número indeterminado de operaciones, consiga alcanzar el mismo estado que el resto de las **réplicas** que forman parte del sistema.

Para conseguir la recuperación de las réplicas se pueden realizar dos aproximaciones:

- **Recuperación a partir del estado**: Cuando una **réplica** necesita realizar su recuperación, pide al resto de las **réplicas** del sistema que le envíen su estado actual. Cada una de las **réplicas** preguntadas le enviarán su estado. La **réplica** tendrá que sustituir su estado actual por el estado recibido.

  La **ventaja** de esta aproximación es que no es necesaria realizar ninguna operación adicional a parte del envío y recepción del estado. Una vez recibido el estado, la **réplica** ya se encuentra actualizada y puede continuar

con su ejecución.

La **desventaja** radica en que es el propio programador de la aplicación el que debe definir cual es el estado que se debe enviar, y la forma de enviar y de procesar el mismo en el receptor.

- **Recuperación a partir de las operaciones**: En este caso , las **réplicas** envían al receptor el conjunto de todas las operaciones ejecutadas dentro de cada una de ellas, indicando el orden en el cual fueron ejecutadas. De esta forma cuando la **réplica** recibe el mensaje ejecuta por orden dichas operaciones, alcanzando el estado deseado en el momento que ha ejecutado todas ellas.

  La **ventaja** de esta aproximación es que el programador de la aplicación no debe preocuparse por la recuperación de la misma, ya que esta se realiza de manera transparente al programa.

  La **desventaja** radica en que se han de ejecutar todas y cada una de las operaciones pendientes, por lo que el tiempo de recuperación puede ser superior al necesario en la recuperación a partir del estado.

Aunque ambos tipos de recuperación puedan parecer excluyentes entre si, las **réplicas** pueden utilizar uno y otro dependiendo del tipo de recuperación que necesiten en un momento dado, por ejemplo:

- Una nueva **réplica** se añade al sistema, no contiene ningún estado y necesita recuperar todas las operaciones que han sido procesadas hasta el instante de su conexión. La recuperación ideal sería la **recuperación a partir del estado**, se consigue actualizar la **réplica** hasta el estado actual y podrá proseguir con un funcionamiento normal a partir de dicho punto.

- Una **réplica** se mantiene en el sistema pero algunas operaciones no son entregadas a la misma, existiendo huecos en el procesamiento, impidiendo de esta forma su avance. En este caso la recuperación ideal sería la **recuperación a partir de las operaciones**. La **réplica** realizaría una petición de sus operaciones pendientes y las ejecutaría en el orden pertinente. Una vez realizado este paso la réplica podrá proseguir su ejecución de forma normal.

8.3.1. Algoritmo de recuperación

En los siguientes diagramas se muestran los dos casos de recuperación que se pueden dar en las **réplicas** y cual es el algoritmo utilizado para cada uno de estos casos. En el primer caso se muestra como se recupera una **réplica** cuando alguno de los **líderes** no ha ejecutado su recolector de basura. En el segundo caso se muestra como se recupera la **réplica** cuando todos los líderes han ejecutado su recolector de basura. En ambos casos se utiliza la **recuperación a partir de las operaciones**.

Algoritmo líderes que no han ejecutado recolector de basura

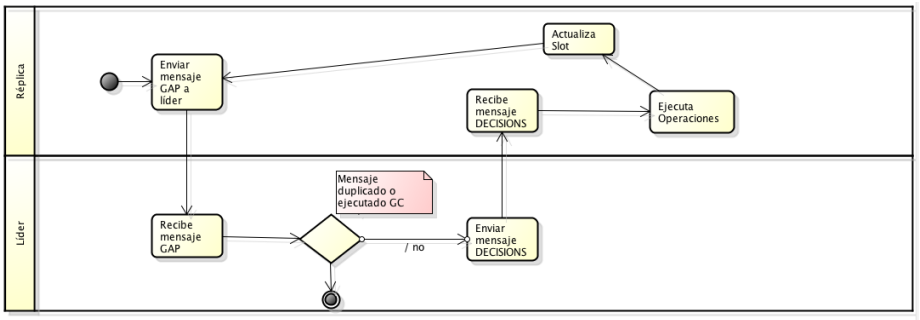


Figura 8.10: Diagrama de Actividad: Recuperación de operaciones con Lider sin ejecutar GC

1. Cuando la **réplica** se añade al sistema envía a todos los líderes del sistema un mensaje **GAP** conteniendo su identificador, el último *slot* que ha procesado y el último *slot* al cual le ha llegado una operación decidida. Dado que la **réplica** se acaba de conectar al sistema, el último *slot* procesado será cero y el último *slot* con una operación decidida será *undefined*.

2. El **líder** recibe el mensaje **GAP** y comprueba si el intervalo de operaciones contenidas en los *slots* del mensaje no han sido eliminados por el recolector de basura. Si este es el caso genera un mensaje **DECISIONS**, el cual contiene una lista con todas las operaciones decididas para dichos *slots*. Una vez creado el mensaje, lo envía a la réplica.

3. La **réplica** recibe el mensaje **DECISIONS** y procesa cada una de las operaciones en el orden indicado por los *slots* a los cuales pertenecen. Una vez procesadas las peticiones actualiza el **último slot procesado** con el valor del *slot* más alto de las operaciones recibidas.

4. La **réplica** envía periódicamente un mensaje **GAP** a los líderes pare recibir operaciones pendientes.

Algoritmo líderes que han ejecutado recolector de basura

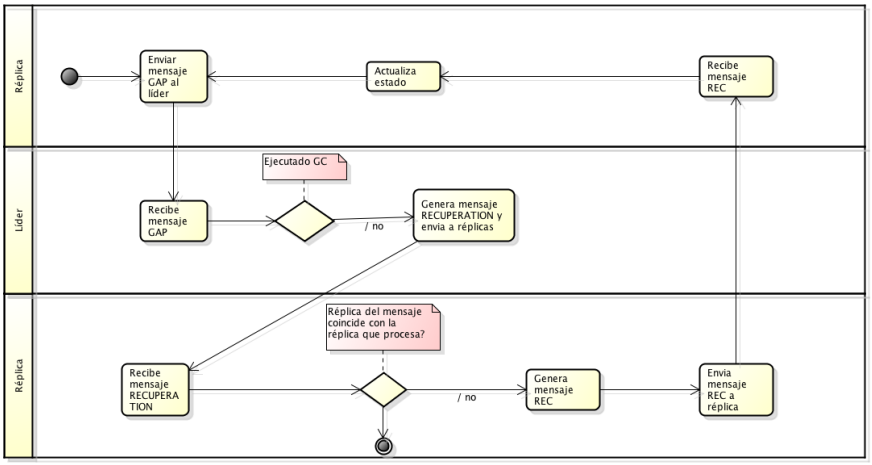


Figura 8.11: Diagrama de Actividad: Recuperación de operaciones con Lider ejecutado GC

1. Cuando la **réplica** se añade al sistema envía a todos los líderes conocidos un mensaje **GAP** conteniendo su identificador, el último *slot* que ha procesado y el último *slot* al cual le ha llegado una operación decidida. Dado que la **réplica** se acaba de conectar al sistema, el último *slot* procesado será cero y el último *slot* con una operación decidida será *undefined*.

2. El **líder** recibe el mensaje **GAP** y comprueba si el intervalo de operaciones contenidas en los *slots* del mensaje han sido eliminados por el recolector de basura. De ser así, genera un mensaje **RECOLECT** conteniendo el intervalo de los *slots* a recuperar y la dirección de la **réplica** que realiza la petición de recuperación. A continuación envía dicho mensaje a todas las **réplicas** conocidas dentro del sistema.

3. Cuando una **réplica** recibe el mensaje **RECOLECT** comprueba que la réplica indicada no es ella misma. Si no es ella misma genera un mensaje *REC*, conteniendo las operaciones que ha procesado así como los valores de estado necesarios para el correcto funcionamiento de la réplica. Una vez generado el mensaje es enviado a la **réplica** que se encuentra en recuperación.

4. La **réplica** en recuperación recibe el mensaje **REC** enviado por las **réplicas**. Ejecuta cada una de las operaciones enviadas en los mensajes, siempre y cuando no hayan sido procesadas anteriormente, y actualiza las variables de estado.

5. La **réplica** envía periódicamente un mensaje **GAP** a los líderes pare recibir operaciones pendientes.

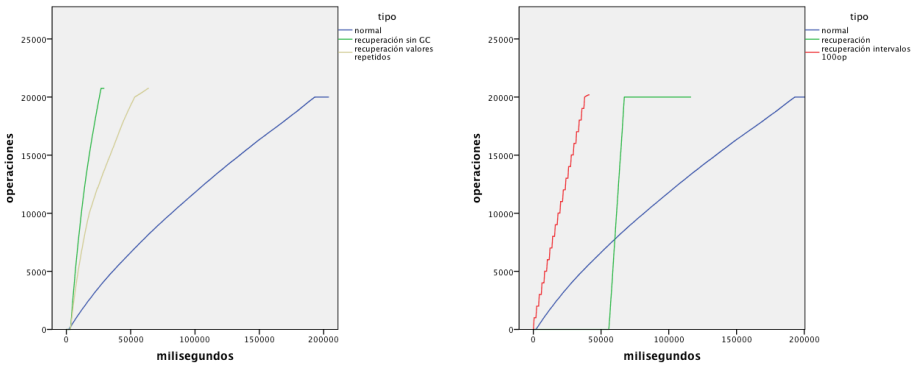
8.3.2. Rendimiento

A continuación se muestran las gráficas de los rendimientos obtenidos al ejecutar el algoritmo de recuperación en las **réplicas**.

Recuperación sin recibir nuevas peticiones

En las siguientes gráficas se muestran los tiempos de recuperación de una **réplica** que se añade al sistema y tiene que actualizar su estado a partir de las **réplicas** ya existentes dentro del sistema. En dichas pruebas una vez que la **réplica** se ha actualizado en el sistema no recibe nuevas peticiones, y tampoco recibe peticiones mientras se encuentra en el estado de recuperación. Para realizar las pruebas se han medido los tiempos de recuperación utilizando dos supuestos:

- Los **líderes** que forman parte del sistema **no han ejecutado el recolector de basura** y por lo tanto contienen las operaciones que han sido aceptadas. En este caso la **réplica** recibirán directamente las operaciones de los **líderes**.
- Los **líderes han ejecutado el recolector de basura**. Las **réplicas** han de enviar sus operaciones a la nueva **réplica** para que proceda a su recuperación.



(a) (b)

Figura 8.12: (a)Recuperación de una réplica con líder sin ejecutar su GC. (b) Recuperación de una réplica con líder y GC ejecutado

En la figura (a) se muestran los tiempos necesarios para recuperar 20.000 operaciones. Las mediciones se han realizado en los siguientes supuestos:

- Una **réplica** nueva realiza peticiones de manera secuencial y todas ellas son decididas y procesadas de manera correcta. Es el funcionamiento normal de las **réplicas**.

- Una **réplica** se añade al sistema y recupera todas las peticiones que han sido decididas anteriormente.

- Una **réplica** se añade al sistema, intenta proponer operaciones y el sistema le retorna aquellas operaciones que ya han sido decididas para las posiciones que intenta proponer.

Debido a que el **líder** no ha ejecutado el recolector de basura, cuando una **réplica** realiza la petición de recuperación recibe directamente las operaciones solicitadas, pudiendo en dicho momento realizar la recuperación de todas ellas. En el caso de que la **réplica** al mismo tiempo se encuentre realizando peticiones duplicadas, se producirá un ligero aumento del tiempo necesario, ya que el **líder** ha de indicar a la réplica que dichos valores ya han sido decididos.

En todo caso, aunque se produzcan colisiones, el tiempo de recuperación es notablemente inferior al tiempo consumido por el procesamiento normal de las operaciones, lo que hace indicar que una **réplica** que se encuentre en recuperación, podrá eventualmente alcanzar el estado final del sistema y aceptar peticiones de manera normal una vez haya finalizado dicho estado de recuperación.

A diferencia de la figura anterior, en la figura (b) actual se muestra la recuperación de las **réplicas** cuando el **líder** ya ha ejecutado su recolector de basura. Debido a que el **líder** ya no contiene los datos de las operaciones, la **réplica** ha de comunicarse con el resto de las **réplicas** para que estas le envíen las operaciones que han sido procesadas.

Como se puede observar, si se intentan enviar todas las peticiones procesadas en un mismo mensaje, el tiempo que conlleva la creación y el envío del mismo es superior al tiempo necesario para procesar todas las operaciones, aún así el tiempo total entre el envío y el procesamiento de la recuperación sigue siendo notablemente inferior al tiempo consumido por el procesamiento normal.

Para paliar el problema de latencia entre el envío y la recepción de los mensajes de recuperación, se propone realizar una recuperación escalonada. En vez de enviar en un único mensaje todas las operaciones, se envía un número n de las mismas de forma que el tiempo necesario para la creación del mensaje sea inferior, y por lo tanto en teoría la recuperación se realice en un espacio de tiempo menor.

Esto se puede observar en la línea de color rojo, en la cual se ha realizado una prueba realizando la recuperación con mensajes que contienen como máximo 1000 operaciones. De esta forma el tiempo de latencia es prácticamente inexistente en comparación con el caso anterior, y la finalización de la recuperación finaliza prácticamente en el momento en el cual comienza el procesamiento de las operaciones del anterior caso.

Tiempo de recuperación recibiendo nuevas peticiones

En la siguiente gráfica se muestra la progresión de una **réplica** en recuperación con respecto a una **réplica** que se ejecuta de manera normal. Para realizar la gráfica se han realizado 5.000 operaciones de forma secuencial y se ha ejecutado una nueva **réplica** cuando ya habían procesado aproximadamente 500 operaciones.

De esta forma la réplica deberá recuperar estas operaciones pendientes y proseguir con las operaciones posteriores. Como se puede observar, al inicio de la ejecución la réplica realiza varías recuperaciones seguidas, ya que ha de recuperar el estado inicial perdido y a continuación las operaciones que no ha procesado mientras se mantenía en el estado de recuperación. Debido a este proceso, mientras se recupera es posible que algunas operaciones recibidas no sean procesadas y por lo tanto posteriormente deban ser recuperadas, lo cual explica las pequeñas escaleras que se aparecen a lo largo de la gráfica. Aún así se puede observar como la réplica en recuperación, en un corto periodo de tiempo es capaz de alcanzar el estado de la otra réplica y proseguir con su procesamiento normal.

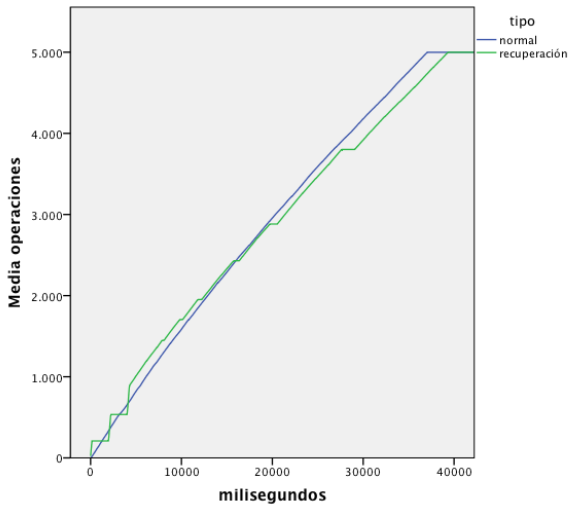


Figura 8.13: Recuperación réplica recibiendo nuevas peticiones

Capítulo 9

Getting Start

En este capítulo se muestran los pasos a seguir para configurar el sistema, de manera que este se ejecute de forma satisfactoria. Finalmente se muestra como crear una aplicación de usuario que haga uso de las características que ofrece el sistema desarrollado.

9.1. Desplegar el Sistema

En este apartado se muestra como configurar cada una de las máquinas para que ejecuten un rol determinado dentro del sistema y realicen su propósito de manera satisfactoria.

9.1.1. Pre-instalación

En primer lugar es necesario instalar la aplicación dentro de cada uno de los dispositivos sobre los que se va a ejecutar el sistema. Para que la instalación se realice de manera satisfactoria el dispositivo ha de tener cumplir las siguientes características.

- Poseer un sistema operativo basado en *GNU/Linux* , *Windows* o *MacOS*.[1].
- Tener instalados *compiladores para C y C++*[3]. Dichos compiladores serán necesarios para realizar la compilación del código de **ZeroMQ**.
- Tener instalada la librería de **ZeroMQ**[2].
- Tener instalado **NodeJs**[8] y **CoffeeScript**[1].

Con el fin de simplificar la instalación de las dependencias, dentro de la carpeta raíz de la aplicación se encuentra el fichero *setup.sh* el cual realiza de forma automática la instalación de todas las dependencias.

9.1.2. Estructura

- **conf**: Ficheros de configuración.
- **lib**: Ficheros NodeJS compilados a partir del código fuente en CoffeeScript.
- **log**: Ficheros de log generados por la aplicación.
- **measured**: Ficheros con las estadísticas de uso de la aplicación.
- **node_modules**: Dependencias de la aplicación.
- **src**: Código fuente de la aplicación.

[1] La aplicación únicamente ha sido probada en sistemas Debian y MacOS

9.1.3. Instalación

Una vez que se han cumplido todos los requisitos para la instalación de la aplicación, es la hora de instalar la aplicación en sí. La instalación permite instalar todas las librerías que utiliza el sistema de manera automática sin tener que realizar la instalación de dichas librerías de forma manual.

Para realizar la instalación de todas las dependencias de la aplicación, desde la terminal de sistema hay que introducir el siguiente comando, siempre estando en el directorio raíz de la aplicación.

```
npm install
```

Con dicha instrucción **npm** se encarga automáticamente de instalar todas las dependencias que tiene la aplicación. Para poder observar las dependencias y el resto de datos de la aplicación se puede observar el fichero *package.json*. Las dependencias instaladas son las siguientes:

- **zmq**: Librería usada para utilizar **ZeroMQ** desde NodeJS.
- **Q**: Librería usada para añadir el soporte de **promesas** en NodeJS.
- **hashmap**: Librería que permite el uso de **HashMaps** en NodeJS.
- **winston**: Librería usada para crear un logger de cada uno de los procesos.
- **measured**: Librería usada para realizar mediciones.
- **usage**: Librería usada para medir el uso de CPU y memoria de la aplicación. Únicamente es compatible con sistemas *GNU/Linux* y *Mac*.

9.1.4. Configuración

La configuración del sistema se realiza mediante un fichero alojado en la carpeta *conf* y de nombre **paxos.config**. Este fichero se ha de encontrar en todas las instancias que se ejecuten dentro del sistema y ha de encontrarse correctamente cumplimentado, para que tanto la instancia como el sistema puedan funcionar de forma satisfactoria. El fichero se encuentra en formato JSON y contiene la siguiente estructura:

```
{"id":"hostName",
        "roles":[],
        "logger":false,
        "stealth":false,
        "settings":
        {
        },
        "network":{

        }
}
```

Cada uno de los campos tiene el siguiente significado:

- **id**: Identificador de la máquina o del proceso que se va a ejecutar. Tiene que ser un identificador único y que pueda ser identificado por todos los procesos que forman parte del sistema. Usualmente será la IP de la máquina.
- **roles**: Array que contiene los roles que van a ser implementados por el proceso. Cada uno de los procesos puede implementar de 0 a 3 roles, siendo los roles **acceptor** , **leader** y **replica**. Por ejemplo si se desea que el proceso ejecute los roles de **réplica** y **líder**, el campo de roles debería ser:

```
"roles":["leader","replica"]
```

- **stealth**: Valor booleano con valores **true** o **false**. Cuando se indica con el valor de **true** el proceso se ejecuta en modo sigiloso; es decir, por defecto el proceso no implementa ningún rol realizando únicamente la monitorización de los procesos que se encuentran dentro del sistema. Según el estado que vaya alcanzado el sistema a lo largo del tiempo, el proceso puede implementar automáticamente un nuevo rol con el fin de asegurar la continuidad del sistema.
- **logger**: Valor booleano con valores **true** o **false**. Indica si se va a utilizar el **logger**. En caso de utilizar el **logger** los ficheros generados se almacenan en la carpeta *logs*, la cual se encuentra en el directorio raíz de la aplicación.
- **settings**: Contiene la configuración de cada uno de los roles que van a ser implementados por el proceso. Se indican los protocolos y puertos de comunicación de cada uno de los roles.
- **network**: Contiene la configuración global del sistema. En él se indican las propiedades de la red sobre la cual se va a implementar el sistema y las características que ha de cumplir nuestro sistema dentro de la propia red. **La configuración de este apartado ha de ser el mismo en todos los procesos**.

Settings

En **settings** se realiza la configuración de cada uno de los roles que van a ser implementados por el proceso. De esta forma si el proceso va a implementar a un **líder** y una **réplica** en este apartado se ha de indicar cuales serán los *endpoints* de entrada de cada uno de ellos.

```
{
"settings":{
        "acceptor":{
                "publisher":{},
                "rec":{}
        },
        "replica":{
```

```
                "router":{},
                "publisher":{},
                "rec":{},
                "program":{}
        },
        "leader":{
                "publisherToReplicas":{},
                "publisherToAcceptors":{},

        },
}
```

Las configuraciones a introducir para cada uno de los roles son las siguientes:

acceptor Se indican los *endpoints* de entrada que utiliza el rol **aceptor**.

- **publisher**: Configuración del *endpoint* publicador por el cual el **aceptor** se comunica con los **líderes**.
- **rec**: Configuración del *endpoint* utilizado por el algoritmo de recuperación.

replica Se indican los *endpoints* de entrada que utiliza el rol **réplica**.

- **router**: Configuración del *endpoint* utilizado para comunicar a la **réplica** con los clientes.
- **publisher**: Configuración del *endpoint* utilizado para realizar la comunicación entre la **réplica** y los **líderes**.
- **rec**: Configuración del *endpoint* utilizado por el algoritmo de recuperación.
- **program**: Configuración del *endpoint* utilizado por la réplica para comunicarse con el programa encargado de procesar las operaciones.

leader Se indican los *endpoints* de entrada que utiliza el rol **líder**.

- **publisherToReplicas**: Configuración del *endpoint* utilizado para comunicar al **líder** con las **réplicas**.
- **publisherToAcceptors**: Configuración del *endpoint* utilizado para comunicar al **líder** con los **aceptores**.

Cada uno de los *endpoint* sigue la siguiente estructura:

```
{
        "protocol":"",
        "address":"",
        "port":""
}
```

- **protocol**: Protocolo utilizado por el *endpoint*, actualmente se soportan *tcp* e *ipc*.
- **address**: Dirección desde la cual se van a aceptar peticiones. Para aceptar peticiones de cualquier dirección se ha de introducir el valor «*». Si se utiliza el protocolo *ipc* es necesario introducir una dirección de disco como por ejemplo */tmp/endpoint*.
- **port**: Puerto de escucha. Únicamente es necesario si se utiliza el protocolo *tcp*.

En el caso del *endpoint* de **program** se añade un nuevo parámetro llamado **type**. Dado que la **réplica** permite que la comunicación con la aplicación encargada de procesar las operaciones, se pueda realizar tanto mediante paso de mensajes como por código, en este campo se introduce el tipo de comunicación ha realizar.
Si el valor de **type** es **code** no es necesario rellenar el resto de los campos. En caso de que tenga un valor diferente será necesario rellenar cada uno de los valores de los campos indicados anteriormente.

```
{
        "program":
        {
                "type":"code|message",
                "protocol":"ipc",
                "address":"/tmp/paxos"
        }
}
```

Network

En **network** se definen las configuraciones globales del sistema. En él se indican las propiedades de la red sobre la cual se va a implementar el sistema, y las características que ha de cumplir nuestro sistema dentro de la propia red.

Hay que tener en cuenta que todos los procesos han de tener la misma configuración de la red, de no ser así el comportamiento del sistema podría no ser el esperado.

```
{
        "network":{
                "broadcast":"192.168.0.255",
                "domains":[["192.168.0.2","192.168.0.100"],["127.0.0.1"],["localhost"]],
                "heartbeat":10000,
                "stealth_time":15000,
                "min_acceptors":2,
                "min_leaders":1,
        }
}
```

- **broadcast**: Dirección en la cual se realiza el *broadcast*, es necesario para implementar el protocolo de descubrimiento y monitorización de los procesos que se encuentran en el sistema. Si este valor no se encuentra bien configurado el proceso no podrá realizar la comunicación con el resto de los procesos.

- **domains**: Define los diferentes dominios de fallos a los que pertenece el proceso. Se utilizan para realizar la monitorización y la aceptación de operaciones dependientes del dominio.

- **heartbeat**: Intervalo de tiempo en el cual se envía un mensaje a la dirección de broadcast con los datos de configuración del proceso. El intervalo se encuentra definido en milisegundos.

- **stealth_time**: Intervalo de tiempo en el cual un proceso encubierto comprueba el sistema. El intervalo se encuentra definido en milisegundos.

- **min_acceptors**: Indica el mínimo número de **aceptores** que son necesarios dentro del sistema para asegurar la continuidad del mismo. Si el proceso se encuentra marcado como **stealth** y el número de **aceptores** que se encuentra en el sistema es inferior al dado, el proceso puede decidir implementar dicho rol para cumplir la condición.

- **min_leaders**: Definición idéntica a **min_acceptors** con la salvedad de referirse a los **líderes**.

9.2. Crear aplicación

En este apartado se muestra como crear una aplicación que utilice los mecanismos implementados en el sistema, con el fin de que desplegar una aplicación altamente disponible.

Las aplicaciones creadas para ser desplegadas en el sistema se han de componer de dos partes: Una parte cliente, encargada de capturar las peticiones enviadas por el cliente, y enviarlas a la **réplica**. Una parte servidora encargada de procesar las operaciones que envía la **réplica**.

Como ejemplo de una aplicación se puede suponer un *servicio web* que ofrece sus servicios mediante *REST*. La aplicación cliente sería el servidor web que ofrece los servicios REST al usuario. La aplicación recibe las peticiones del usuario, las transforma en el formato esperado por el sistema y las envía a la **réplica** o **réplicas**. La aplicación servidora sería aquella parte de la aplicación encargada de procesar la operación y generar un resultado de la misma, como por ejemplo el almacenar un pedido en una base de datos.

Ambas partes de la aplicación se pueden desarrollar de manera conjunta o separada, si únicamente va a existir un punto de entrada a la aplicación y este se conectará con todas las instancias de las **réplicas**, la parte cliente de la aplicación se podrá ejecutar en cualquier tipo de maquina, mientras que la parte servidora se tendrá que ejecutar obligatoriamente en cada una de las máquinas que ejecuten el rol de **réplica**.

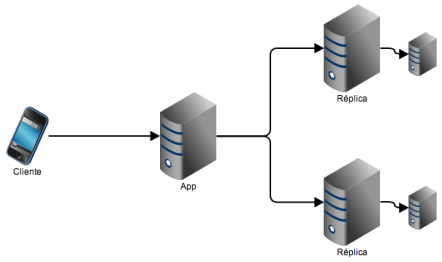


Figura 9.1: Acceso mediante una aplicación a múltiples réplicas

A su vez, es posible implementar la parte cliente de la aplicación en cada una de las máquinas que implementan el rol de **réplica**, replicando de esta forma también la parte cliente de la aplicación. De esta forma un cliente puede elegir individualmente a que réplica realizar la conexión, aumentando la alta disponibilidad de la aplicación implementada.

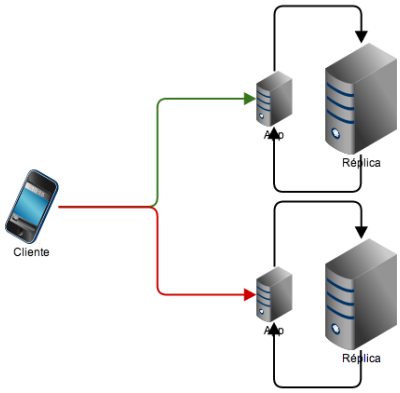


Figura 9.2: Replicación de la aplicación en cada réplica

A continuación se describe como crear una aplicación en **CoffeeScript** que sea compatible con el sistema:

- Se ha de crear un fichero cuyo nombre ha de ser **app.coffee** y dentro de el implementar una clase llamada **App**.
- Dicha clase ha de heredar, de la clase **Connect** en el caso de que la parte servidora utilice comunicación mediante paso por mensajes, o de **ConnectApp** en el caso de que la parte servidora utilice la comunicación mediante el propio código de la réplica.
- Si se implementa la parte servidora se ha de crear un método llamado **execute**. Dicho método recibe un parámetro que será la operación en-

viada por la **réplica**. En este método se realizan todas las operaciones dependientes del programa.

Como se muestra en la figura siguiente, al recibir el mensaje se ha de comprobar el tipo de mensaje recibido, realizar la operación y retornar la respuesta generada.

```
execute:(  operation ) ->
   switch operation.type
      when 'SUM'
      return sum operation.values[0] , operation.values[1]
```

- Si se implementa la parte cliente, se han de utilizar el método *sendRequest*, el cual envía a la **réplica** o **réplicas** la operación implementada.

  Dicho método recibe como parámetros la operación que deseamos enviar al sistema y el tipo de operación (lectura o escritura), finalmente retorna una promesa con la resolución de la operación.

```
resultado = ( result ) =>
        console.log result

operation = {type:'SUM',values:[@value,++@value]}
@sendRequest(operation , Connect.WRITE).then resultado
```

A continuación se muestra el código completo de una aplicación que implementa la parte cliente y servidora. La aplicación simplemente realiza una suma y resta de dos números dados, enviando una nueva petición una vez se ha procesado la anterior; el objetivo de dicha aplicación es mostrar de forma simple como implementar una aplicación para que sea ejecutada en el sistema.

```
Connect = require('./connectApp').ConnectApp

App = exports? and exports or @App = {}

class App.App extends Connect

   value : 0

   constructor:( options ) ->
      super( options )
      setTimeout @enviar , 5000


   enviar:( ) =>
      resultado = ( result ) =>
         console.log result
         @enviar()

      operation = {type:'SUB',values:[++@value,@value-1]}
      @sendRequest(operation , Connect.WRITE).then resultado


   execute:( operation ) ->
      switch operation.type
         when 'SUM'
            return @sum operation.values[0] , operation.values[1]
         when 'SUB'
            return @sum operation.values[0] , - parseInt(operation.values[1])
```

```
sum:(value1 , value2) ->
   return value1 + value2
```

9.2.1. Configuración

Para que el cliente funcione de manera correcta es necesario configurar cual es la dirección de la réplica o réplicas a las que se desea conectar. Esta configuración se puede realizar bien a través un fichero de configuración en formato *JSON* o mediante código.

9.2.2. Fichero de configuración

Para realizar la configuración mediante fichero es necesario crear un fichero llamado *client.config* en la carpeta *./conf*. Dicho fichero contendrá los siguientes campos:

- **id**: Identificador único utilizado por el cliente.
- **timeout**: Valor en milisegundos del tiempo que ha de esperar el cliente para dar por perdida una petición. Una vez que se ha producido el timeout se produce un reenvío de la petición.
- **replicas**: Su valor es un array conteniendo el *endpoint* de entrada de cada una de las réplicas a las que se van a realizar las peticiones. Este campo es opcional en el caso de que el cliente se ejecute en el mismo equipo que la réplica, en este caso la conexión se realiza utilizando los datos que se encuentran en el fichero de configuración del sistema.

```
{
        "id":"localhost",
        "timeout":25000,
        "replicas":["tcp://127.0.0.1:8000"]
}
```

9.2.3. Configuración mediante código

Para configurar la réplica mediante código es necesario definir estos valores mediante un mapa en el constructor del cliente. El mapa contendrá la misma estructura que el fichero de configuración, de esta forma el código de llamada sería como se muestra a continuación.

```
class Test extends ConnectApp

        constructor:( args ) ->
                super(args)

arguments = {id:'192.168.0.100',timeout:4000,replicas:['192.168.0.101','192.168.0.102']}
test = new Test(arguments)
```

Bibliografía

[1] CoffeeScript. *CoffeeScript*. 2013. URL: http://coffeescript.org/.

[2] iMatix Corporation. *ZeroMQ*. 2007. URL: http://www.zeromq.org/.

[3] GNU. *Gcc*. 2013. URL: http://gcc.gnu.org/.

[4] Leslie Lamport. «How to make a multiprocessor computer that correctly executes multiprocess programs». En: *Computers, IEEE Transactions on* 100.9 (1979), págs. 690-691.

[5] Leslie Lamport. «Paxos made simple». En: *ACM SIGACT News*. ACM SIGACT News (Distributed Computing Column) 32.4 (2001), págs. 18-25. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.69.3093%5C&rep=rep1%5C&type=pdf%20http://www.cs.utexas.edu/users/lorenzo/corsi/cs380d/past/03F/notes/paxos-simple.pdf.

[6] Leslie Lamport. «The part-time parliament». En: *ACM Trans. Comput. Syst.* 16.2 (mayo de 1998), págs. 133-169. ISSN: 0734-2071. DOI: 10.1145/279227.279229. URL: http://doi.acm.org/10.1145/279227.279229.

[7] Leslie Lamport, Robert Shostak y Marshall Pease. «The Byzantine generals problem». En: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), págs. 382-401.

[8] NodeJs. *NodeJs*. 2013. URL: http://www.nodejs.org/.

[9] Brian M Oki. *Viewstamped replication for highly available distributed systems*. Inf. téc. DTIC Document, 1988.

[10] Robbert Van Renesse. «Paxos made moderately complex». En: (2011). URL: http://www.cs.cornell.edu/courses/CS6452/2012sp/papers/paxos-complex.pdf.

[11] Martin Sustrik. *0mq-3-0-dealer-and-router*. 2011. URL: http://zeromq.org/tutorials:dealer-and-router.

[12] Martin Sustrik. *0mq-3-0-pubsub*. 2011. URL: http://www.zeromq.org/whitepapers:0mq-3-0-pubsub.