Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA METROPOLITANA

PCyTI

UNIVERSIDAD AUTÓNOMA METROPOLITANA - IZTAPALAPA
DIVISIÓN DE CIENCIAS BÁSICAS E INGENIERÍA

DISEÑO DE UN MIDDLEWARE TOLERANTE A FALLAS BASADO EN EL PROTOCOLO PAXOS

Tesis que presenta
Ricardo Adán Madrid Trejo
Para obtener el grado de
Maestro en ciencias y tecnologías de la información

Asesor: Dr. Ricardo Marcelín Jiménez

Jurado calificador:

Presidente: Dr. Sergio Rajsbaum Gorodezky

Secretaria: Dra. Graciela Román Alonso

Vocal: Dr. Ricardo Marcelín Jiménez

México, D.F. Noviembre 2015

AGRADECIMIENTOS

Quiero agradecer con todo mi corazón a mis padres por el cariño que me han brindado y que me han dado su apoyo incondicional en el transcurso de mi vida.

A mi mamá por cuidarme y guiarme por el buen camino dándome consejos y siempre estando alentándome a lograr mis objetivos.

A mi papá que siempre me enseño a ser una buena persona, viendo por mí y por mis hermanos para que no faltara lo esencial en el hogar y que por sus consejos he logrado mis propósitos.

A mis hermanos Carlos y Nancy que han sido mi inspiración para llegar a ser un buen profesionista y que siempre han estado conmigo para seguir adelante.

Un agradecimiento especial a mi asesor y amigo Ricardo Marcelín, por sus consejos, oportunidades y conocimientos que me brindó para el desarrollo de este trabajo.

Gracias a mis sinodales por su dedicación y tiempo para revisar mi proyecto de investigación y que formaron parte importante en conseguir este logro de la terminación de mis estudios de posgrado.

Gracias a mis amigos y colegas del ARTe que me brindaron su amistad.

RESUMEN

El consenso, de manera genérica, consiste en un conjunto de procesos que registran, cada uno, algún valor de entrada y, basados en estos valores, deben coincidir en un valor de salida. Paxos es un algoritmo que resuelve el problema del consenso para el manejo consistente de registros duplicados en una red de procesos expuestos al riesgo de fallas de paro con una recuperación posterior. Se le utiliza ampliamente en la construcción de sistemas con fuertes requerimientos de disponibilidad. En este trabajo describimos el sistema de archivos Babel y la experiencia en la construcción de una implementación de Paxos, pensada para su aplicación en un sistema de gestión de metadatos para la recuperación de la información que se almacena al interior de Babel.

La transformación de una descripción formal en una aplicación eficiente, escalable y confiable es un proceso difícil que requiere abordar una serie de cuestiones prácticas y tomar decisiones de diseño cuidadosas. Existe un conjunto de condiciones que garantizan el funcionamiento del protocolo. Sin embargo, también pueden presentarse combinaciones de eventos de falla que cancelan estas condiciones de operación.

Nuestra contribución se basa en el uso de una implementación que acepta inyección de fallas, mediante las cuales fuimos capaces de reconocer estas combinaciones indeseables, así como también fuimos capaces de proponer los mecanismos que permiten restablecer las condiciones de base para la operación correcta de Paxos.

CONTENIDO

LISTA DE FIGURAS

LISTA DE TABLAS

CAPÍTULO

1

INTRODUCCIÓN

El sistema de archivos Babel, es un sistema de almacenamiento masivo de información con garantías de alta disponibilidad y escalabilidad. Se compone de un conjunto de máquinas con capacidades de almacenamiento y procesamiento conectadas mediante una red local. Los clientes de Babel perciben una sola máquina, denominada coordinador o proxy, que despacha las solicitudes de servicio (almacenamiento, búsqueda y recuperación de archivos) y administra los recursos.

Los archivos que se reciben desde los clientes de Babel se guardan de manera redundante, esto quiere decir que se crea un exceso en la información que codifica a los archivos y este exceso se guarda de forma distribuida entre los dispositivos de almacenamiento que componen al sistema. Se trata de una solución que puede articular un número masivo de dispositivos y presentarlos bajo una interfaz única. En la Figura 1.1 se presenta el contexto del funcionamiento del sistema Babel.

El beneficio inmediato del almacenamiento distribuido es que se logra la independencia entre la información y el medio en que se almacena. Visto de otra forma, los archivos que se guardan en un repositorio colectivo no dependen de un solo dispositivo para su recuperación. Si un documento estuviera guardado en una sola máquina, entonces la falla de ésta cancelaría su recuperación. En tanto, el exceso de información constituye una forma de respaldo que ofrece garantías de tolerancia a fallas y mejora la disponibilidad del sistema [1, 2].

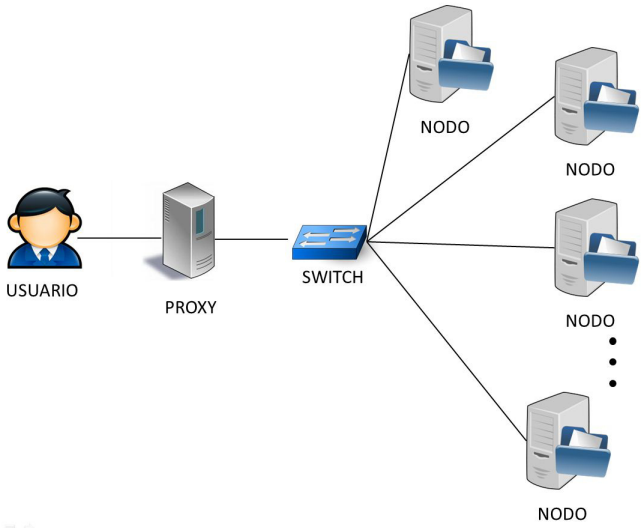


Figura 1.1: Funcionamiento del sistema Babel

1.1. Problemática

El proxy almacena los metadatos que hacen posible la recuperación de la información de los clientes que utilizan Babel. Actualmente se tiene un solo proxy, si éste fuera llevado al límite de sus capacidades por un exceso de peticiones podría crear un cuello de botella, así mismo seria el punto más débil del sistema si quedara fuera de servicio, entonces podría limitarse severamente la posibilidad de recuperar cualquier archivo almacenado al interior del sistema.

Para evitar cualquiera de estas contingencias se propuso implementar un conjunto redundante de proxies que puedan garantizar la consistencia de la información que gestionan, haciendo que nuestro sistema tenga alta disponibilidad.

Al contar con un conjunto redundante de proxies se solucionan los dos problemas mencionados anteriormente, sin embargo nos encontramos con un problema de consenso como se explica a continuación.

Imaginemos, por ejemplo, un usuario que es atendido por un proxy A, a través del cual almacena un archivo X. Los metadatos necesarios para recuperar X quedan registrados en A. Sin embargo, cuando el usuario regresa al sistema para recuperar su información, se le asigna un proxy B, para atender su nueva solicitud. Evidentemente, B debe disponer de una copia de los metadatos inicialmente registrados en A, de manera que pueda recuperar la información del usuario.

Lo anterior significa que es necesario garantizar que cualquier metadato registrado por un proxy quede registrado en todos los demás pares en servicio, aun bajo ciertas restricciones en la operación de los participantes. La condición anterior describe un problema del cómputo distribuido conocido como el problema de consenso.

La solución, si existe, debe exhibir tres propiedades: i) validez, ii) acuerdo, y iii) terminación. La primera indica que el valor de salida debe ser alguno de los valores inicialmente registrados. La segunda propiedad expresa el requisito de coincidencia. La tercera, requiere que la solución se alcance al cabo de un tiempo finito.

El conocido resultado de Fischer, Lynch y Patterson [3], demuestra que el consenso es imposible en un sistema totalmente asíncrono si, al menos, uno de los participantes puede experimentar una falla de paro. Los protocolos de consenso son la base para la construcción de la llamada máquina de estados, según lo sugiere Schneider [4]. El enfoque de máquina de estados es una técnica para construir sistemas distribuidos tolerantes a fallas.

Es un resultado bien conocido que el consenso es imposible [3], a menos que se ofrezca un conjunto mínimo de garantías sobre el tiempo, el orden de las comunicaciones o el procesamiento [5] [6].

En la Figura 1.2 se observa una instancia de Babel, con un conjunto redundante de proxies. Este es el escenario sobre el que se enfoca el proyecto que aquí se describe.

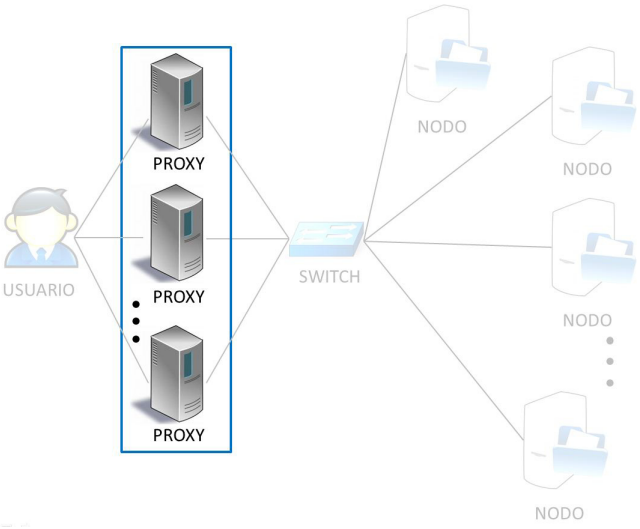


Figura 1.2: Lugar donde se produce el problema de la consistencia de los metadatos

1.2. Propuesta

Se implementó un mecanismo sobre el que pueda construirse la consistencia de los metadatos replicados en cada proxy. El objetivo de este trabajo es el desarrollo de un prototipo para garantizar la consistencia de las copias de una base de datos (BDD), que están a cargo de un conjunto redundante de servidores, usando para ello el conocido protocolo Paxos, propuesto por Lamport [7].

Puede entenderse como la construcción de un mecanismo de comunicación (o middleware), tolerante a fallas, a través del cual los servidores se comunican para garantizar que cualquier cambio en una instancia de la BDD queda registrado en todas las otras copias. Se sabe que, bajo ciertas condiciones de operación, Paxos garantiza la solución del consenso.

El valor de nuestra contribución radica en el hecho de usar la simulación de eventos discretos para reproducir diferentes escenarios de operación sobre los que puede desplegarse el protocolo Paxos. Gracias a esta metodología pudimos reconocer condiciones que en el trabajo original se asumen resueltas, o se dejan en manos del equipo a cargo de la implementación. Otra ventaja de nuestro enfoque de construcción es que planteamos un módulo a cargo del protocolo, que puede exportarse con facilidad hacia un entorno de producción.

1.3. Objetivos

A continuación se describen de manera puntual el objetivo general y los objetivos específicos que se persiguen en este trabajo.

1.3.1. General

- Diseñar y construir un middleware que garantice la consistencia de la información duplicada bajo escenarios de fallas de paro, con su recuperación.

1.3.2. Específicos

- Especificar el protocolo Paxos usando una herramienta formal que acepte un desarrollo incremental.
- Plantear las entidades de una arquitectura que implemente dichas operaciones.

- Diseñar un manejador de metadatos distribuidos y su relación con las entidades de la arquitectura resultante.
- Construir un prototipo del sistema.

1.4. Metodología

Se utilizó una plataforma de software que ofrece un ambiente para la simulación y análisis de algoritmos distribuidos. Con esta herramienta, un programador codifica su algoritmo en Python [8] con el fin de ligarlo a las bibliotecas de nuestro sistema. El diseño se basa en el concepto de máquina de estados (un recurso teórico de cómputo distribuido) para describir la interacción entre entidades autónomas.

1.4.1. Características del simulador

- Una plataforma de código abierto que permite separar por un lado, el algoritmo y, por otro, la gráfica que representa la red de comunicaciones. Con ello es posible simular el algoritmo sobre diferentes topologías, sin modificar el código de la simulación.
- Permite simular eventos aleatorios, dejando al programador en libertad para especificar cualquier función de distribución de probabilidad, ya sea para caracterizar el tiempo de un paso de procesamiento, el retardo de transmisión de un mensaje, o el tiempo en que se presenta una falla.
- Cuenta con un mecanismo de comunicación entre las entidades activas basado en el modelo de paso de mensajes. El usuario puede extender la definición de los mensajes, para establecer sus propias unidades de información.
- Permite simular la ejecución concurrente del mismo algoritmo, para estudiar, por ejemplo, situaciones de competencia.
- Permite simular, por ejemplo, situaciones de cooperación entre entidades (e.g. sincronizadores o elección).

El resto de este trabajo incluye las siguientes partes o secciones. En el capítulo 2 se explica el algoritmo Paxos, los supuestos de operación con los cuales trabaja, sus propiedades, el funcionamiento de una instancia del algoritmo y se describe una revisión de otros trabajos acerca de Paxos y su aplicación en los sistemas que se encuentran en producción. También presentamos una visión panorámica de otros mecanismos alternativos, usados para la solución del consenso.

En el capítulo 3 se describen varios escenarios de ejecución donde detectamos ciertas condiciones que imposibilitan resolver el problema del consenso y así como los mecanismos para solucionarlos. En el capítulo 4 sometimos nuestro prototipo a una serie de experimentos con diferentes parámetros de configuración para determinar el rendimiento y funcionalidad del módulo. Finalmente, en el capítulo 5 se hace una reflexión sobre la contribución y el aprendizaje que nos dejó este proyecto, así como unas conclusiones acerca del mismo.

CAPÍTULO

2

EL PROTOCOLO PAXOS

El protocolo Paxos fue publicado por primera vez en 1989, en formato de reporte técnico, y lleva el nombre de una isla griega en la que, según el autor, existió una civilización que desarrolló un mecanismo a través del cual, los miembros de su poder legislativo podían alcanzar acuerdos sobre la agenda de gobierno, sin caer en contradicciones, aun cuando los legisladores entraban y salían del recinto de asambleas. El tono de la presentación es un divertimento que se aleja del estilo tradicional con el que se presentan los artículos de computación.

Algunos investigadores consideraron que este tono retrasó su valoración por parte de la comunidad de los sistemas distribuidos. Fue reimpreso más tarde, con cambios menores, como un artículo de revista en 1998 [7]. Finalmente, el propio autor escribió una versión desprovista de toda alegoría, cuya lectura es relativamente más fácil y, en nuestra opinión, es un mejor punto de partida para embarcarse en su estudio [9].

La propuesta original de Paxos incluye el llamado protocolo múltiple o multi-Paxos, en el que se describe cómo puede extenderse el procedimiento básico para manejar de manera concurrente varias instancias del protocolo Paxos básico. A lo largo del tiempo, varios autores han hecho sus propias versiones modificando el algoritmo original para diferentes entornos. Con este hecho, Paxos se ha convertido en una familia de protocolos para la solución de consenso en una red de procesos susceptibles de experimentar ciertos tipo de falla.

La familia de protocolos Paxos incluye una gama de soluciones que combina el número de procesos participantes, el nivel de actividad individual de los mismos, los tipos de falla que estos pueden experimentar, entre otros parámetros de operación.

Aunque, como ya se dijo, no existe un protocolo de consenso determinista para garantizar el progreso en una red asíncrona, el algoritmo Paxos garantiza la seguridad y define las condiciones para alcanzar el consenso. Paxos se usa normalmente en situaciones que requieren durabilidad (por ejemplo, para replicar un archivo o una base de datos), donde el sistema permanece en estados estables por largos períodos. El protocolo intenta avanzar incluso durante períodos en los que un limitado número de procesos no responden.

2.1. Supuestos de operación

Para lograr el correcto funcionamiento del algoritmo Paxos se requiere el cumplimiento de los siguientes supuestos de operación. En muchos de estos, se trata de condiciones límite más allá de las cuales no se garantizan los resultados del protocolo.

Sobre los procesos

- Los procesos operan a una velocidad arbitraria.
- Los procesos pueden experimentar fallas de paro con su recuperación posterior.
- Los procesos disponen de un almacenamiento estable que les permite reconocer su historia y reintegrarse al protocolo después de recuperarse de su fallo.
- Los procesos no confabulan, mienten o hacen cualquier otro intento de desestabilizar el protocolo, es decir, las fallas bizantinas no se producen.

Sobre las comunicaciones

- Los procesos envían mensajes a cualquier otro proceso.
- Los mensajes se envían de forma asíncrona y pueden tomar un tiempo arbitrario de entrega.
- Los mensajes enviados no se pierden.
- Los mensajes se entregan sin daños.

2.2. Roles

Paxos describe las acciones de los procesos por sus roles en el protocolo. Estos son los siguientes: el cliente, el aceptante, el aprendiz y el líder. En las implementaciones típicas, un mismo proceso puede desempeñar uno o varios de estos roles sin detrimento del protocolo.

- **Cliente:** El cliente envía una petición al líder en el sistema distribuido para alcanzar el consenso sobre un valor en el que está interesado y espera una respuesta. Por ejemplo, una solicitud para modificar el estado de la entrada en una tabla de una BDD o para recuperar el valor previamente acordado.
- **Aceptantes:** Los aceptantes actúan como la *memoria* con tolerancia a fallas del protocolo. Los aceptantes se reúnen en grupos llamados cuorums. Un cuórum se define como un subconjunto de los aceptantes, que expresa las característica de seguridad de Paxos. Ello significa que en un cuórum existe al menos algún proceso sobreviviente que conserva el conocimiento de los resultados previos, esto es porque ha participado en anteriores rondas del protocolo. Típicamente, un cuórum es cualquier mayoría simple de aceptantes, es decir, la mitad del total de procesos más uno. Por lo anterior se prefiere que el número de participantes sea impar.
- **Líder:** Un líder aboga por la petición de un cliente tratando de convencer a los aceptantes de llegar a un acuerdo sobre un valor y actúa como un coordinador para evitar que el sistema se estanque en caso de conflictos. Bajo condiciones indeseables, varios procesos pueden considerar que son líderes, pero el protocolo sólo garantiza el progreso si finalmente se elige exactamente a uno. Si, de manera simultánea, más de un proceso considera que es el líder en funciones, entonces puede atascarse el protocolo ya que se entra en una situación de competencia en la que se proponen continuamente actualizaciones conflictivas.
- **Aprendices:** Los aprendices actúan como factor de replicación para el protocolo. Una vez que una solicitud del cliente ha sido acordada por los aceptantes, el aprendiz puede emprender una acción en beneficio del cliente inicial, es decir, cumplir la solicitud y enviar una respuesta al cliente.

2.3. Propiedades del protocolo Paxos básico

Paxos define tres propiedades y garantiza que siempre se llevan a cabo, independientemente del patrón de fallas:

- **No trivialidad.** Solo los valores propuestos se pueden aprender (es decir, registrarlo en cada proceso).
- **Seguridad (Safety).** A lo mas un valor se puede aprender.
- **Vitalidad (Liveness).** Si un líder vive lo suficiente, entonces es capaz de sacar adelante la solución de consenso.

2.4. Despliegue típico del protocolo

Este protocolo es el más sencillo de la familia Paxos. Cada instancia del protocolo Paxos básico decide sobre un único valor de salida. El protocolo procede en varias rondas. Una ronda exitosa tiene dos fases. Un líder no debe iniciar Paxos si no se puede comunicar con al menos un cuórum de los aceptantes. A continuación se describen las dos fases por las que transita el protocolo Paxos.

- **Fase 1_a: Prepara.** El líder crea una propuesta identificada con un número de ronda N. Este debe ser mayor que cualquier número de propuesta del que tenga conocimiento. Después, envía un mensaje *PREPARA* a todos los participantes, conteniendo el número de ronda.
- **Fase 1_b: Promesa.** Si el número de la propuesta N es mayor que cualquier número recibido previamente, entonces el aceptante envía un mensaje *PROMESA* al líder y con ello se compromete a ignorar todas las propuestas que tienen un número menor que N. Si el aceptante aceptó una propuesta en algún momento en el pasado, debe incluir el número de la propuesta anterior y el valor anterior en su respuesta al líder. En caso contrario, si el número de propuesta es menor que otro con el que ya se comprometió, el aceptante puede ignorar la propuesta recibida.

  No tiene que responder en este caso para que funcione Paxos. Sin embargo, en bien de la economía de mensajes, el envío de una respuesta negativa (Nack) le diría al líder que puede poner fin a su intento de crear un consenso con la propuesta N.

- **Fase 2_a: Aceptar.** Si un líder recibe suficientes promesas de un cuórum de los aceptantes, es necesario que asocie un valor con su propuesta. Si alguno de los aceptantes había aceptado previamente cualquier propuesta, entonces ellos han enviado sus valores al líder, que ahora debe igualar el valor de su propuesta con el valor más reciente reportado por algún aceptante.

  Es decir, el valor asociado con el mayor número de propuesta reportado en un mensaje *PROMESA*, recién recibido. Si ninguno de los aceptantes había aceptado una propuesta hasta este punto, entonces el líder puede elegir cualquier valor para su propuesta. El líder envía un mensaje *ACEPTAR* a un cuórum de aceptantes con el valor elegido para su propuesta.

- **Fase 2_b: Aceptado.** Si un aceptante recibe un mensaje *ACEPTAR* de la propuesta N, debe aceptarlo si y sólo si aún no ha prometido tener en cuenta sólo las propuestas que tengan un identificador mayor que N, es decir si aún no ha respondido a ningún mensaje *PREPARA* (enviar un mensaje *PROMESA*). En este caso, se debe registrar el valor correspondiente y enviar un mensaje *ACEPTADO* para el líder y cada aprendiz. Si no, se puede ignorar el mensaje *ACEPTAR*.

Las rondas fallan cuando varios líderes envían mensajes contradictorios *PREPARA*, o cuando el líder no recibe respuestas de un cuórum (mensajes *PROMESA* o *ACEPTADO*). En estos casos, otra ronda se debe iniciar con un número de propuesta superior. Obsérverse que cuando los aceptantes aceptan una propuesta, también reconocen el liderazgo del líder. Por lo tanto, Paxos se puede utilizar para seleccionar un líder en un clúster de nodos.

Si el líder es relativamente estable, la fase 1 se convierte en innecesaria. Por lo tanto, es posible omitir la fase 1 para futuras instancias del protocolo con el mismo líder.

2.5. Escenarios del protocolo Paxos

A continuación se muestran los diferentes escenarios que puede presentar el protocolo Paxos.

2.5.1. Paxos Básico

Aquí se muestra una representación gráfica del protocolo Paxos básico. Un cliente realiza una solicitud al líder, el líder aboga por la solicitud y envía un mensaje *PREPARA* a un cuórum de aceptantes para establecer el consenso sobre el valor propuesto. Hay que tener en cuenta que los valores devueltos en el mensaje *PROMESA* son nulos cuando la propuesta es hecha por primera vez, ya que ningún aceptante ha aceptado un valor antes de esta ronda.

Los aceptantes envían un mensaje *PROMESA* al líder indicando que están de acuerdo con el valor propuesto, a continuación el líder al recibir una mayoría de mensajes *PROMESA* de un cuórum, envía un mensaje *ACEPTAR* al cuórum de aceptantes para que registren el valor correspondiente en esa instancia del algoritmo.

Los aceptantes reciben este mensaje, registran el valor acordado y notifican al líder y a cada aprendiz mediante el envió de un mensaje *ACEPTADO* que indica que ya se realizó el registro del valor. En la figura 2.1 se muestra el flujo de mensajes del protocolo Paxos básico.

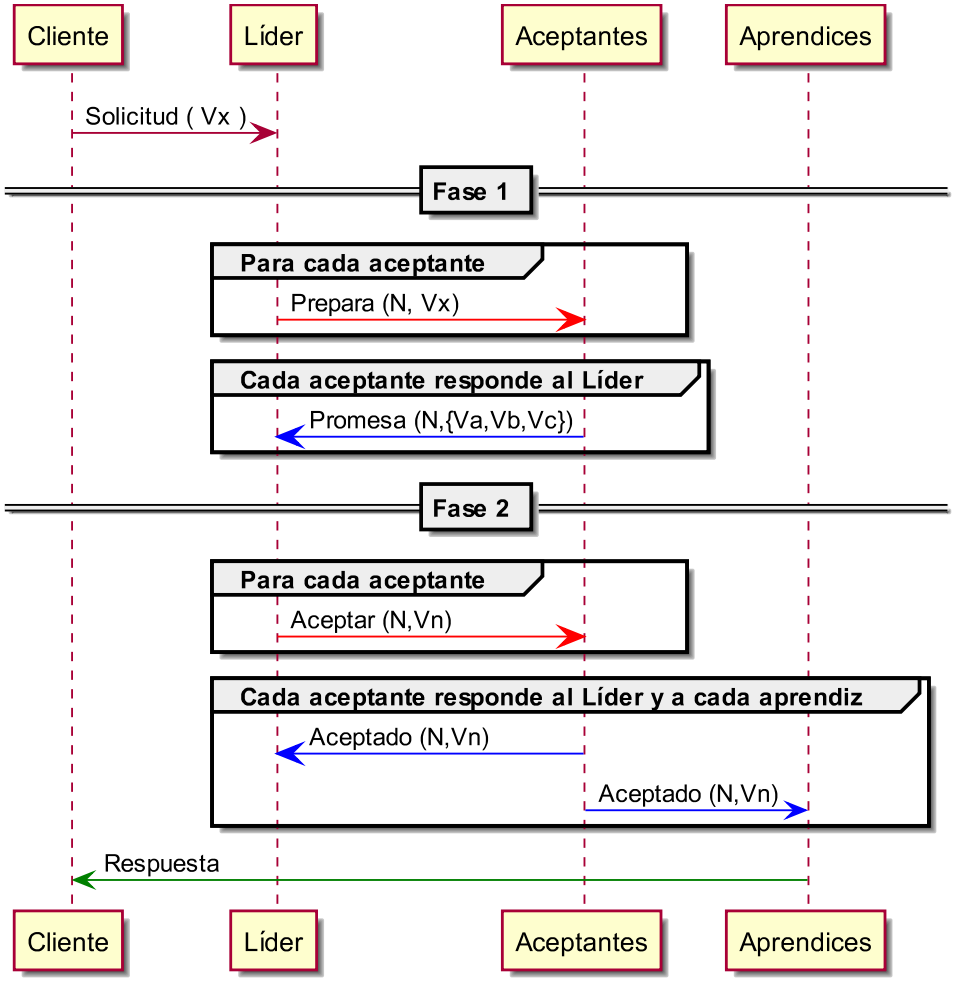


Figura 2.1: Paxos Básico. Primera ronda es exitosa

2.5.2. Casos de error en Paxos Básico

Los casos de error más simples son el fallo de un aprendiz redundante o el fallo de un aceptante cuando un cuórum de los aceptantes se mantiene vivo. En estos casos, el protocolo no requiere recuperación. No son necesarios rondas o mensajes adicionales, como se muestran en las figuras 2.2 y 2.3.

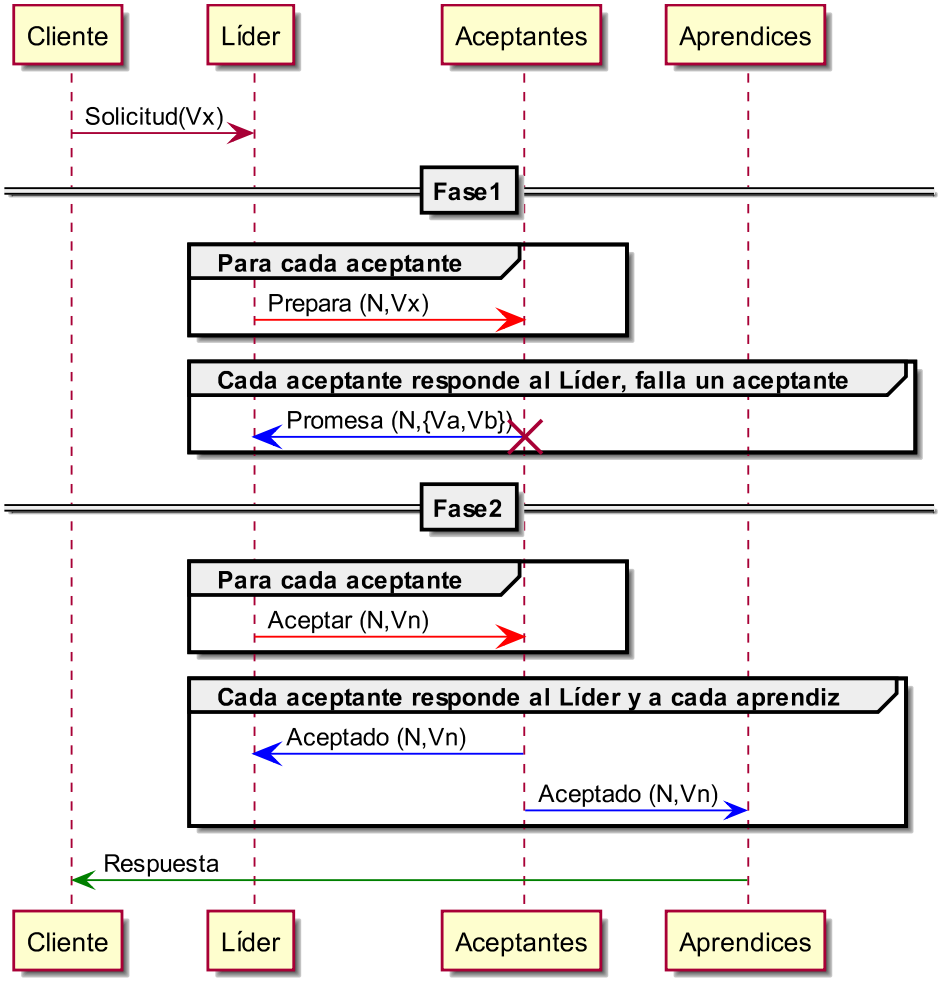


Figura 2.2: Paxos Básico, falla de un aceptante

El siguiente caso de falla es cuando un líder falla después de proponer un valor, pero se alcanza antes el acuerdo. A continuación mediante un algoritmo de elección de líder se establece un nuevo líder e inicia una nueva ronda del protocolo, un ejemplo de un flujo de mensajes se muestra en la figura 2.4.

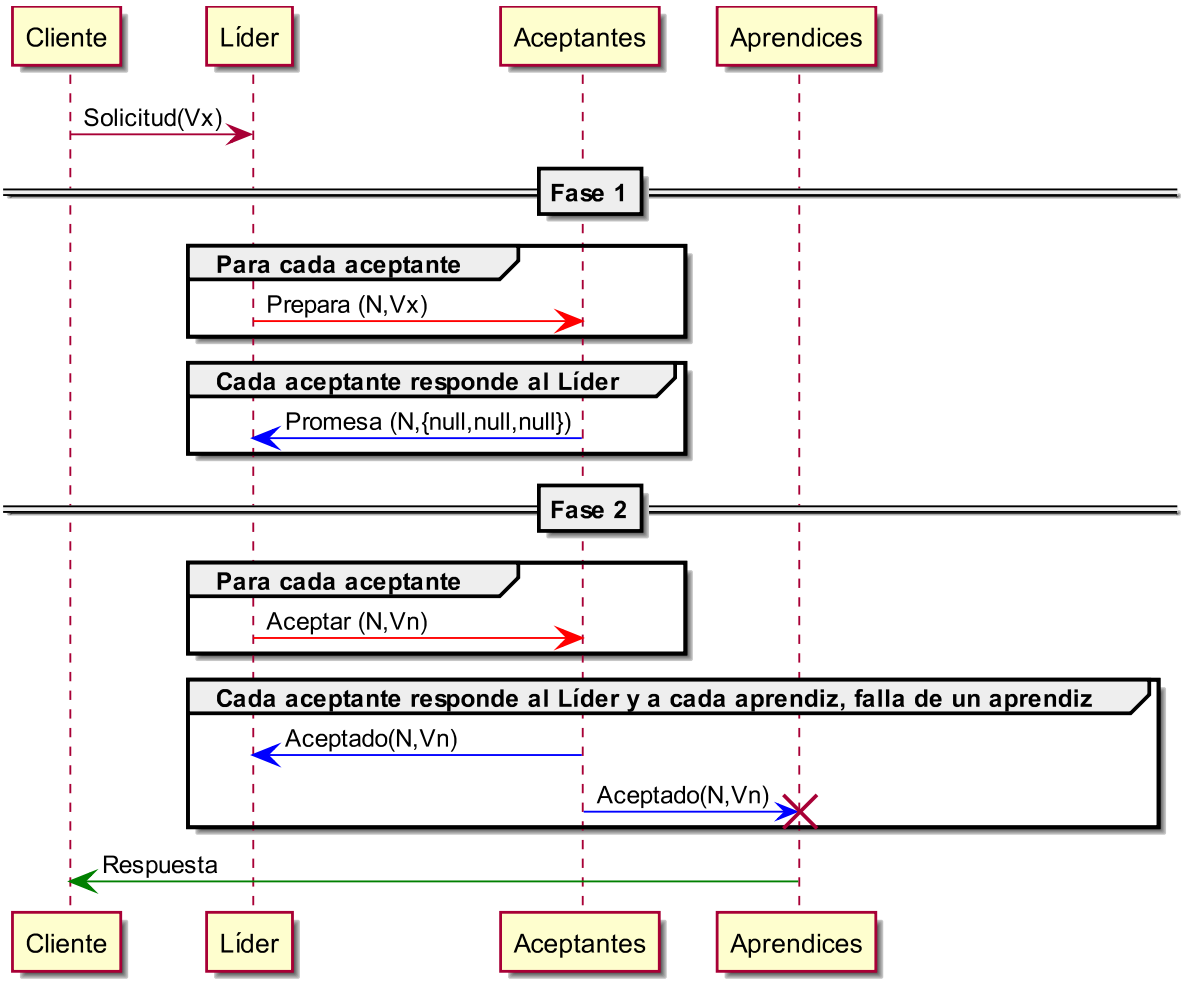


Figura 2.3: Paxos Básico, falla de un aprendiz redundante

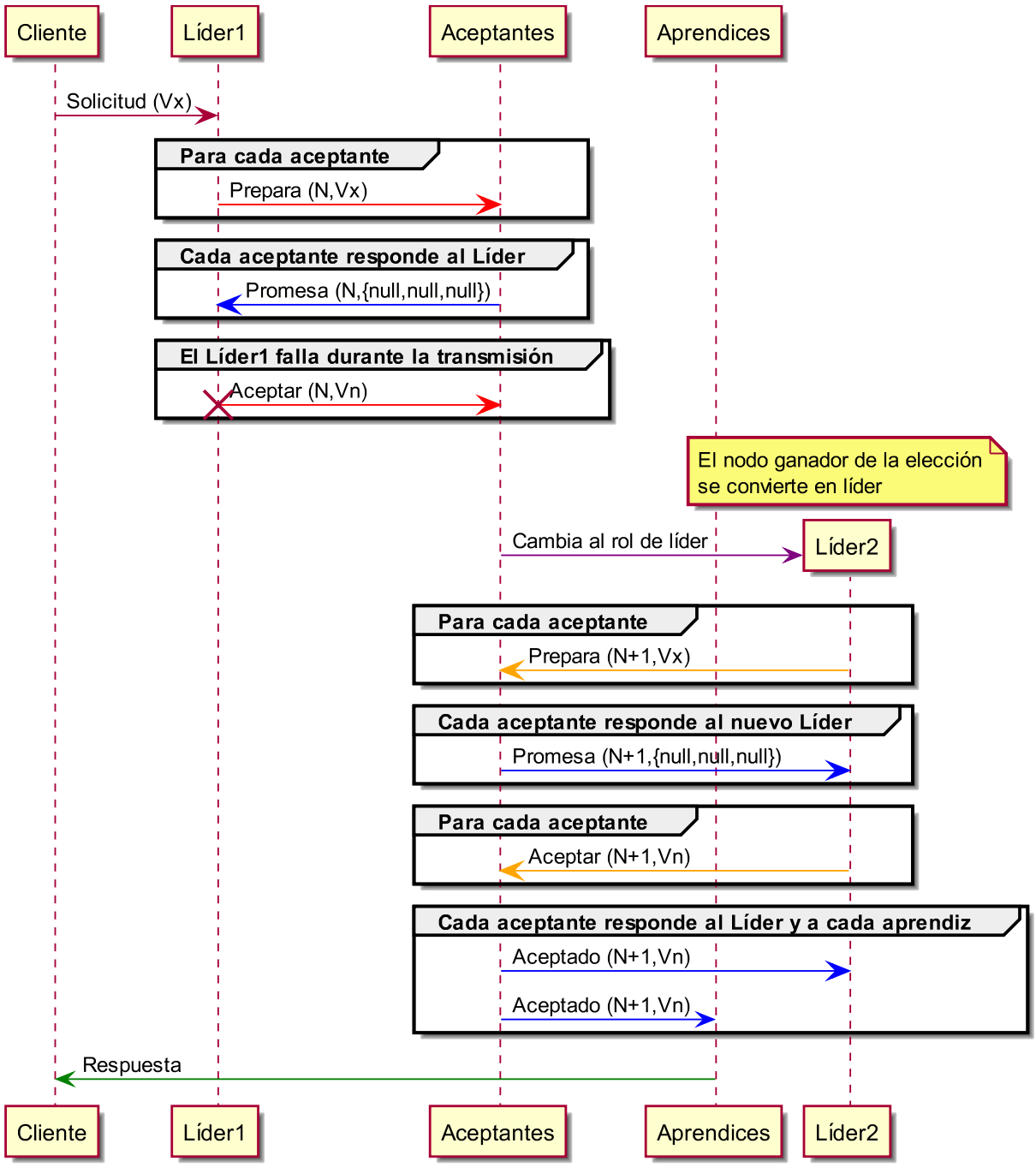


Figura 2.4: Paxos Básico, falla del líder

El caso más complejo es cuando varios participantes creen que son líderes. Por ejemplo, el actual líder puede fallar y luego recuperarse, pero los otros procesos ya han designado a un nuevo líder. El anterior líder se recupera y no ha aprendido que existe un nuevo líder, esto provoca que los intentos de iniciar una ronda entren en conflicto con el actual líder. El flujo de mensajes de este caso se muestran en las figuras 2.5 y 2.6.

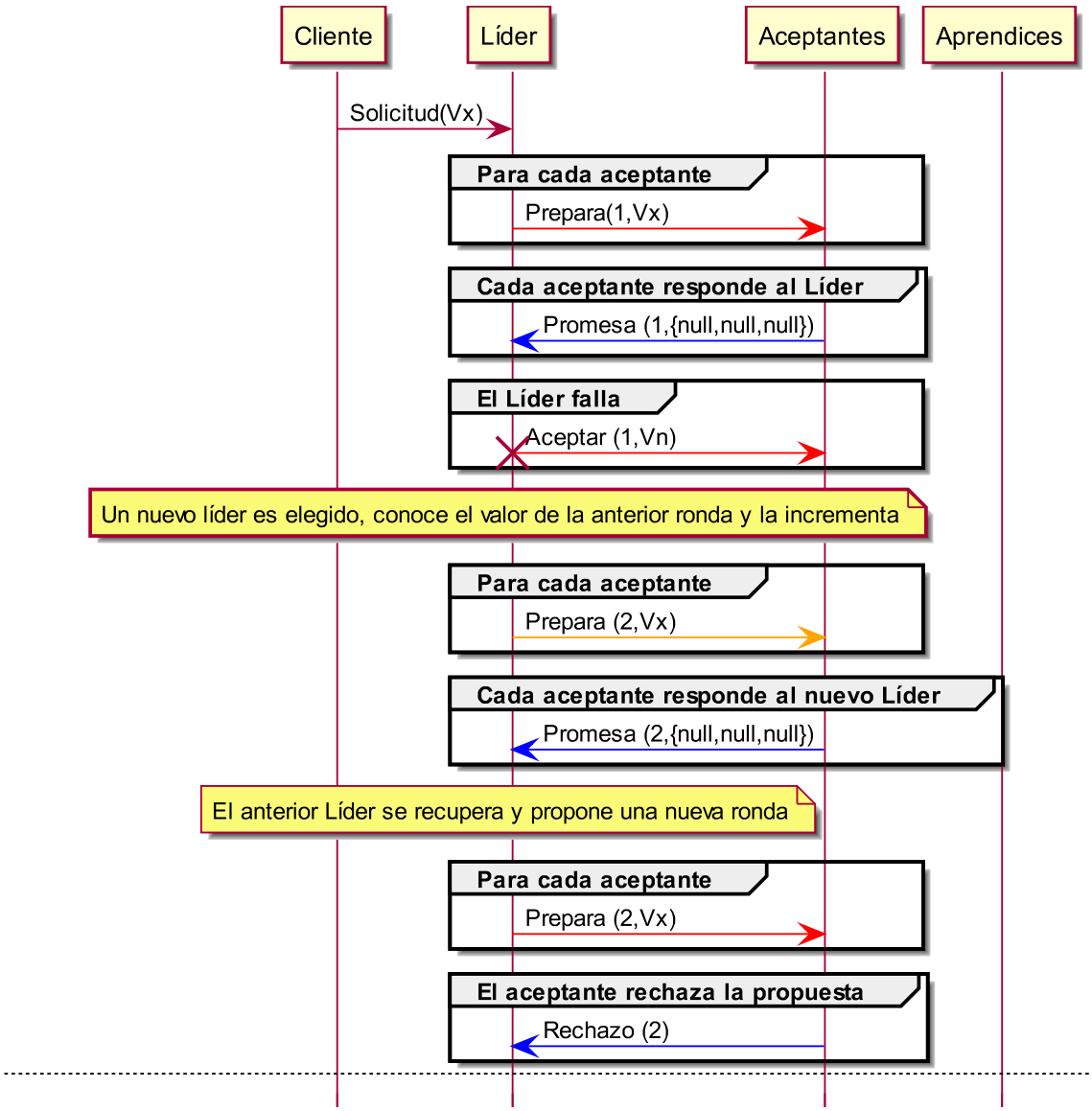


Figura 2.5: Paxos Básico, líderes en duelo

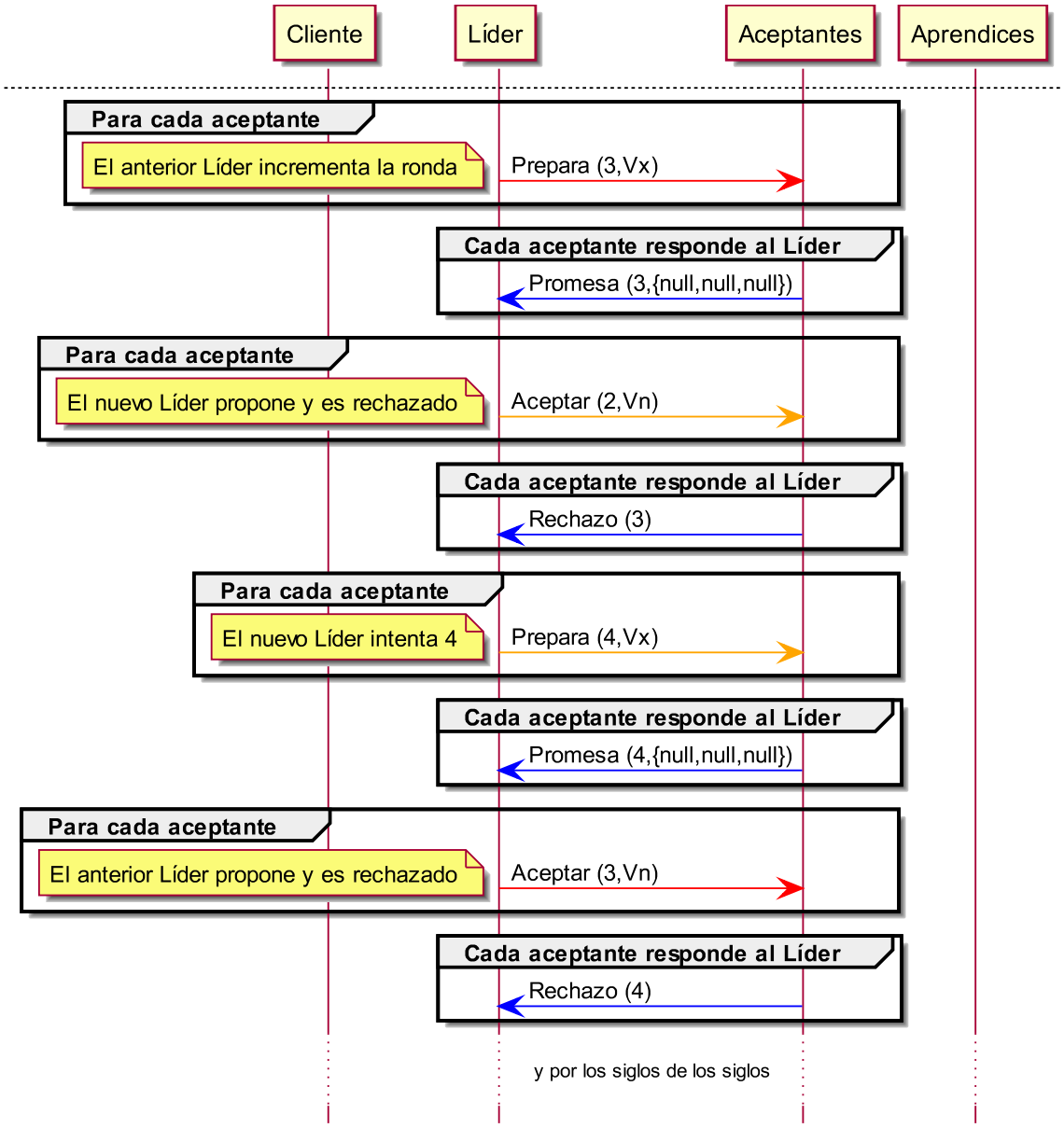


Figura 2.6: Paxos Básico, líderes en duelo, Cont.

2.6. Multipaxos

Una implementación típica de Paxos requiere un flujo continuo de valores acordados que actúan como comandos en una máquina de estados distribuidos. Si cada comando es el resultado de una única instancia del protocolo Paxos básico, daría como resultado una cantidad significativa de sobrecarga. Como se mencionó anteriormente, si el líder es relativamente estable, la fase 1 se convierte en innecesaria.

Por lo tanto, es posible omitir la fase 1 para futuras instancias del protocolo con el mismo líder. Para lograr esto, el número de instancia I es incluido junto con cada valor sobre el que se busca alcanzar un acuerdo. Las figuras 2.7 y 2.8 muestran el flujo de mensajes de Multipaxos.

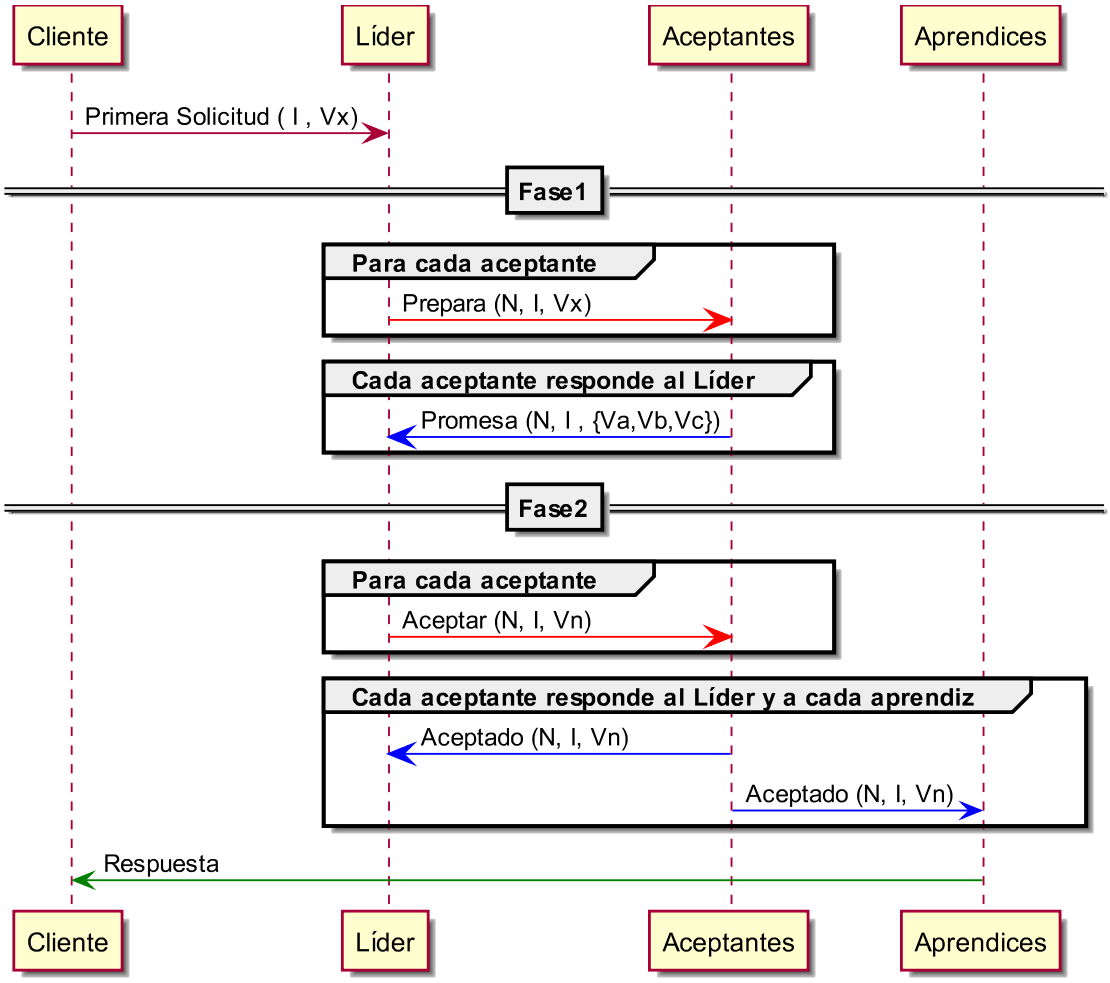


Figura 2.7: Multipaxos. Primera instancia con el nuevo líder

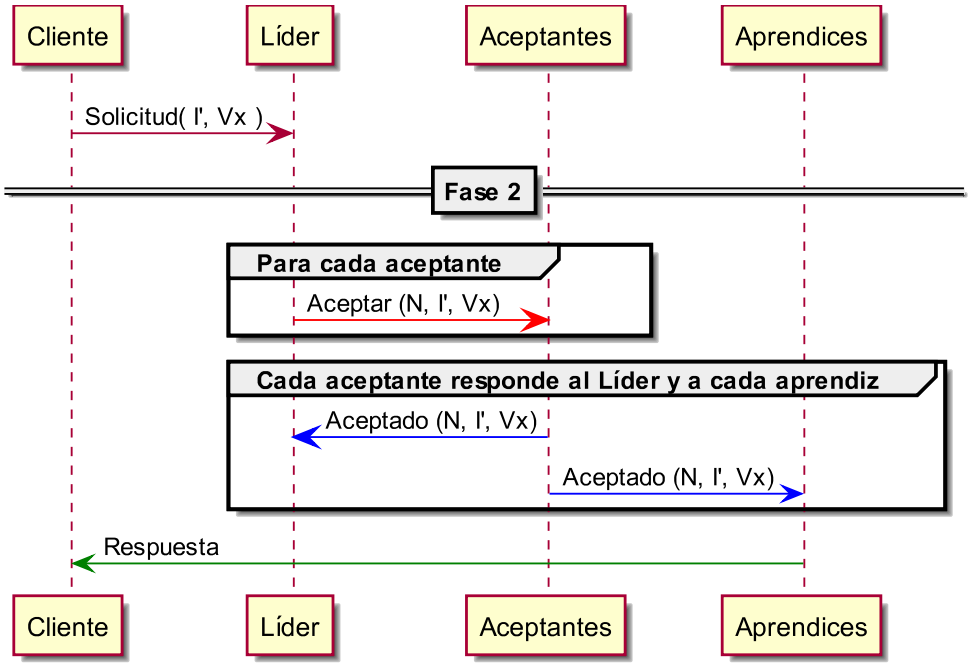


Figura 2.8: Multipaxos. Instancias posteriores con el mismo líder

2.7. Trabajo relacionado con Paxos

Desde su presentación original, se han publicado varios trabajos que han profundizado la comprensión del algoritmo Paxos, De Prisco, Lampson y Lynch [10] especifican y demuestran la exactitud del protocolo utilizando un autómata general de reloj sincronizado para modelar los procesos y canales en el sistema. Su especificación hace uso de un detector de fallas para la elección de líder y proporcionan un análisis teórico del comportamiento de Paxos.

Dutta, Guerraoui y Boichat [11] utilizan dos abstracciones principales para modularizar la funcionalidad del protocolo Paxos: una débil abstracción de elección de líder (ejecutado por un detector de fallas) y una abstracción de registro por turnos. También muestran cuatro variantes de Paxos que hacen diferentes supuestos y por lo tanto, son útiles en diferentes contextos. Lampson [12] presenta una versión abstracta de Paxos y luego muestra cómo las diferentes versiones de Paxos (es decir, Paxos Bizantino [13, 14] y Disk Paxos [15]) se pueden derivar de la versión abstracta.

Li, et al. [16] especifican Paxos utilizando una abstracción de registro. El protocolo explica cómo se llevan a cabo las operaciones de lectura y escritura en el registro de Paxos. Un trabajo desarrollado por Chandra, et al. [17] describe la experiencia de los autores en el uso de Paxos para construir una base de datos con tolerancia a fallas. Describen los problemas de los sistemas conexos con los que se encontraron, muchos de los cuales no se abordaron en previas especificaciones de Paxos.

El equipo que diseñó el Google File System (GFS) [18] desarrolló una implementación propietaria denominada Chubby [19]. Chubby es un sistema tolerante a fallas que proporciona un mecanismo distribuido para el almacenamiento de llaves. Típicamente, hay una instancia Chubby por cada centro de datos. Varios sistemas de Google, como el GFS y Bigtable [20], utilizan Chubby para la coordinación distribuida y para almacenar sus metadatos.

Petal [21] fue uno de los primeros sistemas de almacenamiento que utilizo Paxos para replicar metadatos. Más tarde Boxwood [22] propuso Paxos como herramienta de propósito general para mantener el estado global, así como un componenete subyacente de un servicio de bloques. Recientemente, Paxos se ha utilizado para la gestión de los metadatos en sistemas tales como Niobe [23].

Amir, et al. [24] especifican completamente el algoritmo Paxos en un lenguaje para la construcción de sistemas de tal manera que uno puede ponerlo en práctica, sin embargo, comentan que una aplicación práctica de Paxos aborda muchos problemas reales por los protocolos de duplicación, y para solucionar estos problemas son necesarios mecanismos adicionales que no son especificados en el algoritmo original.

En [25] presentan un esquema de gestión de metadatos basado en HDFS [26] para un entorno de computación en la nube. Para mantener la fiabilidad, los metadatos se replican en diferentes nodos, y adoptan el algoritmo Paxos para la replicación consistente de los metadatos.

En un reciente trabajo [27], muestran un enfoque basado en Paxos para la construcción de un servicio de gestión de metadatos en sistemas de archivos distribuidos, alcanzando una alta disponibilidad sin incurrir en una penalización de rendimiento.

En [28], los autores muestran una implementación propia de Paxos denominada RS-Paxos (Reed Solomon Paxos), en la cual combinan la codificación de borrado [29, 30], que es una técnica muy eficaz y común para reducir el costo de almacenamiento y de comunicaciones, en la replicación de datos. Explican que mediante la redefinicion del número de cuórum en cada fase Paxos, RS-Paxos reduce en gran medida los costos de operaciones de E/S en disco y en la red. Sin embargo un efecto secundario en RS-Paxos es que tolera menos fallos que en el protocolo original Paxos en una instancia.

Viewstamped Replication (VSR) [31,32] es un protocolo de replicación originalmente dirigido a la replicación de participantes en un protocolo de confirmación en dos fases (2PC) [33]. VSR fue utilizado en la implementación de Harp File System [34].

Zab (ZooKeeper Atomic Broadcast) [35] es un protocolo de replicación utilizado para el servicio de configuración de ZooKeeper [36]. ZooKeeper fue diseñado en Yahoo! y ahora es un producto de código abierto distribuido por Apache.

En [37] muestran las características de diseño que comparten los protocolos Paxos, VSR y Zab así como consideran el impacto del rendimiento de cada uno de estos protocolos. Los autores llegan a la conclusión que los servicios de cálculo intensivo están mejor con una estrategia de replicación pasiva [38], tal como se utiliza en VSR y Zab (siempre que las actualizaciones de estado son de un tamaño razonable). Para lograr un rendimiento predecible de bajo retardo, para operaciones cortas tanto durante la ejecución de caso normal y la recuperación, la mejor opción es una estrategia de replicación activa [4, 39] sin mayorías designadas, como se utiliza en Paxos.

En [40], construyeron un prototipo derivado de Paxos básico, denominado Packed-MultiPaxos, el cual consiste en el empaquetamiento de varias solicitudes de Paxos básico en una sola transmisión de red, es decir, en un paso de mensaje. Los autores definen los mecanismos de coordinación para las solicitudes de lectura y escritura de los metadatos. Evalúan su prototipo Packed-MultiPaxos con el protocolo original Multi-paxos [7] y muestran cómo su implementación reduce la latencia de los mensajes intercambiados entre los nodos ofreciendo un buen rendimiento de acuerdo a la carga de solicitudes del cliente.

Recientemente, se publicó un nuevo algoritmo de consenso llamado Raft [41], que es similar en muchos aspectos a Paxos. Sin embargo, Paxos ha dominado la discusión de los algoritmos de consenso durante la última década y la mayoría de las implementaciones de consenso se basan en Paxos. Los autores de Raft hacen énfasis en que Paxos es bastante difícil de comprender, y es por eso que definieron un algoritmo de consenso para los sistemas prácticos describiéndolo de una manera que es mucho más fácil de aprender que Paxos. Por otra parte, se sabe que Paxos es óptimo en el número de mensajes que intercambia, durante una condición de trabajo estable [42].

CAPÍTULO

3

ESCENARIOS CRÍTICOS

Basados en multi-Paxos construimos una descripción usando un autómata de estados finitos, que se muestra en el apéndice A de este trabajo, con el que programamos una versión escrita en el lenguaje Python. Luego, como parte del proceso de desarrollo, decidimos someter la implementación a una serie de pruebas de ingeniería. Considerando los alcances y los costos de estas pruebas preferimos "incrustar" nuestro código dentro de un simulador de eventos discretos.

Con ello conseguimos un control más fino sobre las diferentes condiciones de prueba, tales como el número de participantes, las tasas de falla y recuperación, entre otros factores. A partir de nuestra agenda de experimentos pudimos descubrir que nuestra implementación cumplía cabalmente con las especificaciones del protocolo, bajo condiciones estables. También fuimos capaces de detectar una serie de condiciones de inestabilidad en las que se pierde la garantía de progreso del protocolo.

El trabajo original no menciona las contingencias que pueden llevar a estos casos, y mucho menos la manera como pueden resolverse. Consideramos que la aportación de nuestro trabajo radica en el hecho de proponer los mecanismos para detectar las situaciones de inestabilidad que reconocimos, así como los procedimientos para llevar de nuevo al sistema hasta un estado donde se garantice el progreso.

En este capítulo se describen los problemas encontrados al someter a nuestro prototipo a una agenda de pruebas asumiendo tasas aleatorias de paro y recuperación, sobre cada uno de los procesos participantes. Reconocemos que la clave para el funcionamiento correcto de Paxos es que siempre exista exactamente un proceso distinguido o líder, sobre el que recae la responsabilidad de administrar las etapas del protocolo, para cada instancia del consenso.

Desde la perspectiva de nuestra implementación, identificamos los siguientes momentos críticos del protocolo:

- **Elección del líder:** un nuevo líder debe ser elegido al iniciar las operaciones o cuando se presume la ausencia del líder conocido.
- **Duplicación de funciones:** si existiera más de un líder, no se puede avanzar en el protocolo, un nuevo proceso de "desempate" debe arrancarse para superar esta condición.
- **Asignación del número de instancia:** cada instancia del consenso debe recibir un número diferente que la identifica de forma única. Debe existir un mecanismo para repartir los posibles números de instancia entre los procesos que fungirán como clientes. Evidentemente, el conjunto de números gestionados por un cliente debe ser ajeno al conjunto gestionado por cualquier otro cliente.
- **Consistencia y actualización de los registros:** todos los participantes deben tener las mismas entradas en sus registros, o un valor nulo, en caso de ignorar el acuerdo alcanzado para una instancia. En esta última circunstancia deberá actualizarse invocando los servicios del líder.
- **Solicitudes pendientes:** el líder en funciones puede experimentar una falla durante una instancia del protocolo Paxos sin posibilidad de llegar a un consenso sobre la solicitud que recibe.
- **Falta de cuórum:** El líder arranca una instancia del protocolo Paxos, sin embargo no recibe respuestas suficientes para asumir la existencia de un cuórum, imposibilitando con ello el avance del consenso.

Para los fines de nuestro estudio, asumimos que cada proceso a cargo del protocolo se encuentra asignado en un nodo diferente de la red de comunicaciones subyacente. Por ello, en lo sucesivo usaremos como sinónimos los términos "nodo" y "proceso".

3.1. Elección de líder

La elección de líder pasa por dos estados: *postulación* y *decisión*. Cada nodo se identifica por un número entero único. Para empezar, cada nodo implicado en la elección cambia al estado

postulación. A continuación, vota por sí mismo y, de forma concurrente, envía un mensaje *CANDIDATO*, donde incluye su propio identificador, a cada uno de los demás nodos del conjunto.

Un candidato gana una elección y se convierte en líder si tiene el mayor identificador dentro del grupo que participa en la elección. La primera etapa se cierra al cabo de un periodo de tiempo controlado por un temporizador llamado $Timer_{Cand}$. Si dentro de este plazo, un nodo recibiera noticias de la existencia de un candidato con un identificador mayor que el que tiene registrado, entonces actualiza este valor y lo reexpide a los demás participantes.

Al terminar el tiempo de espera, los nodos cambian al estado *decisión*. Se considera ganador al dueño del mayor identificador intercambiado. Sólo el ganador envía un mensaje latido a todos los demás nodos para establecer su autoridad. El resto, espera el latido del nodo que tienen registrado, para confirmarlo como el nuevo líder en funciones. En la figura 3.1 se muestra la elección de un líder sin contratiempos.

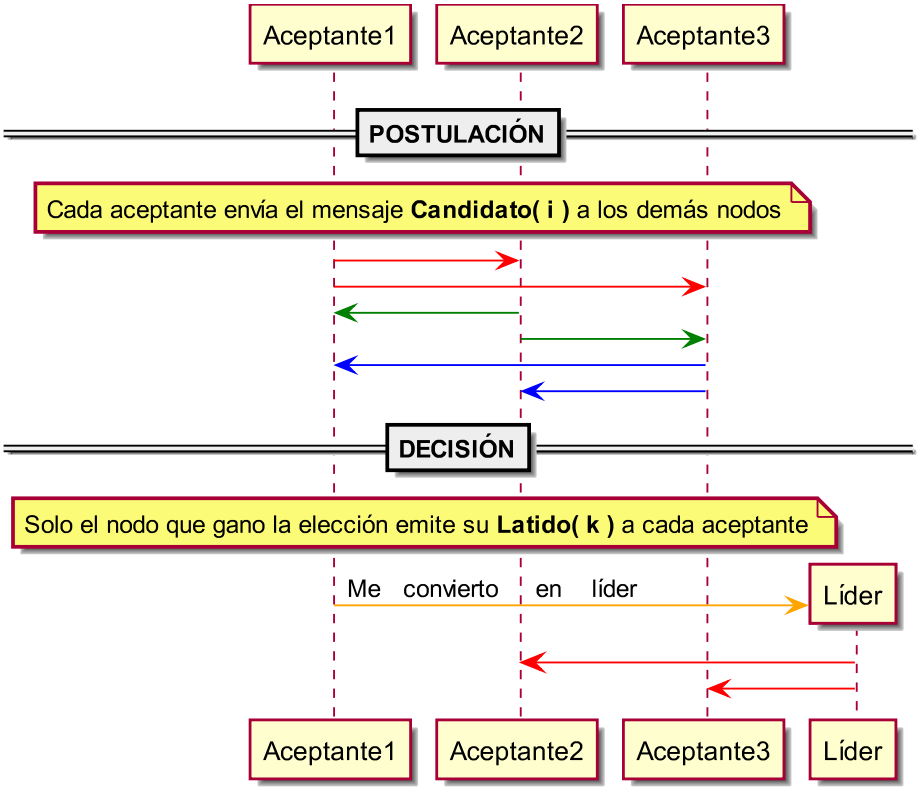


Figura 3.1: Procedimiento normal de la elección de un líder

Cuando el sistema inicia sus operaciones por primera vez, los nodos disparan la elección de líder. Una vez electo, el líder envía periódicamente latidos (heartbeats) a todos los aceptantes, con el fin de confirmar su disponibilidad. En la figura 3.2 se muestra este mecanismo.

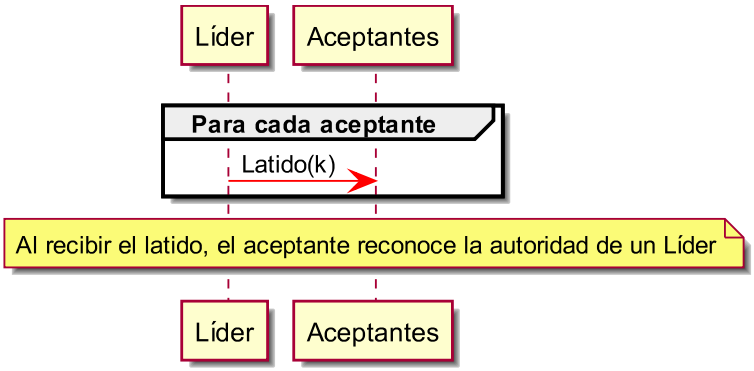


Figura 3.2: El líder envía periódicamente latidos a los aceptantes

El latido del líder incluye el tiempo cuando fue emitido y el tiempo en que enviará su próximo latido, los aceptantes al recibir esta señal, guardan esta información en sus registros. Si un aceptante no recibe el latido en el tiempo establecido, inicia el mecanismo para verificar si existe un líder.

Cada aceptante inicia su propio $Timer_{Latido}$ y envía un mensaje *LÍDER* a todos los demás nodos, sin embargo solo el líder en funciones responderá con el envío de su latido, con el fin de mantener su autoridad. Este mecanismo se repite cuando el aceptante no ha recibido la señal del próximo latido que espera dentro del período $Timer_{Latido}$. En la figura 3.3 se muestra el procedimiento descrito anteriormente.

Después de N intentos, si el aceptante no ha recibido el latido del líder, se asume que el líder en funciones falló y se arranca una nueva elección con los nodos que se encuentran activos. En la figura 3.4 se muestra cuando inicia el protocolo de elección para establecer un nuevo líder.

A lo largo del proceso de elección pueden presentarse alguna de las siguientes situaciones: (a) que expire el plazo para establecer al nuevo líder, o (b) que un nodo caído se recupere en el curso de la elección.

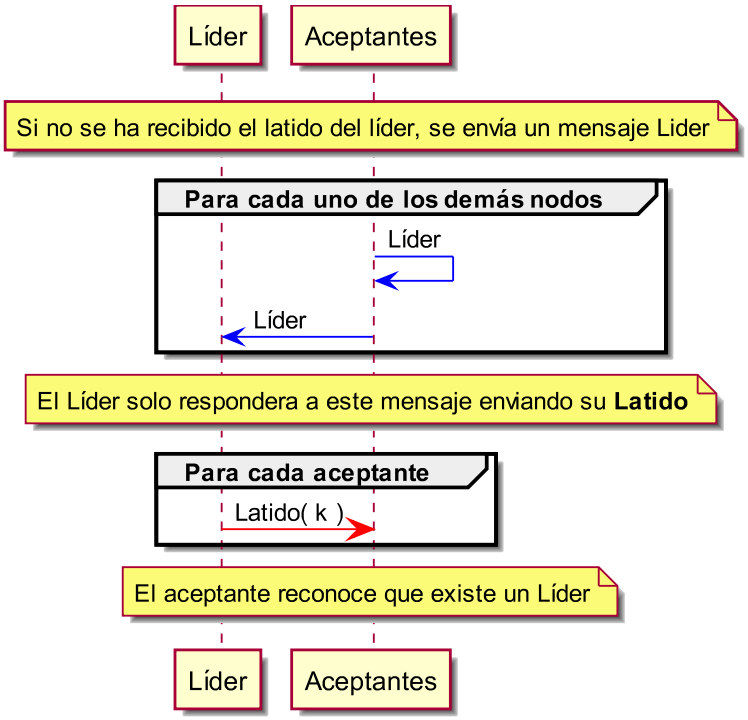


Figura 3.3: Mecanismo de verificación de la existencia de un líder

El primer escenario se produce cuando cae en falla el nodo que sería ganador. Para contrarrestar este problema, se añade un tiempo de espera adicional llamado $Timer_{Elect}$ que arranca en el momento que se cambia al estado *decisión*. Este mecanismo nos garantiza que los nodos tendrán un tiempo de espera para recibir el latido del nuevo líder, si expira este tiempo, se asume que el nodo electo falló y a continuación se realiza una nueva elección de líder con los nodos que se encuentran activos. En la figura 3.5 se muestra el mecanismo de los temporizadores $Timer_{Cand}$ y $Timer_{Elect}$.

El segundo escenario se presenta cuando un nodo ausente se recupera durante alguna etapa de la elección. Si regresa en la primera etapa, los nodos le comunican el estado del proceso y el identificador del mejor candidato del que se tiene noticia. Cuando el nodo recibe el mensaje, cambia su estado a *postulación*, emite su propio voto y continúa con la operación normal de elección. Si el nodo que regresa fuera el anterior líder, los nodos que participan saben que deben ignorar el latido que pudieran recibir de su parte.

Por otro lado, si el nodo se recupera en la segunda etapa, los nodos activos le comunican la identidad del nodo electo durante el transcurso del $Timer_{Elect}$ para que actualice su estado y espere el latido del nuevo líder. En las figuras 3.6 y 3.7 se muestran los escenarios descritos con un anterior líder y un aceptante respectivamente.

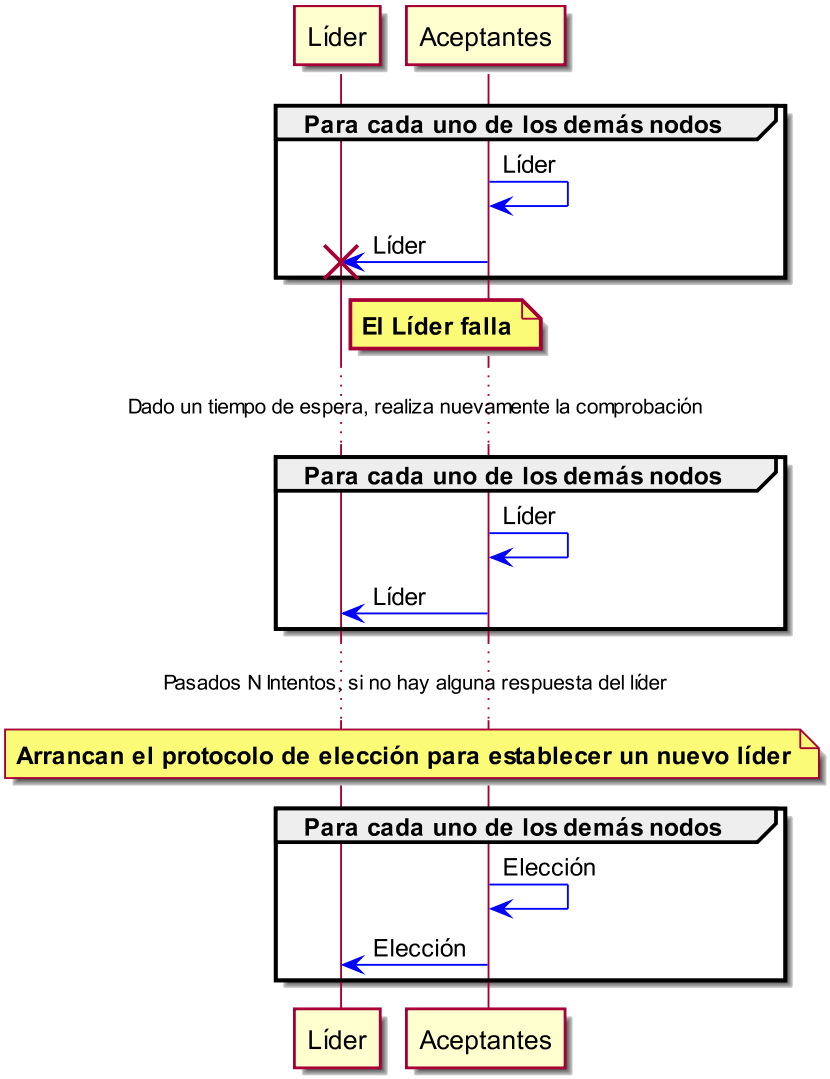


Figura 3.4: Los aceptantes inician el protocolo de elección

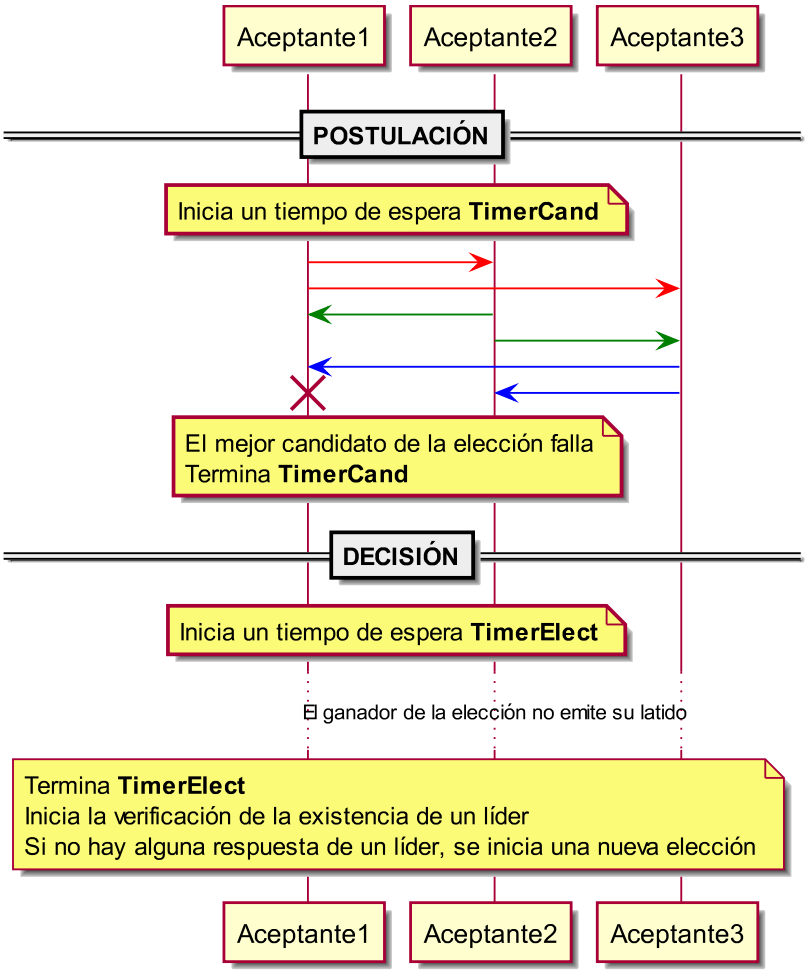


Figura 3.5: Mecanismo de temporizadores en la elección de líder

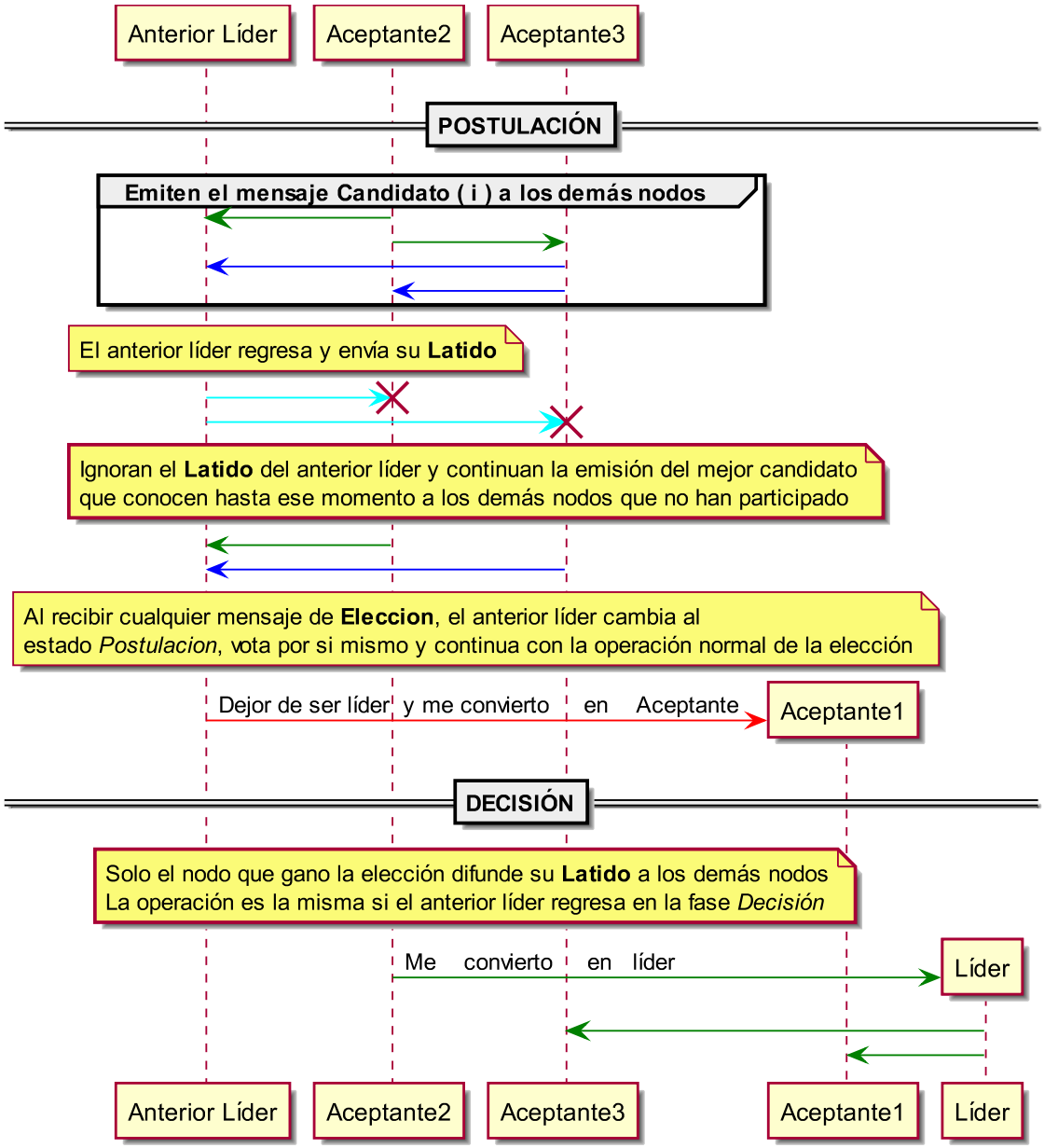


Figura 3.6: Un anterior líder regresa en el transcurso del protocolo de elección de líder

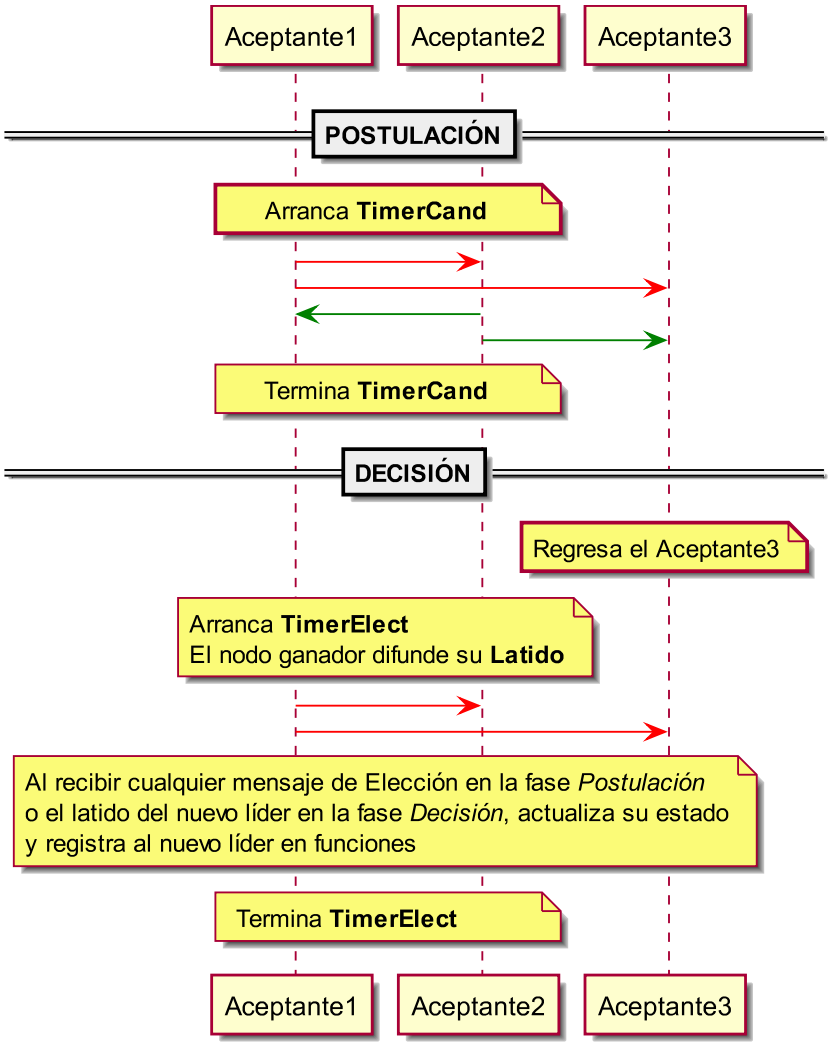


Figura 3.7: Un aceptante regresa en el transcurso del protocolo de elección de líder

3.2. Duplicación de funciones

Se sabe que el protocolo Paxos necesita de un líder único para impulsar el consenso. Sin embargo, es posible que durante lapsos muy breves y luego de una combinación muy particular de eventos, existan dos nodos que asuman este rol al mismo tiempo, imposibilitando con ello el progreso del protocolo. Esta situación se presenta si un líder ausente se recupera justo al terminar la elección que se arranca para sustituir a un líder caído. Entonces habrán dos líderes en funciones.

Se observó que el anterior líder asume que sigue teniendo autoridad y emitirá su latido a los demás nodos. Para evitar este conflicto se introdujo un número de secuencia que se incrementa cada vez que se emite un latido. Al recibir este mensaje, los demás nodos registran el número de secuencia.

Esto permite que un nuevo líder recupere este número. De esta forma, se sabe que el líder es el único que establece el número de secuencia de latido, por lo tanto, los aceptantes sólo recibirán por bueno un latido con el número de secuencia que esperan y que proceda del único líder al que reconocen.

Cuando el líder que regresa recibe un latido desde otro nodo, y con un número de secuencia mayor del que tenía registrado, entonces reconoce que su rol ha expirado y queda como un simple aceptante. En la figura 3.8 se muestra el mecanismo descrito anteriormente.

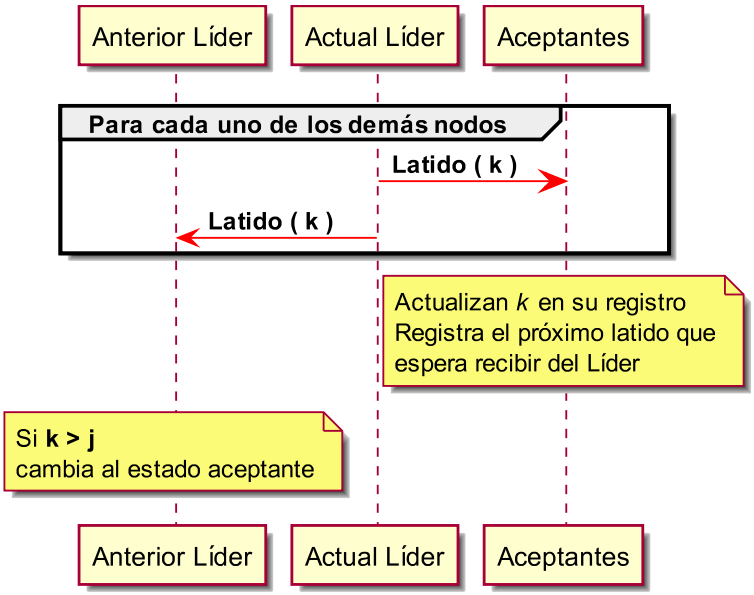


Figura 3.8: Un anterior líder reconoce que ha terminado su liderazgo

Se tiene un caso particular donde ambos líderes en funciones emiten y reciben el mismo número de secuencia de latido. Para solucionar este conflicto, cualquiera de ellos debería arrancar nuevamente la fase de elección para establecer a un nuevo líder y sacar adelante el consenso. Sin embargo en bien de la economía del protocolo, el líder con el mayor identificador será el que gane el desempate y se establecerá como el líder en funciones. Al recibir el latido proveniente de un nodo con un mayor identificador, el otro líder declinará este rol. Se muestra el mecanismo mencionado en la figura 3.9.

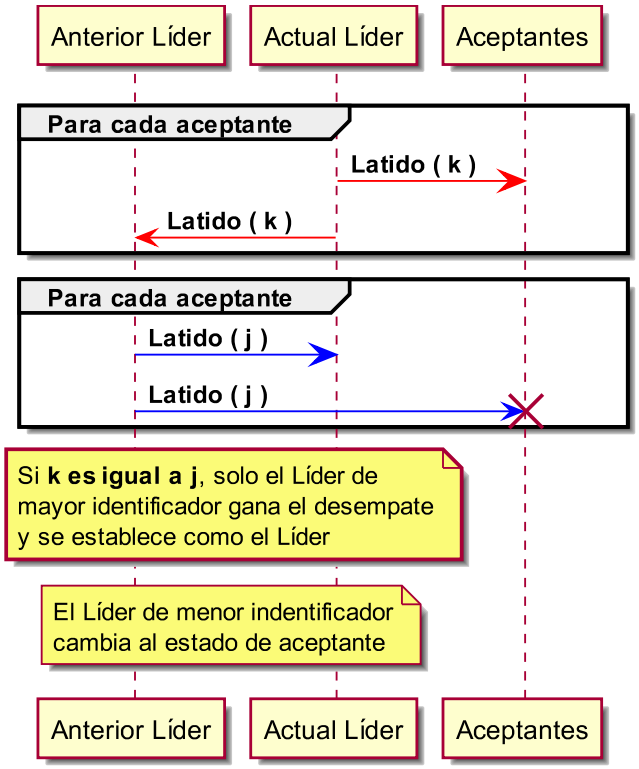


Figura 3.9: Conflicto entre líderes con el mismo número de latido

Por otro lado, los aceptantes sólo registran el latido del líder del que tienen conocimiento o con un número de secuencia mayor. Al recibir el latido proveniente de un nodo que no reconocen como líder o con un número de secuencia menor, los aceptantes hacen caso omiso de este latido. Eventualmente el líder real emitirá su latido a todos los demás pares en servicio, el nodo que cree seguir siendo el líder recibirá el latido y si éste es mayor del que tiene registrado declinará este rol y asumirá el rol de aceptante. En la figura 3.10 se muestra el mecanismo descrito.

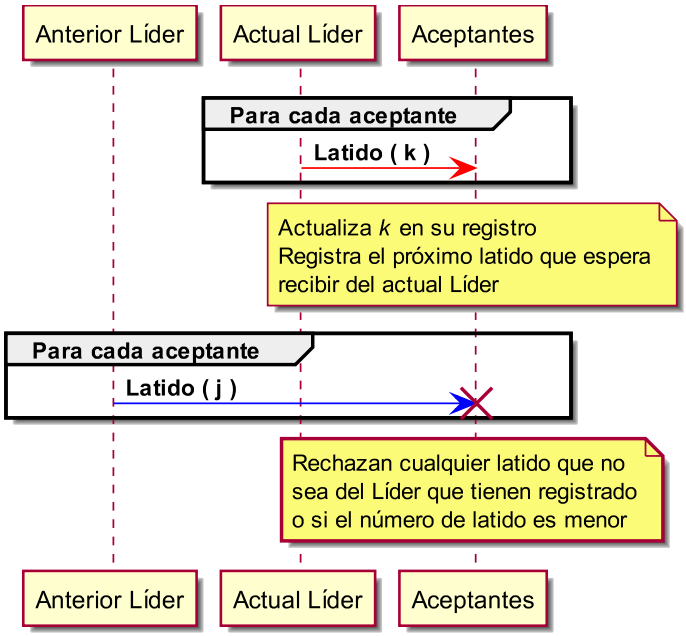


Figura 3.10: Los aceptantes solo guardan el latido del líder que tienen registrado

3.3. Asignación del número de instancia

Se ejecutan instancias del algoritmo Paxos para mantener la consistencia de la información relativa a esa instancia. El valor acordado para cada instancia está registrado o debe registrarse en cada uno de los nodos participantes. Estas instancias son numeradas de manera secuencial.

Para ello se introduce un mecanismo que permite a cada nodo cliente establecer el número de la instancia que él origina (es el dueño). Este mecanismo garantiza que cada número es único y sólo los dueños de la instancia pueden asignarlo.

A continuación se describe este mecanismo: se tiene un registro que se incrementa en una unidad cada vez que un cliente establece una solicitud, llamado *vista*, también cada nodo conoce el total de nodos con los que trabaja, al que llamaremos P, se sabe que cada nodo tiene un número entero que lo identifica de manera única llamado $ID : 1, \ldots, P$.

Mediante una sencilla operación matemática se asigna un número de instancia igual a $(vista * P) + (ID - 1)$. Dicho de otra forma, todos los números de instancia de un mismo dueño son congruentes con su identificador, en módulo P. En la figura 3.11 se muestra el procedimiento descrito.

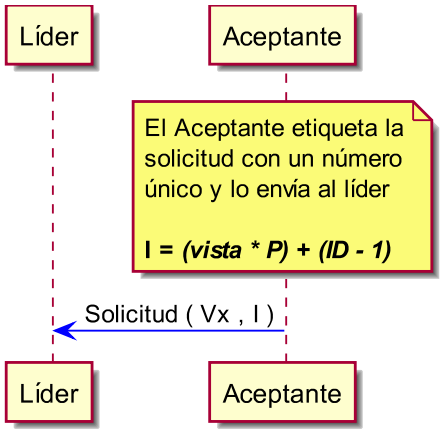


Figura 3.11: Asignación del número de instancia de la que es dueño el aceptante

3.4. Consistencia y actualización de los registros

Luego de recuperarse de una falla, un nodo se reincorpora al protocolo y puede reconocer la presencia de "huecos" en su registro. Esto es, instancias de las que desconoce el valor acordado. El mecanismo utilizado para determinar las instancias faltantes es el siguiente: Se sabe que el aceptante tiene conocimiento del próximo latido del líder. Si el número de latido que recibe coincide con el que espera, se asume que ha estado activo junto con el líder, entonces actualiza este número y continua su operación normal.

Por otro lado, si el número de latido es mayor, se asume que el aceptante estuvo ausente y arranca un procedimiento de actualización. En cada nodo aceptante utilizamos una variable llamada *apuntador* que indica la última instancia consecutiva para la que se acordó un valor, en dicho nodo. Por ejemplo, si tenemos las instancias $[1, 2, 3, 5, 8, 12, 17, 19]$, el valor del *apuntador* es 3.

Cuando el aceptante sabe que estuvo ausente, busca en sus registros las instancias faltantes a partir de un recorrido secuencial, que es iniciado desde el valor del *apuntador* hasta la última instancia registrada hasta entonces y las encola en su lista de instancias faltantes. En la figura 3.12 se muestra el mecanismo descrito.

Sin importar si es el dueño o no de una instancia faltante, es decir, instancias en las que no estuvo presente, el nodo debe intentar ponerse al día. Se analizan los siguientes casos: (a) el líder debe actualizar sus registros, (b) un aceptante debe mantenerse al día acerca de las instancias de las que es el dueño y (c) un aceptante determina qué instancias le hacen falta en sus registros, pero no es dueño de las mismas.

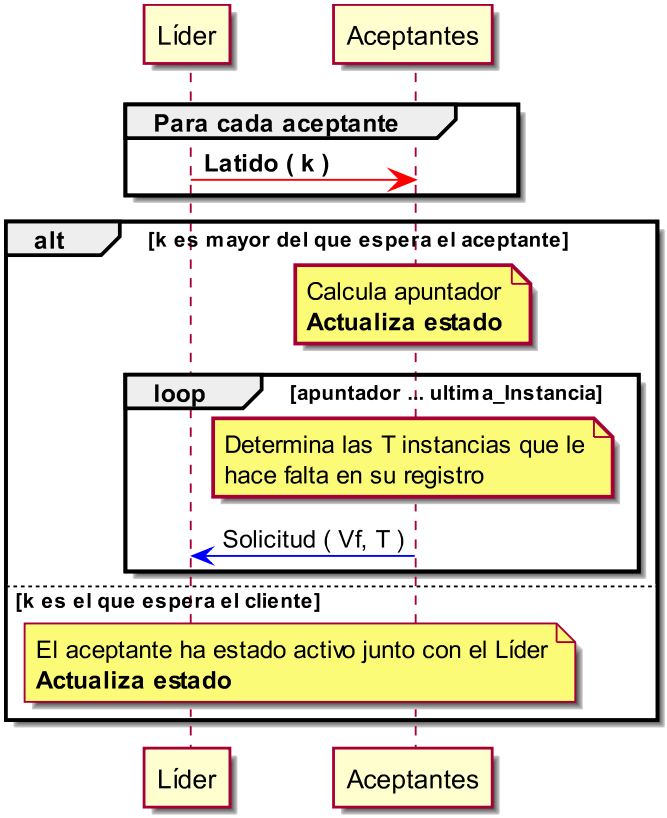


Figura 3.12: Un aceptante reconoce si ha estado activo junto con el líder

En el primer caso, se sabe que las solicitudes de los clientes tienen un número de instancia que es asignado por el cliente que origina la instancia (i.e. es el dueño de la misma).

En condiciones normales, se ejecturará una instancia del protocolo Paxos para replicar el registro que recibe. El líder determina si el cliente es dueño de la instancia que solicita mediante una operación modular P, donde P es el tamaño del conjunto total de nodos. Si le corresponde esa instancia, entonces verifica en qué fase del algoritmo Paxos se encuentra.

Recordemos que Paxos se divide en dos fases. Si se encuentra en estado inicial ejecuta el consenso a partir de la fase 1_a *Prepara*, en caso contrario ejecuta la fase 2_a *Aceptar*. En la figura 3.13 se muestra el mecanismo descrito.

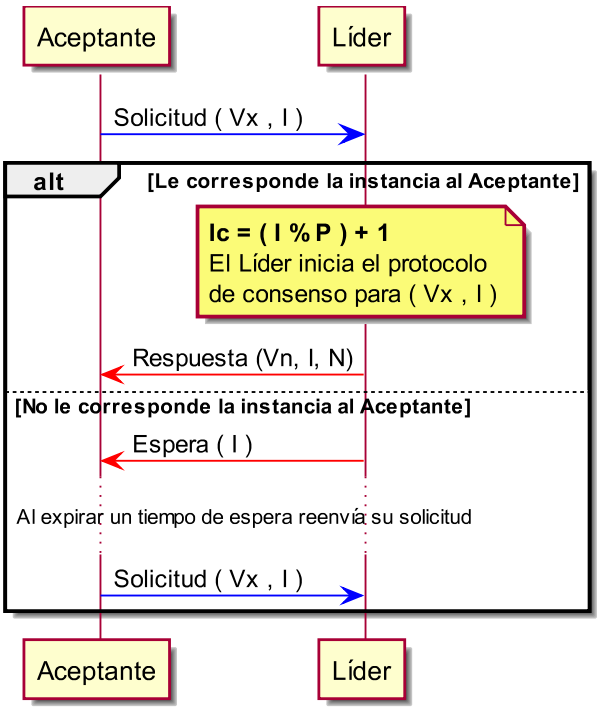


Figura 3.13: El líder determina si realiza la solicitud del cliente

Sin embargo, el propio líder podría tener huecos en su registros, para detectar esta contingencia y poder resolverla, el líder registra el valor de la siguiente instancia que espera recibir del cliente que recién atendió y así sucesivamente.

Si el líder recibiera una solicitud de un cliente con un número de instancia mayor del que él espera, entonces reconoce que le faltan anteriores instancias de ese cliente y arranca la etapa de actualización iniciando el protocolo para establecer el consenso de la instancia recibida y de las instancias faltantes.

Para cada instancia faltante, el líder inicia la fase inicial del protocolo con un valor ficticio, un número de ronda igual a 0 y el número de instancia. Cuando se ejecuta una instancia Paxos, se sabe que los aceptantes rechazarán el mensaje *PREPARA* ya que el número de ronda es menor del que tienen conocimiento de esa instancia. Igualmente, en ese mensaje que devuelven comunican el valor acordado, así como la ronda en la que se alcanzó el consenso.

A continuación el líder emite un nuevo mensaje *PREPARA* pero con un número de ronda superior y un valor actualizado. A partir de aquí se sigue el funcionamiento del protocolo Paxos descrito en la sección 2.4. En la figura 3.14 se muestra el protocolo de actualización de los registros del líder.

Para el segundo caso, esto es, cuando el nodo se recupera de su falla y determina las instancias faltantes, se asume que el líder tiene sus registros al día. De otro modo, regresamos al caso 1 y luego volvemos al caso 2. Puesto que el líder tiene un registro actualizado de las instancias exitosas, puede responder inmediatamente a las solicitudes de los nodos que requieran actualizar sus registros.

Si un cliente le manda una solicitud al líder y el líder tiene registro de esa instancia, le responde con un mensaje *RESPUESTA* que incluye el valor acordado, el cliente recibe el mensaje y lo guarda en su registro.

Para el tercer caso, cuando un nodo detecta una instancia faltante de la que no es dueño, el nodo invoca los servicios del líder para ponerse al día. Sin embargo, el líder mismo podría tener un registro desactualizado y no disponer de la información que le solicitan. Como hemos visto, un cliente asigna un número de instancia único a su solicitud, sin embargo sabemos que este número de instancia puede usarse por los demás nodos.

Cuando el líder recibe una solicitud y verifica que no le corresponde esa instancia al cliente que lo emite, se introduce un mecanismo de espera, es decir, al saber el líder que el cliente no es dueño de esta instancia, le manda un mensaje *ESPERA* al cliente indicando que vuelva a emitir su solicitud dentro de un periodo de tiempo determinado. En la figura 3.15 se muestra el funcionamiento del protocolo de consistencia de los registros de los aceptantes.

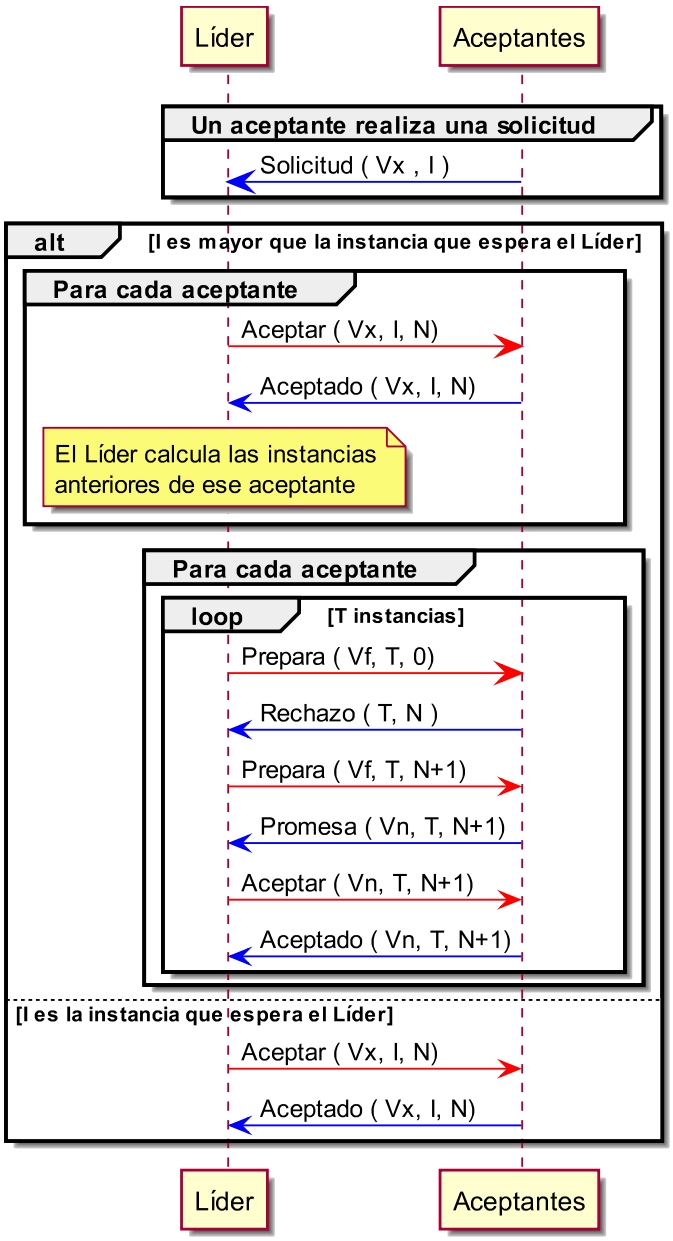


Figura 3.14: Consistencia de los registros del líder

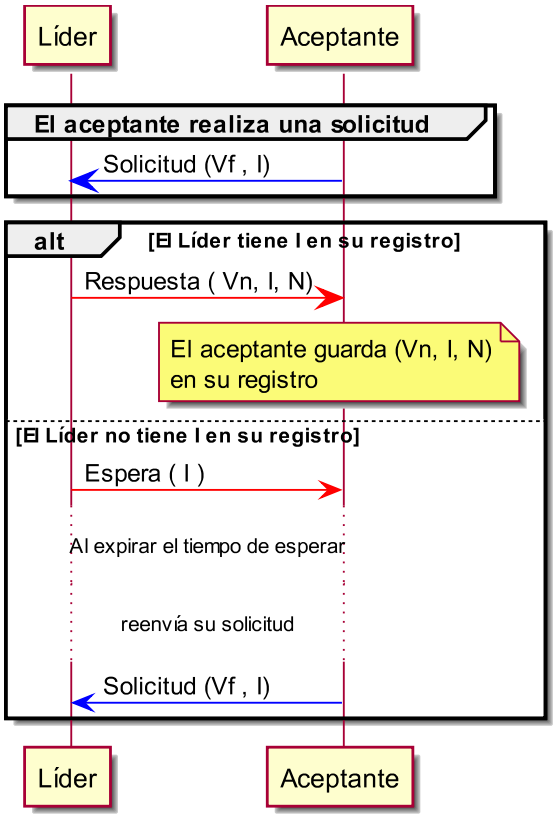


Figura 3.15: Mecanismo para tener consistente los registros de un aceptante

Con esto se garantiza que sólo los valores de los clientes que son dueños de sus propias instancias se pueden aprender. Luego de que el dueño de una instancia faltante actualice su registro, el líder podrá responder a todos los demás nodos, acerca del resultado de esta instancia. Esperando con ello que para cuando vuelvan a buscarle preguntando por la instancia, ya disponga de la información que se le requiere. En otro caso, es decir, si el líder cuenta con un registro al día, entonces puede responder inmediatamente.

Para el segundo y tercer caso el cliente envía su solicitud al líder con un valor ficticio junto con el número de instancia faltante, el líder realiza las operaciones descritas según sea el caso. Es importante mencionar que se incluye un mecanismo para que el cliente sólo encole las instancias faltantes pero no las instancias pendientes, es decir, instancias de las que es dueño pero no fueron atendidas por la ausencia de un líder, esto se describe con mas detalle en la sección 3.5. En la figura 3.16 se muestra el mecanismo que permite encolar las solicitudes de las instancias pendientes y faltantes de un cliente.

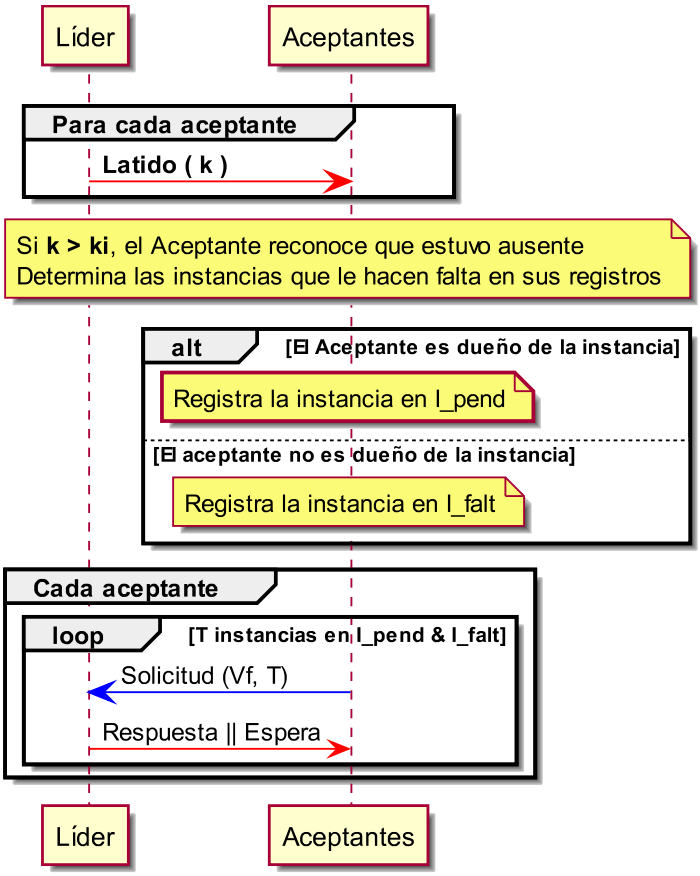


Figura 3.16: Mecanismo para las asignaciones de instancias faltantes y pendientes

3.5. Solicitudes pendientes

Este escenario se presenta cuando los clientes someten una solicitud al líder y éste cae en paro. Para solucionar este problema, al momento de que un aceptante envía una solicitud al líder, la almacena temporalmente en una lista de instancias pendientes, que permite conocer que instancias no han sido atendidas. Recordemos que cada nodo maneja sus propias instancias y que tenemos un almacenamiento estable.

Cuando el líder le responde a un cliente, el cliente quita la instancia que emitió de la lista de instancias pendientes. Esto pasa también si existiera un nuevo líder, en cuyo caso, al tener un nuevo líder, eventualmente el cliente enviará su solicitud al nuevo líder para que pueda ser atendida. En la figura 3.17 se muestra el mecanismo descrito.

3.6. Falta de cuórum

Esta situación se presenta cuando el líder se encuentra coordinando una instancia del protocolo pero no existe un cuórum de aceptantes que le responda, es decir, una mayoría. Esto se debe a que más de la mitad del conjunto de nodos fallan. Sin embargo, aquí tenemos un simple mecanismo de control para solucionar este problema y depende del estado en que se encuentra el líder.

Cuando el líder se encuentra en fase inicial o en fase estable, arranca un tiempo de espera llamado $Timer_{00}$ y $Timer_{01}$, para cada fase del protocolo respectivamente. El líder queda en espera de recibir respuestas de un cuórum de aceptantes. Si expira el timer correspondiente y no existiera un cuórum para sacar adelante la instancia en progreso, entonces el líder hará caso omiso de la solicitud y continuara en funciones hasta que exista una mayoría y reciba nuevamente la solicitud del dueño de la instancia, para finalmente atenderla y duplicarla consistentemente en cada uno de los demás nodos.

Con esto se garantiza mantener el progreso del consenso a pesar de que no exista un cuórum, en algún momento, los clientes cuando se recuperen de su fallo y arranquen el protocolo de actualización de sus registros, verificará las instancias que le hacen falta en sus registros y enviará las solicitudes de dichas instancias para que posteriormente sean atendidas por el líder en funciones. En la figura 3.18 se muestra el mecanismo de los tiempos de espera $Timer_{00}$ y $Timer_{01}$.

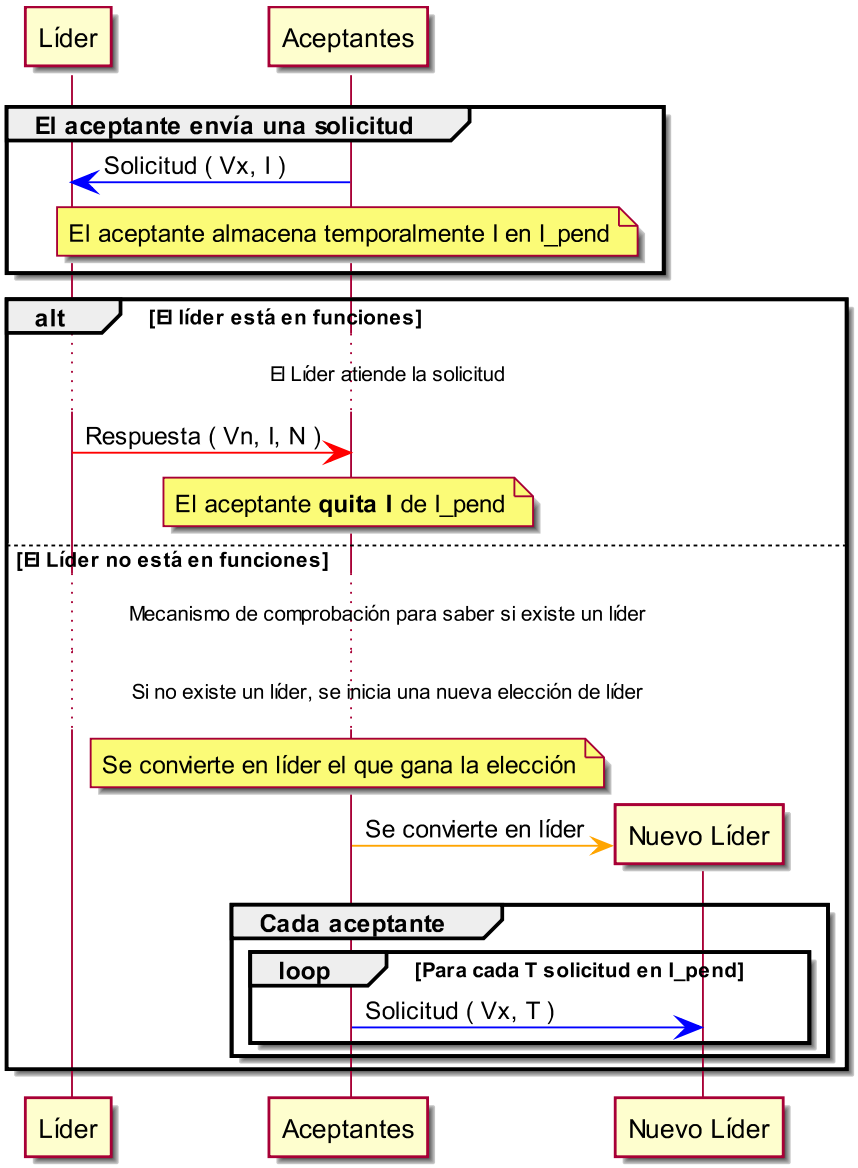


Figura 3.17: Mecanismo que permite conocer que instancias han sido atendidas

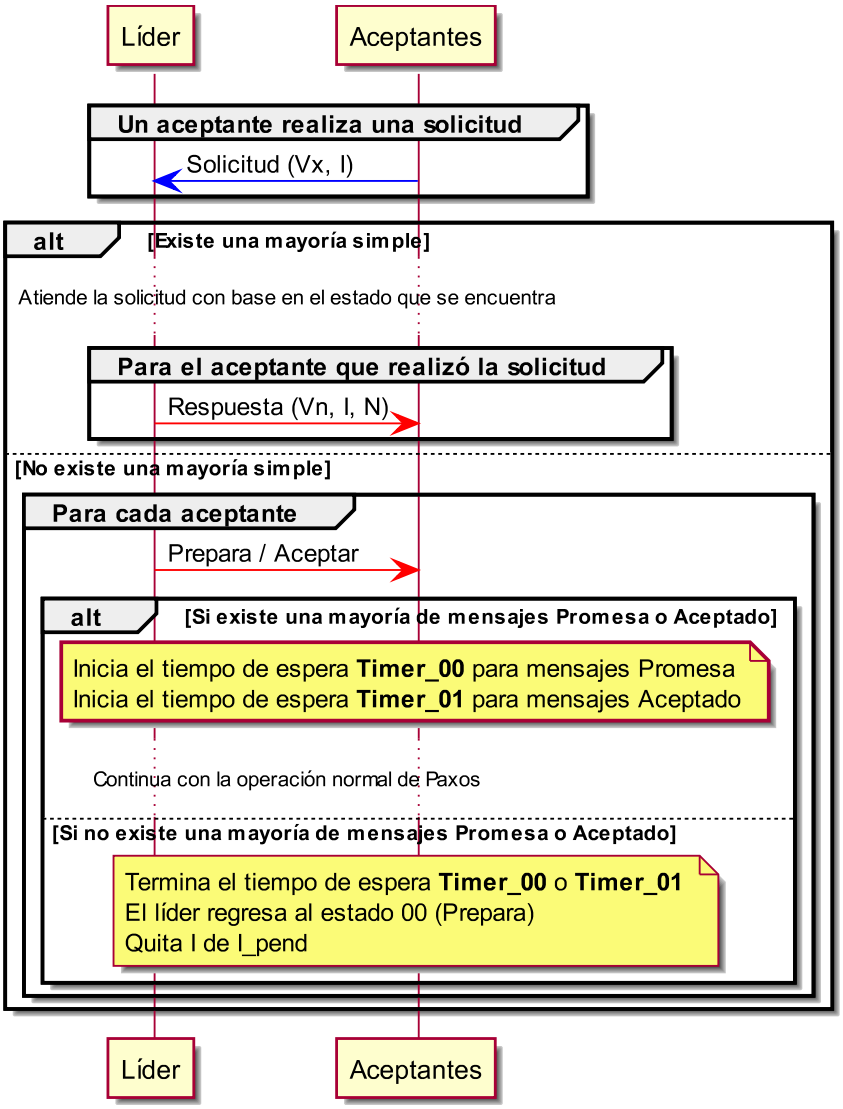


Figura 3.18: Mecanismo del tiempo de espera para la recepción de mensajes promesa o aceptado en el estado inicial

CAPÍTULO

4

EVALUACIÓN DE RENDIMIENTO

En este capítulo se presentan los resultados de una serie de experimentos a los cuales sometimos nuestro prototipo en diferentes escenarios. Los escenarios que se presentan fueron probados en un simulador de eventos discretos escrito en el lenguaje python. Se sometió el prototipo con diferentes configuraciones como son el número de proxies, tasas de falla y recuperación y tasas de solicitudes.

El objetivo fue desarrollar un módulo o middleware que permite realizar la funcionalidad de multipaxos en un clúster de nodos, el cual cumplió cabalmente con la operaciones del protocolo. También se evaluó el rendimiento de las fallas de cuórum, es decir, cuando no existía una mayoría simple de aceptantes para progresar en el consenso. Cabe mencionar que la evaluación se realizó con el propio algoritmo Paxos, es decir, no se comparó con otro algoritmo, también se sabe que, en fase estable, Paxos es óptimo en el costo de mensajes [42] y no existe hasta el momento un mejor algoritmo para solucionar el problema de consenso.

4.1. Hardware

Se utilizo un equipo para correr las diferentes simulaciones, mediante un simulador de eventos discretos codificado en python. Las configuraciones que se usaron se describen en las siguientes

secciones. Las especificaciones del equipo se presentan en la tabla 4.1.

Memoria RAM	14 GB
Procesador	Intel(R) Xeon(R) CPU E31220 @ 3.10 GHz
Sistema operativo	CentOS 6.5

Tabla 4.1: Especificaciones de los equipos usados

4.2. Fallas y su recuperación

Los escenarios de los experimentos realizados involucran fallas de paro con su recuperación con tasas aleatorias en cada nodo y son manejados por el simulador de eventos discretos usando diferentes parámetros por cada experimento. El nodo que falla no emprende alguna acción especial durante el experimento, esto significa que el nodo no participa en instancias del algoritmo Paxos o en la elección de un nuevo líder durante el experimento.

Tanto los instantes de falla, como los de recuperación se simulan usando timers específicos, que obedecen a tiempos aleatorios uniformemente distribuidos. Estos tiempos se muestran en la tabla 4.2.

Tasa de falla (seg)	**Tasa de recuperación (seg)**
1-1000	1-10
	1-100
	1-1000

Tabla 4.2: Tasas utilizadas de falla y recuperación en los experimentos

4.3. Diseño de experimentos

Cada nodo participa en el experimento y emite solicitudes al líder para duplicar los registros del nodo en todos los demás pares en servicio, es decir, ejecutar instancias Paxos que garantizan mantener de forma consistente la información que administran. Todas las solicitudes contienen un número de instancia asignado por el nodo que la emite y un número entero generado de forma pseudoaleatoria, sobre el que se quiere conseguir el consenso. Todos los experimentos se simularon con una duración de 100,000 unidades de tiempo.

Cada nodo solicita los servicios del líder con base en las tasas de solicitudes, estas tasas se presentan en la tabla 4.3. Los nodos emiten las solicitudes dentro de los tiempos asignados sin esperar una respuesta de que la solicitud ha sido atendida satisfactoriamente, es decir, los nodos no esperan algún periodo de tiempo para volver a enviar una nueva solicitud.

Tasa de solicitudes (seg)
1-10
1-100
1-1000

Tabla 4.3: Tasa de solicitudes utilizada en los experimentos

Los experimentos fueron realizados con diferentes tamaños de clúster, se asignó el intervalo del tamaño a 3, 5, 7, 9 y 11 nodos. Con esto se obtuvo una serie de experimentos que combinaban diferentes configuraciones, es decir, para cada tamaño de clúster, se asignó cada una de las tasas de falla y recuperación junto con cada tasa de solicitudes como se describió anteriormente.

4.4. Fallas en cuórum

En cada experimento se obtuvo el porcentaje de las fallas en cuórum y se determino el número óptimo de nodos que permitan garantizar avanzar en el protocolo ante la presencia de fallas.

A continuación se muestran los resultados obtenidos al someter nuestro prototipo a diferentes simulaciones. Las tasas de recuperación utilizadas en los experimentos son los descritos en la tabla 4.2 y se sometieron con cada una de las tasas de solicitud (S)de la tabla 4.3.

En la figura 4.1, se muestra la primera configuración establecida con tasa de recuperación de tiempo aleatorio de 1-10 (seg.), y con cada una de las tasas de solicitud. Los resultados de este experimento en todos los tamaños de clúster son 0, es decir, no hubo ausencia de cúorum, ya que al tener una rápida recuperación en los nodos, siempre existía una mayoría simple para sacar adelante una instancia del protocolo Paxos. Obsérvese que la tasa de recuperación es una centésima parte con respecto a la tasa de falla.

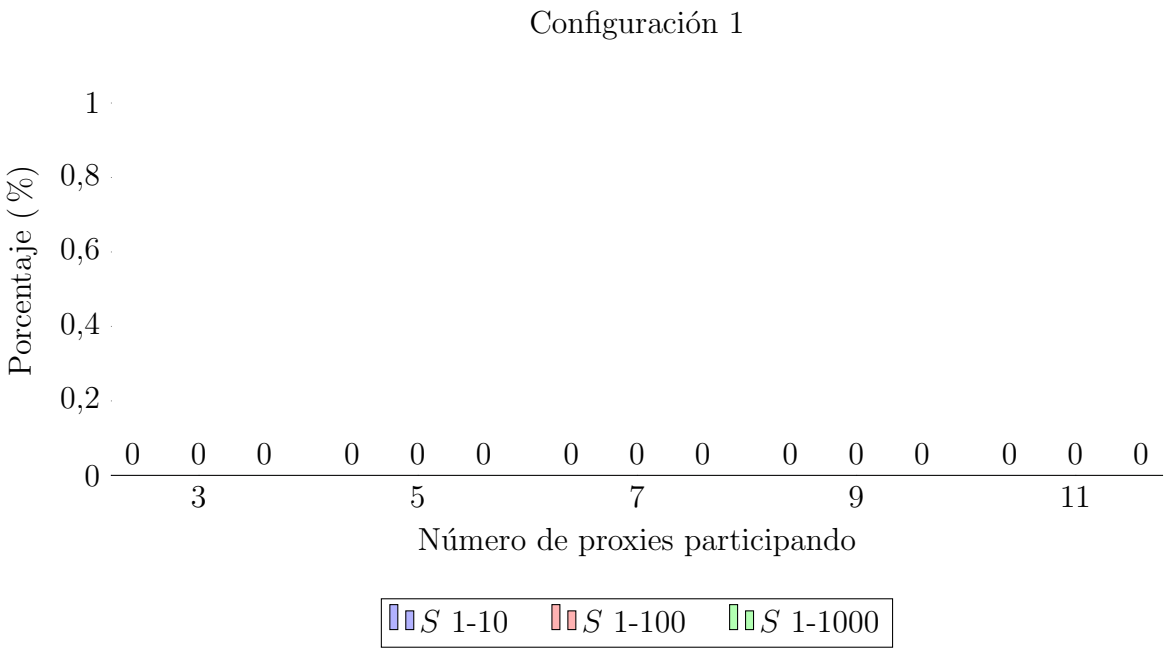


Figura 4.1: Porcentaje de exp. con fallas de cuórum. Tasa de recuperación: 1-10 seg.

En la figura 4.2, se muestra la segunda configuración establecida con tasa de recuperación de tiempo aleatorio de 1-100 (seg.), de igual manera con cada una de las tasas de solicitud. Los resultados de este experimento muestran que con la tasa de solicitudes 1-10(seg.), cualquier tamaño de clúster funciona sin contratiempos y son capaces de realizar el progreso.

Con tasa de solicitudes de 1-100(seg.), se observa que los tamaños de clúster de 3 y 5 proxies, presentan un ligero porcentaje en ausencia de cuórum, sin embargo es mínimo este porcentaje. En los demás tamaños no hay ausencia de cuórum.

Con tasa de solicitudes 1-1000(seg.) hubo un incremento en el porcentaje de ausencia de cúorum para todos los tamaños de clúster, sin embargo sigue siendo mínimo el impacto que afecta el progreso del protocolo Paxos. En resumen, los tamaños de clúster de 7,9 y 11 proxies son ideales para esta configuración. Obsérvese que la tasa de recuperación es una décima parte con respecto a la tasa de falla.

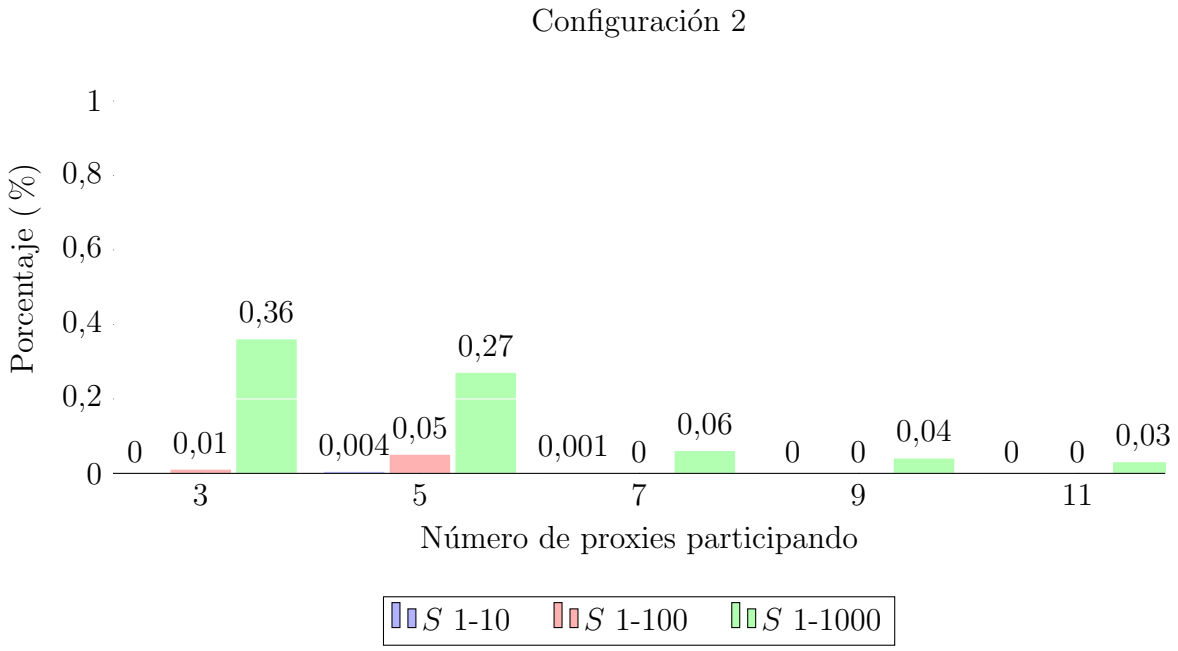


Figura 4.2: Porcentaje de exp. con fallas de cuóurum. Tasa de recuperación: 1-100 seg.

En la figura 4.3, se muestra el resultado de la configuración 3, con tasa de recuperación de 1-1000(seg.). Obsérvese que en este experimento, las tasas de recuperación son equiparables con las tasas de falla.

En esta configuración, se observó el peor escenario en donde existe un mayor retraso de recuperación para los proxies, para este experimento se observó que con el mínimo tamaño del clúster, es decir, con 3 nodos se tiene un mejor rendimiento ya que solo necesitamos cualesquiera 2 nodos para sacar adelante la solución de consenso.

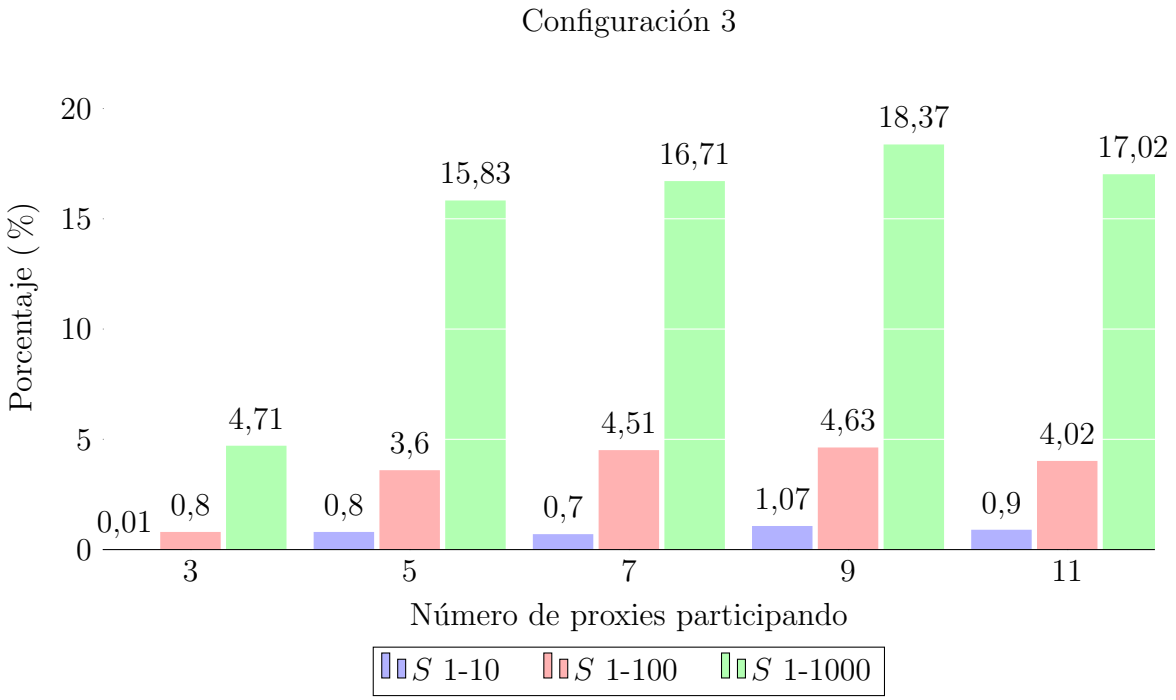


Figura 4.3: Porcentaje de exp. con fallas de cuóurum. Tasa de recuperación: 1-1000 seg.

CAPÍTULO

5

CONCLUSIONES Y TRABAJO FUTURO

El algoritmo Paxos permite conseguir que un grupo de máquinas lleguen a un acuerdo sobre un valor. Multi-Paxos es una optimización en el uso de Paxos para manejar varias instancias del mismo problema.

Se sabe que, bajo ciertas condiciones de operación, Paxos garantiza la solución de consenso. Consideramos que nuestra contribución fue reconocer los escenarios en los que dichas condiciones pueden dejar de cumplirse. Gracias a que pudimos inyectar fallas arbitrarias durante la ejecución del protocolo en escenarios que configuramos a nuestra conveniencia.

Asimismo, se explicaron los mecanismos que permiten solucionar estos problemas para restablecer las condiciones de operación, que garantizan el progreso del protocolo. Se puede demostrar que la fase 2 del algoritmo de consenso Paxos tiene el costo mínimo posible que cualquier otro algoritmo de consenso durante situaciones de estabilidad [42]. Por lo tanto, el algoritmo Paxos es esencialmente óptimo.

Paxos es un algoritmo muy robusto, permitiendo que cualquier mayoría simple de servidores progrese de manera segura. Esto hace que sea conveniente para situaciones que requieren durabilidad y un alto nivel de confiabilidad.

Un trabajo futuro es la implementación de un mecanismo que permita al middleware manejar de manera sencilla los números de secuencia de latidos que emite el líder para tener un mejor control sobre estos al momento de alcanzar grandes números de secuencia.

También se considera la posibilidad que el prototipo se desarrolle en otros lenguajes, así como su evaluación de rendimiento en servicios web y cómputo en la nube.

Otra característica a considerar a futuro es el desarrollo de un mecanismo para reducir la latencia de las comunicaciones en la red.

En este trabajo no nos centramos en el control del tamaño de los registros en un nodo, es decir, el mecanismo para realizar las copias de seguridad de los registros y así reducir el tamaño de espacio en disco, esta parte se puede considerar como otro trabajo futuro.

Apéndices

APÉNDICES

A

DIAGRAMAS DE ESTADO DE MULTIPAXOS

Tabla A.1: Descripción de los estados y objetos: Líder

Nombre	**Descripción**
E00	Determina si ejecuta una instancia Paxos
E01	Verifica si el cliente es el dueño de la instancia
E02	Actualiza sus registros
E03	Atiende las instancias pendientes
E04	Existe más de un líder en funciones
E05	Etapa *Prepara* de Paxos
E06	Etapa *Aceptar* de Paxos (Fase inicial)
E07	Notificación al cliente que su solicitud fue realizada (Fase inicial)
Continúa en la siguiente página	

Nombre	Descripción
E08	Etapa *Aceptar* de Paxos (Fase estable)
E09	Notificación al cliente que su solicitud fue realizada (Fase estable)
I_{pend}	Lista que almacena las instancias que no han sido atendidas
I_{last-c}	Última instancia registrada del cliente
I_{next}	Siguiente instancia que espera el líder del cliente
I_x	Variable temporal para calcular las instancias faltantes
V_f	Valor ficticio colocado en la solicitud para las instancias faltantes
V_i	Valor obtenido que se establece en la solicitud al recibir el mensaje *Promesa* de un cliente
N	Número de propuesta asignado a una solicitud
contador	Determina si existe una mayoría simple de respuestas
f	Representa la mitad del conjunto total de procesos
$Id_{líder}$	Número entero único que identifica al líder
$Timer_{00}$	Tiempo de espera para la etapa *Prepara* (Fase inicial)
$Timer_{01}$	Tiempo de espera para la etapa *Aceptar* (Fase inicial y estable)
k	Número de secuencia del latido que emite el líder

Tabla A.2: Descripción de los estados y objetos: Cualquier rol que sea diferente del líder

Nombre	Descripción
E10	Etapa *Promesa* de Paxos
E11	Etapa *Aceptado* de Paxos
E12	Etapa estable para la recepción de solicitudes con el mismo líder
E13	Verificación de la existencia de un líder
Continúa en la siguiente página	

Nombre	Descripción
E14	Determinar las instancias faltantes de los registros
E15	Envió de solicitudes para las instancias pendientes y faltantes de un cliente
I_{falt}	Lista que almacena temporalmente las instancias faltantes
I_{pend}	Lista que almacena temporalmente las instancias que no fueron atendidas por el líder
$ultimaPromesa$	Número de propuesta con la que se ha comprometido con un líder
N_j	Propuesta reportada al líder por el cliente
V_c	Valor que le interesa al cliente llegar a un acuerdo
I_c	Número de instancia que establece el cliente en su solicitud
I_f	Número de instancia que le hace falta al cliente
$Timer_{Latido}$	Tiempo de espera para recibir el próximo latido del líder
$Timer_{Solicitud}$	Tiempo que espera un cliente para volver a reenviar su solicitud
$INTENTOS$	Número de veces que se realiza la verificación de la existencia de un líder
$cont_{int}$	Contador que determina en que momento se realiza una nueva elección de líder
$I_{last-acep}$	Última instancia registrada en orden creciente
I_{fx}	Variable usada para determinar las instancias faltantes de los registros del cliente

Tabla A.3: Descripción de los estados y objetos: Durante el proceso de elección

Nombre	Descripción
E21	Etapa de *Postulación* del protocolo de elección de líder
E22	Etapa de *Decisión* del protocolo de elección de líder
Continúa en la siguiente página	

Nombre	Descripción
E23	Etapa de confirmación para el nuevo líder elegido
C_i	Identificador del cliente que realiza la solicitud
k_i	Número de secuencia del latido del líder que tiene registrado
$Timer_{Cand}$	Tiempo de espera en la fase *Postulación* del protocolo de elección
$Timer_{Elect}$	Tiempo de espera en la fase *Decisión* del protocolo de elección
candidato	Utilizado para determinar el mejor candidato de la elección de líder
líder	Utilizado para establecer quien es el líder en funciones
apuntador	Valor de la última instancia en orden creciente que se tiene en los registros
P	Tamaño total del conjunto de procesos
ID	Número entero único que identifica a cada nodo

Tabla A.4: Significado de las figuras usados en los diagramas

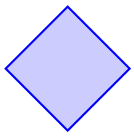
Figura	Descripción
	Estado
	Se recibe un mensaje
	Se envía un mensaje
	Se ejecuta una acción
	Se efectúa una decisión

Tabla A.5: Descripción de los mensajes usados en Multipaxos: Líder

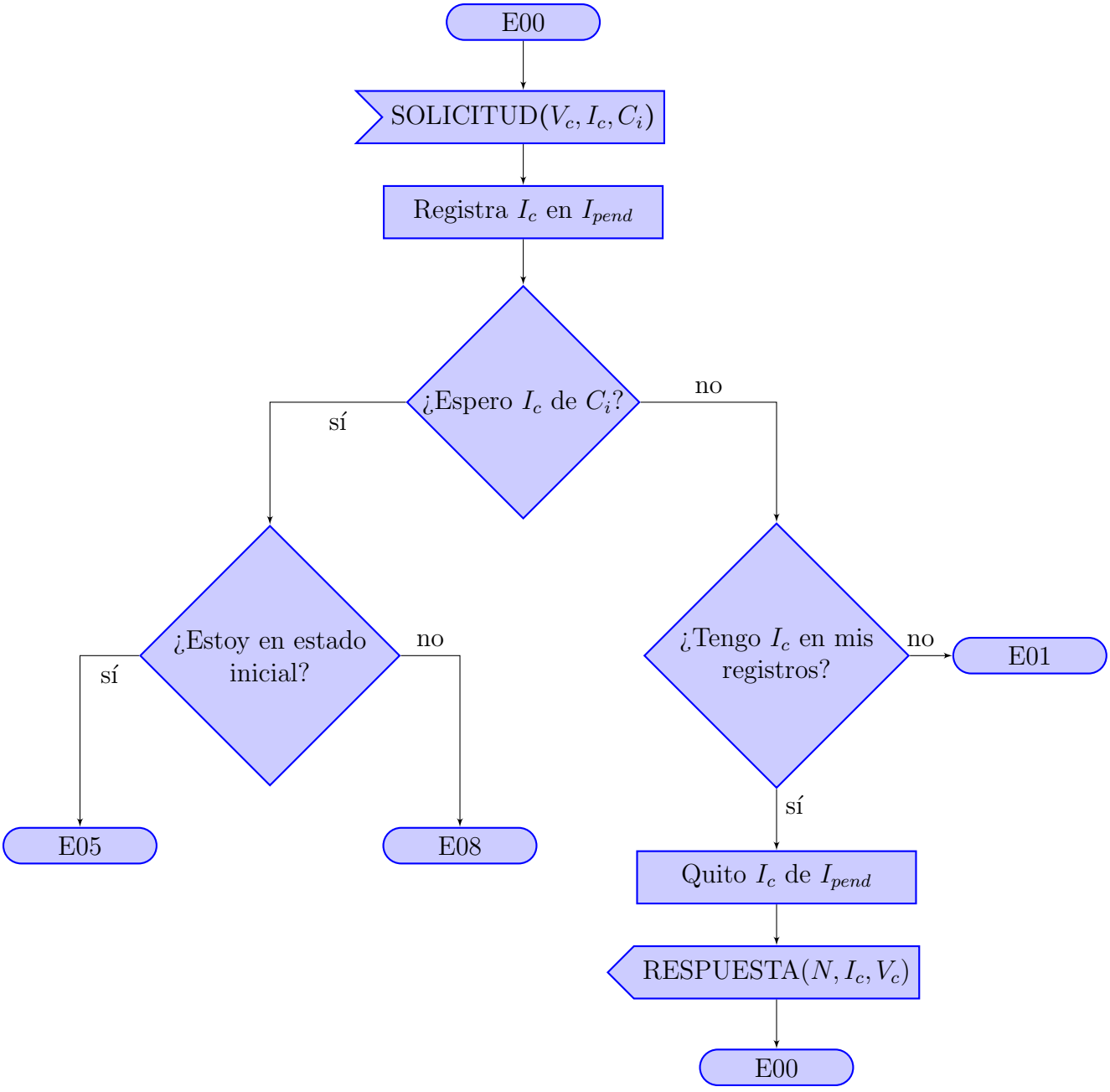
Nombre	**Descripción**
PREPARA	Crea una propuesta identificada con un número de ronda N para alcanzar el consenso sobre un solicitud. También sirve para la actualización de sus registros.
ACEPTAR	Indica a los aceptantes que guarden el valor en sus registros
ESPERA	Indica a un aceptante de reenviar su solicitud después de un tiempo de espera
LATIDO	Determina la autoridad de un líder
RESPUESTA	Actualiza los registros de los aceptantes
COMPLETO	Notifica que la solicitud se realizo con éxito

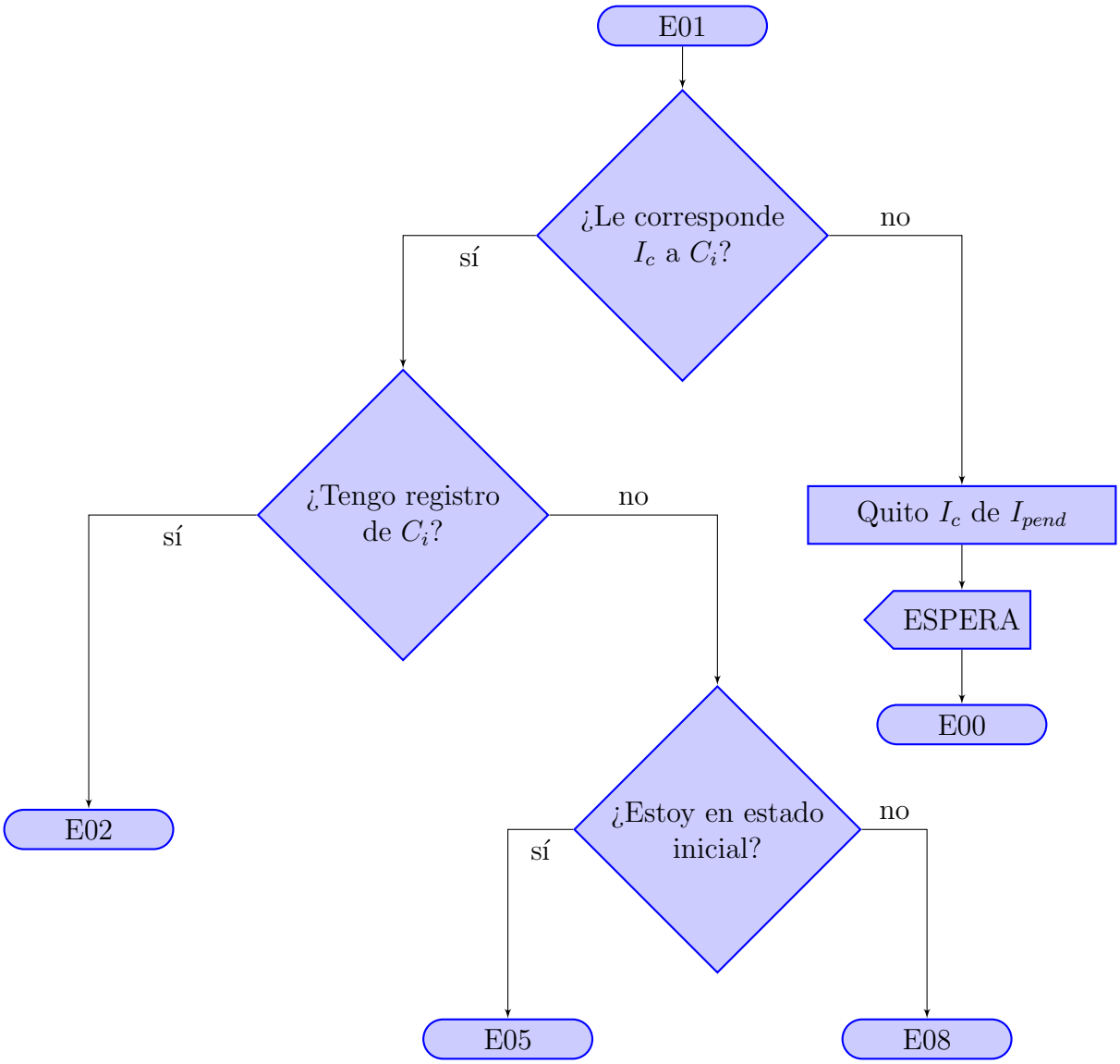
Tabla A.6: Descripción de los mensajes usados en Multipaxos: Cualquier rol que sea diferente del líder

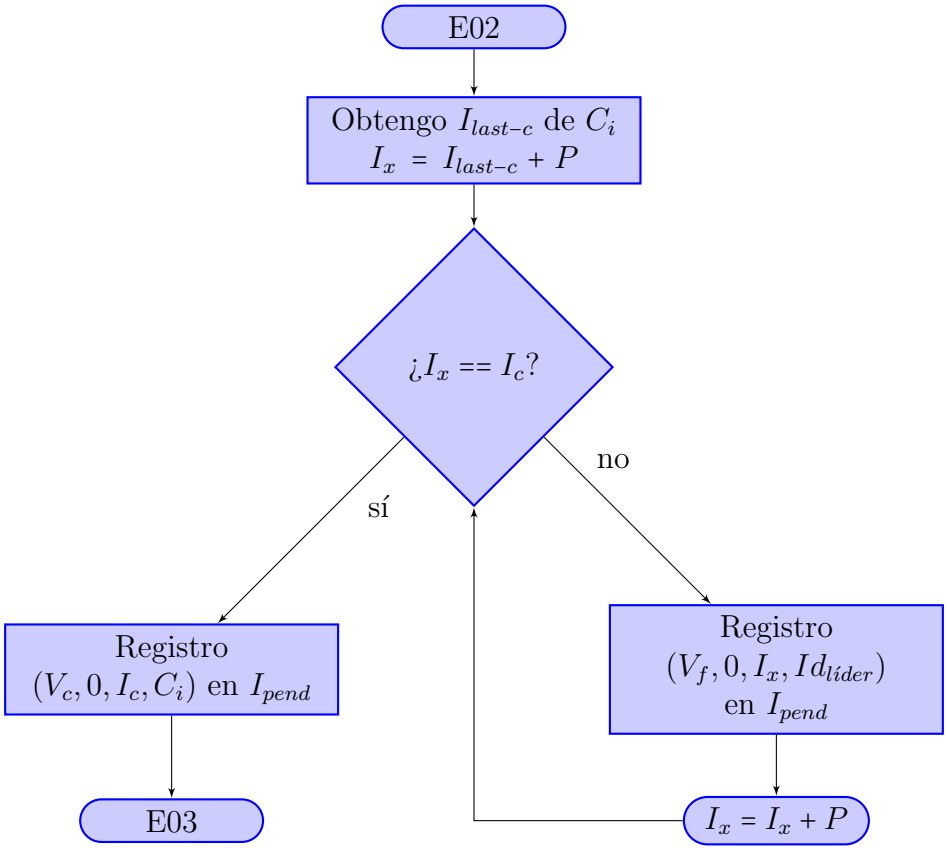
Nombre	**Descripción**
SOLICITUD	Solicitud la cual se quiere alcanzar el consenso sobre un valor
PROMESA	Indica que se compromete a participar con el número de ronda N propuesto. Tambien sirve para la actualización de los registros del líder
ACEPTADO	Indica al líder que el valor fue registrado
RECHAZO	Indica al líder que no participa con el número de ronda N. Le notifica el último número de ronda que se alcanzo para esa solicitud
LÍDER	Verifica si existe un líder en funciones

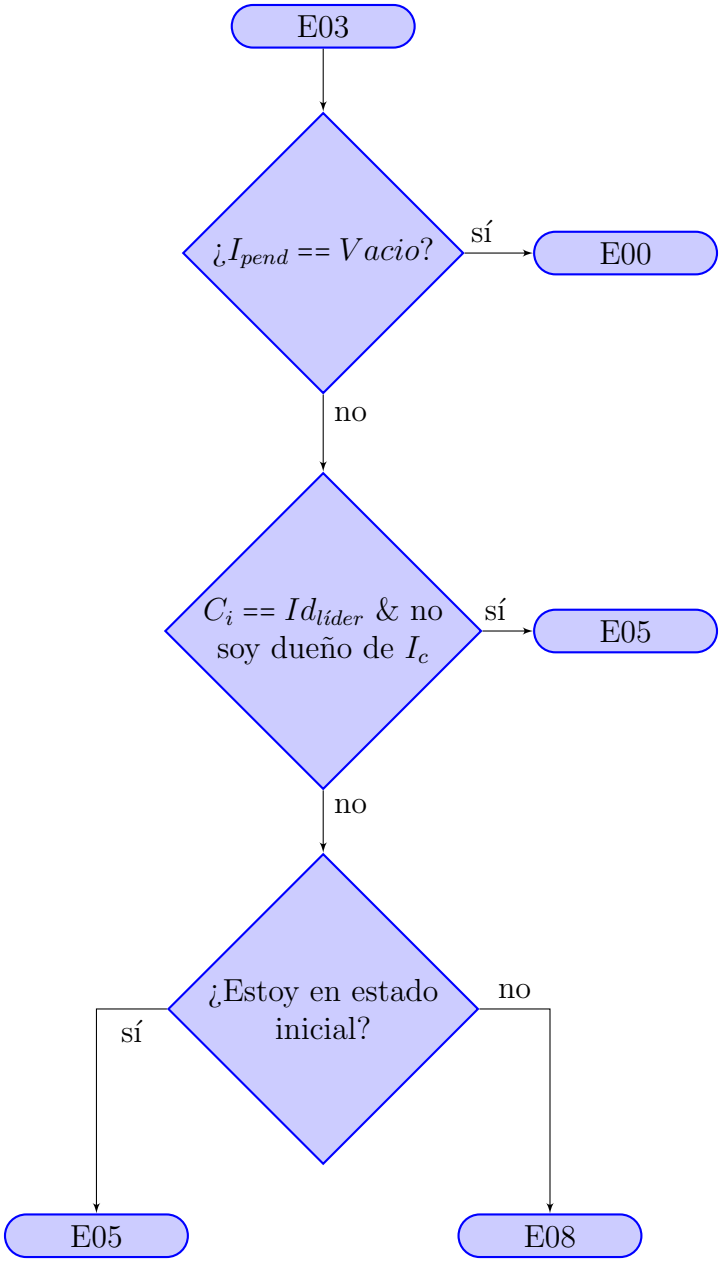
Tabla A.7: Descripción de los mensajes usados en Multipaxos: Durante el proceso de elección

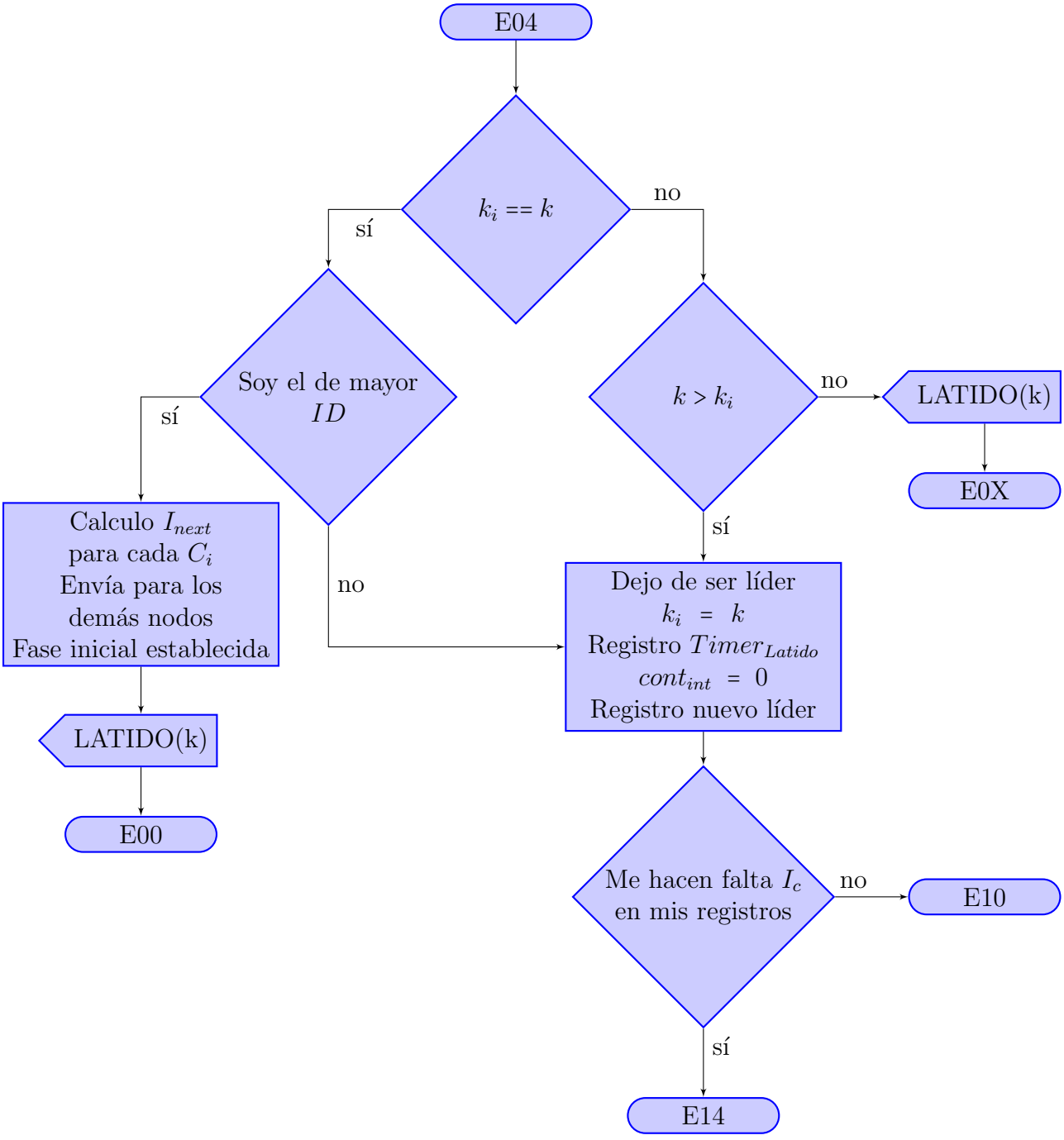
Nombre	**Descripción**
CANDIDATO	Notifica el mejor candidato para ser líder
ELECTO	Confirma el candidato que se convertirá en líder

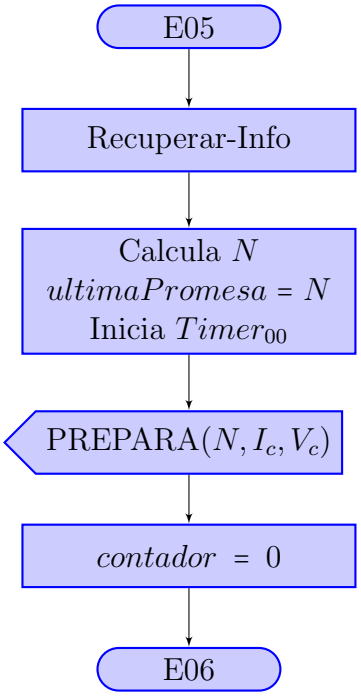

E00
SOLICITUD(V_c, I_c, C_i)
Registra I_c en I_{pend}
¿Espero I_c de C_i?
sí
no
¿Estoy en estado inicial?
sí
no
¿Tengo I_c en mis registros?
no
E01
sí
E05
E08
Quito I_c de I_{pend}
RESPUESTA(N, I_c, V_c)
E00

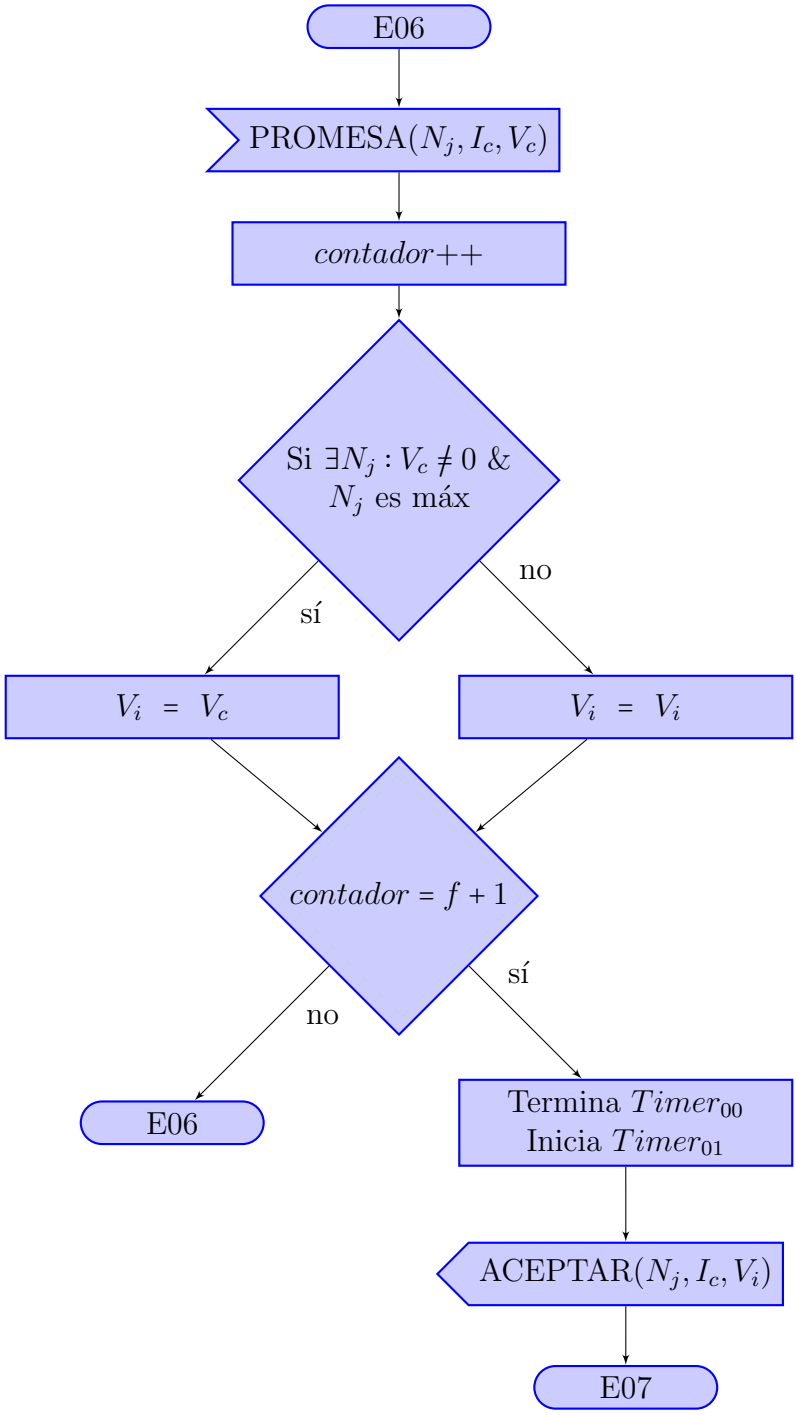

E01
¿Le corresponde I_c a C_i?
sí
no
¿Tengo registro de C_i?
sí
no
Quito I_c de I_{pend}
ESPERA
E00
E02
¿Estoy en estado inicial?
sí
no
E05
E08

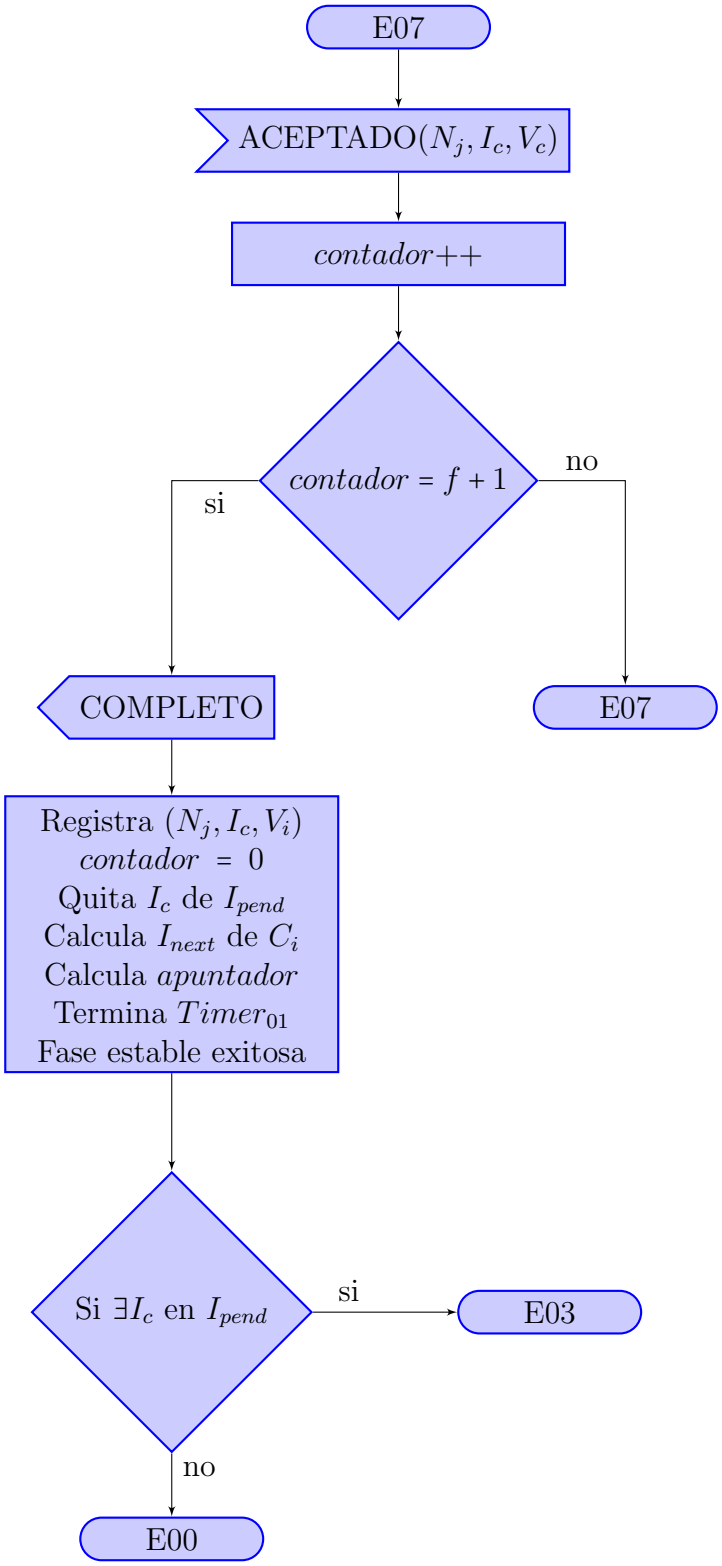

E02
Obtengo I_{last-c} de C_i
$I_x = I_{last-c} + P$
¿$I_x == I_c$?
sí
no
Registro
$(V_c, 0, I_c, C_i)$ en I_{pend}
Registro
$(V_f, 0, I_x, Id_{líder})$
en I_{pend}
E03
$I_x = I_x + P$

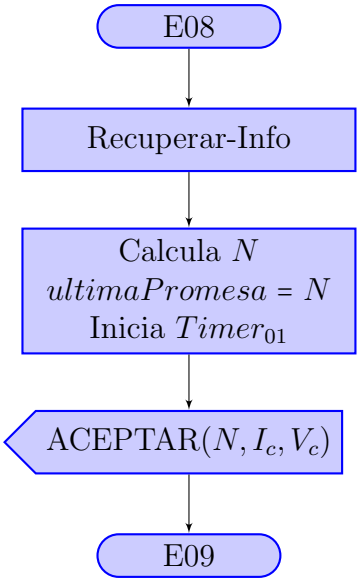

E03
¿I_{pend} == $Vacio$?
sí
E00
no
C_i == $Id_{líder}$ & no soy dueño de I_c
sí
E05
no
¿Estoy en estado inicial?
sí
no
E05
E08

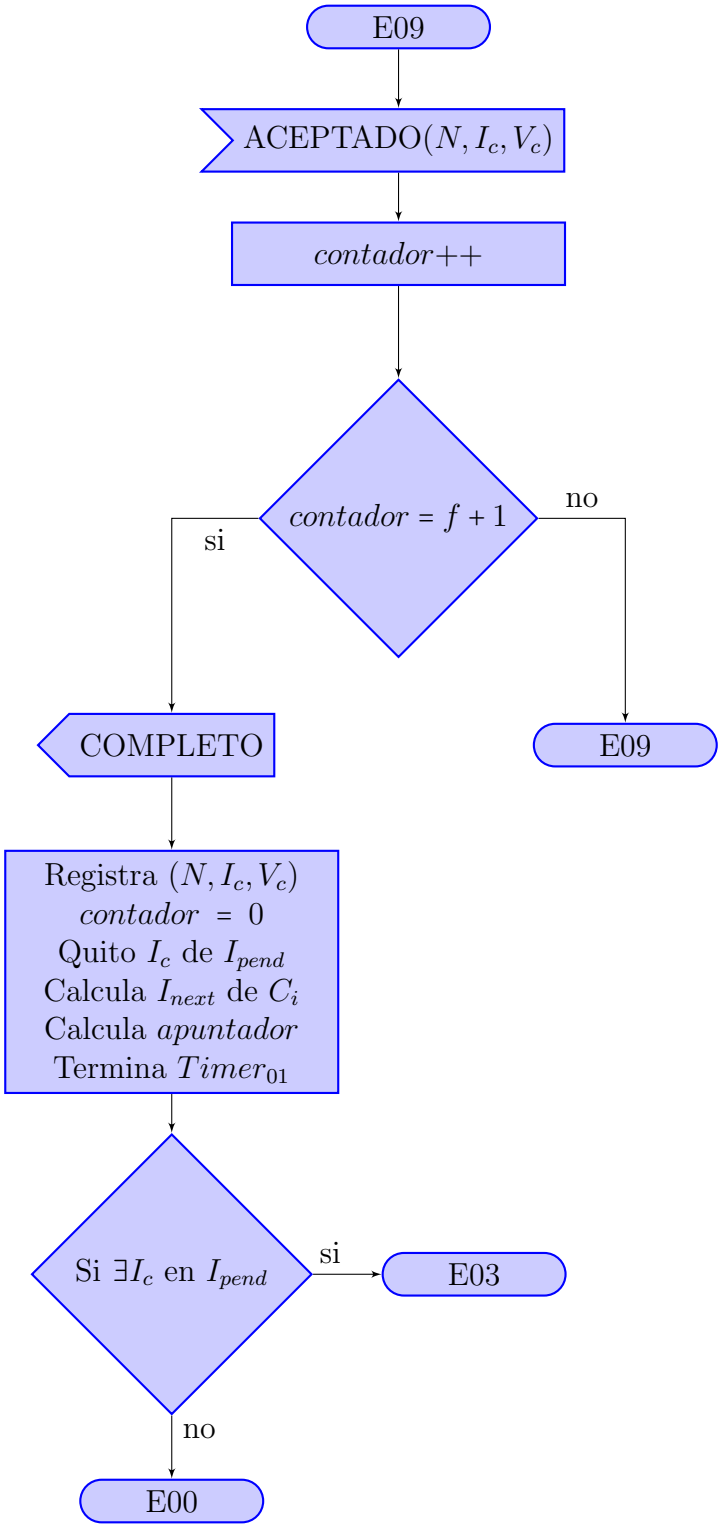

E04
$k_i == k$
sí
no
Soy el de mayor ID
$k > k_i$
no
LATIDO(k)
E0X
sí
sí
Calculo I_{next} para cada C_i Envía para los demás nodos Fase inicial establecida
no
Dejo de ser líder $k_i = k$ Registro $Timer_{Latido}$ $cont_{int} = 0$ Registro nuevo líder
LATIDO(k)
E00
Me hacen falta I_c en mis registros
no
E10
sí
E14

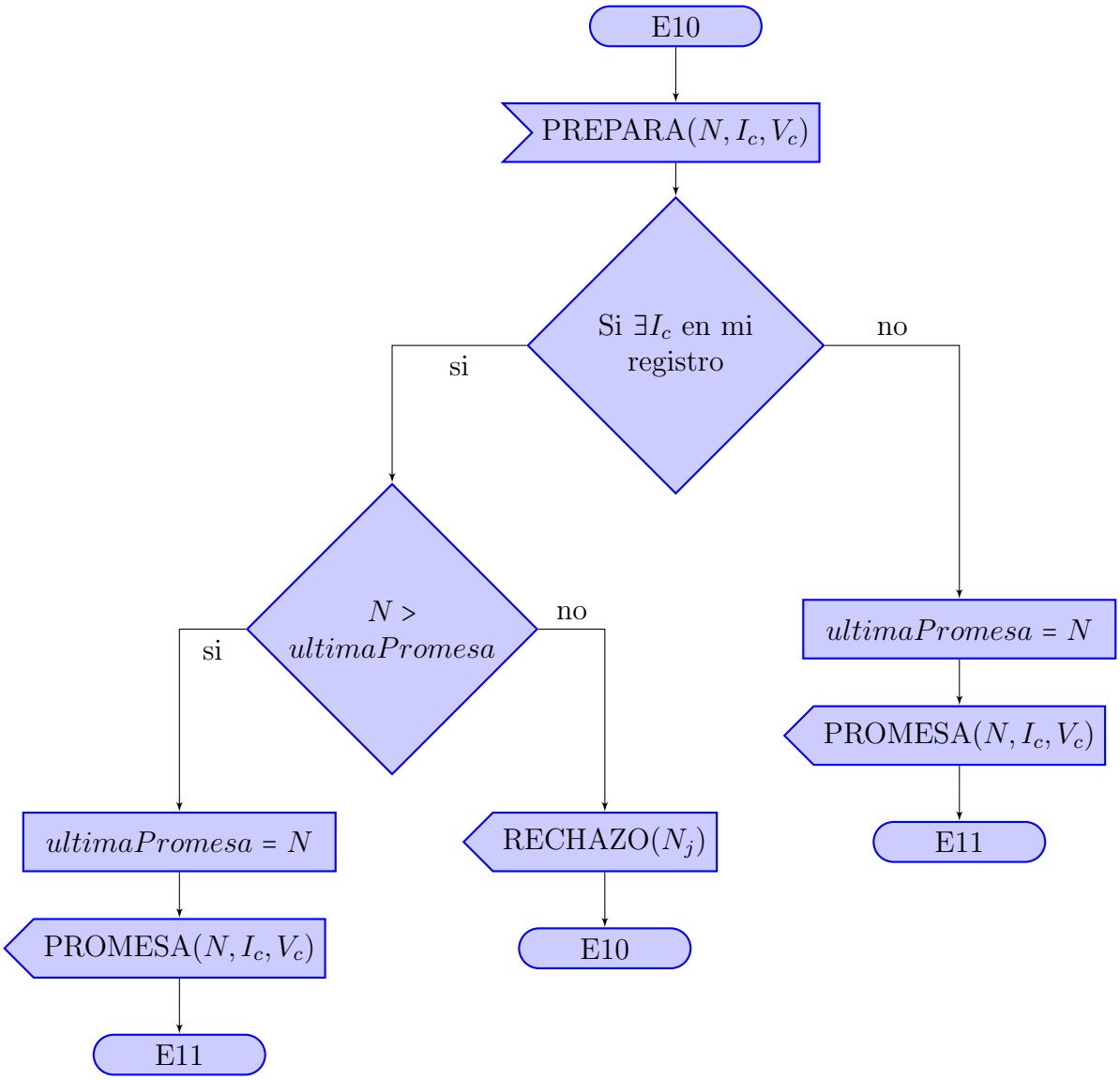

E05
Recuperar-Info
Calcula N
$ultimaPromesa = N$
Inicia $Timer_{00}$
PREPARA(N, I_c, V_c)
$contador = 0$
E06

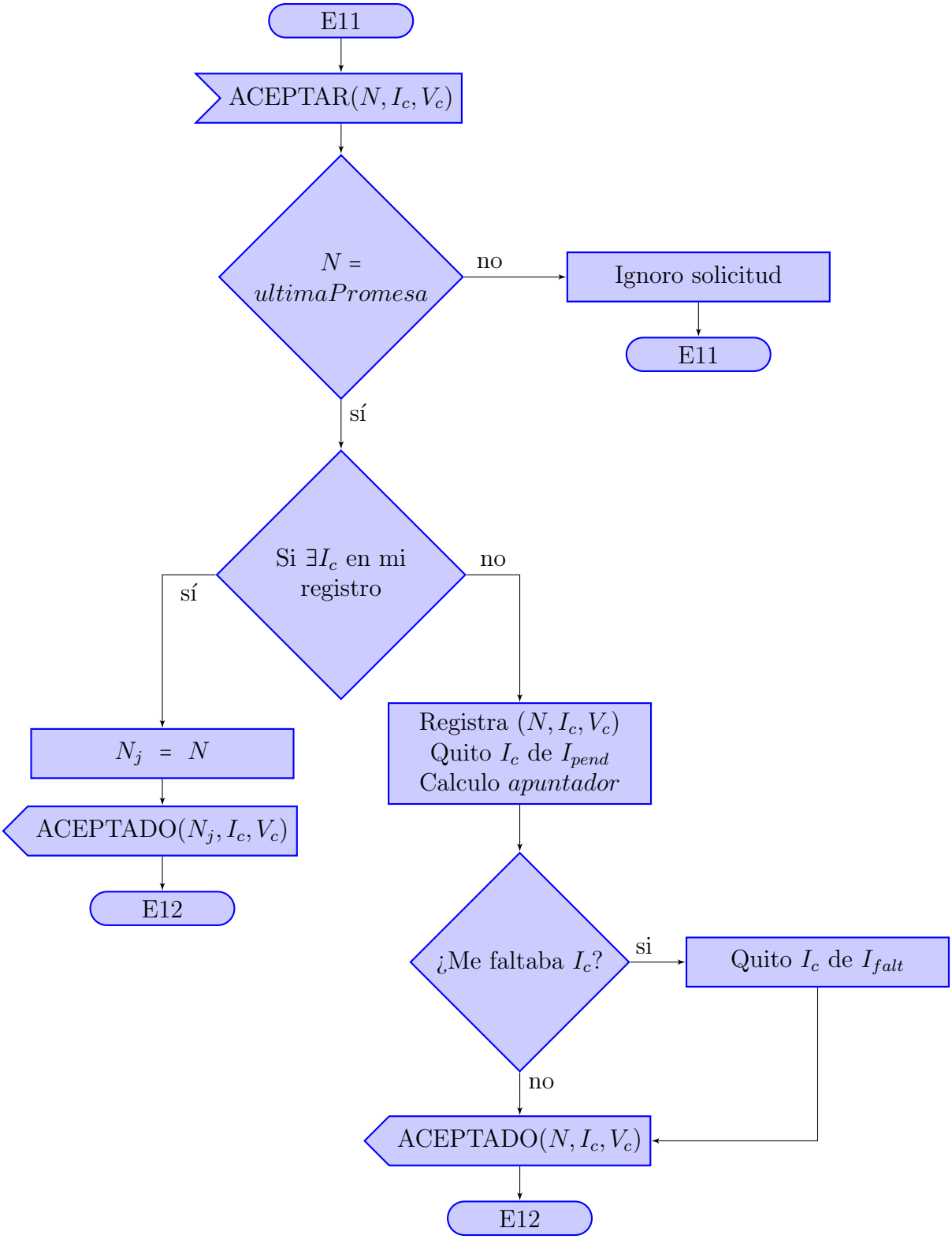

E06
PROMESA(N_j, I_c, V_c)
$contador$++
Si $\exists N_j : V_c \neq 0$ &
N_j es máx
sí
no
$V_i = V_c$
$V_i = V_i$
$contador = f + 1$
no
sí
E06
Termina $Timer_{00}$
Inicia $Timer_{01}$
ACEPTAR(N_j, I_c, V_i)
E07

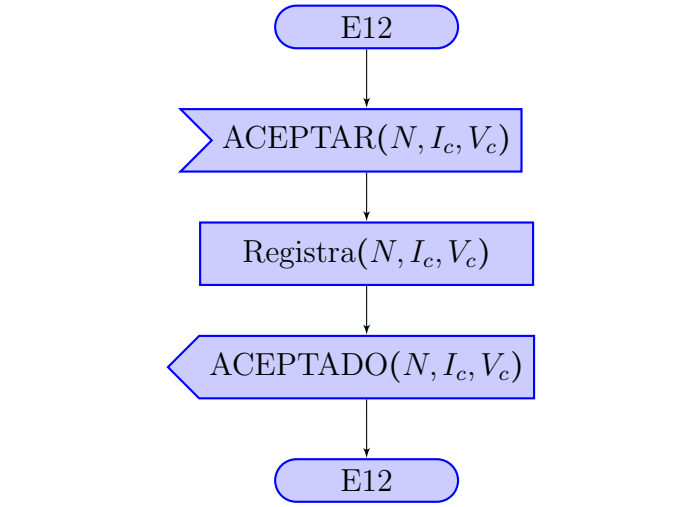

E07
ACEPTADO(N_j, I_c, V_c)
$contador$++
$contador = f + 1$
si
no
COMPLETO
E07
Registra (N_j, I_c, V_i)
$contador = 0$
Quita I_c de I_{pend}
Calcula I_{next} de C_i
Calcula $apuntador$
Termina $Timer_{01}$
Fase estable exitosa
Si $\exists I_c$ en I_{pend}
si
E03
no
E00

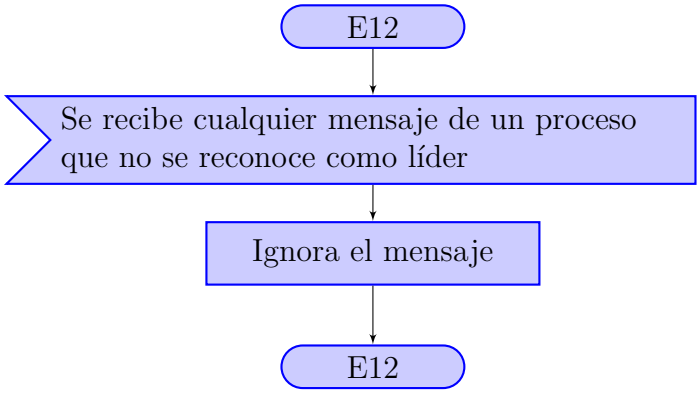

E08
Recuperar-Info
Calcula N
$ultimaPromesa = N$
Inicia $Timer_{01}$
ACEPTAR(N, I_c, V_c)
E09

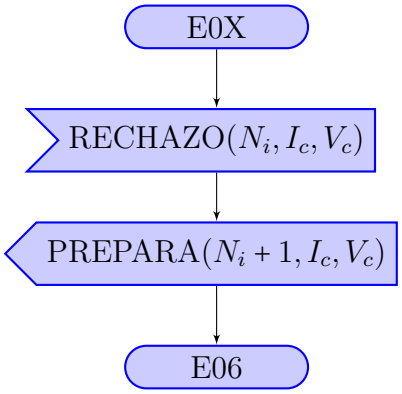

E09
ACEPTADO(N, I_c, V_c)
$contador{+}{+}$
$contador = f + 1$
si
no
COMPLETO
E09
Registra (N, I_c, V_c)
$contador = 0$
Quito I_c de I_{pend}
Calcula I_{next} de C_i
Calcula $apuntador$
Termina $Timer_{01}$
Si $\exists I_c$ en I_{pend}
si
E03
no
E00

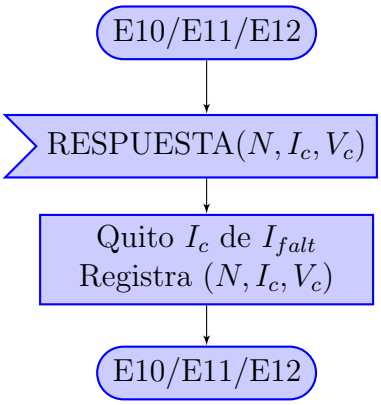

E10
PREPARA(N, I_c, V_c)
Si $\exists I_c$ en mi registro
si
no
$N >$ $ultimaPromesa$
si
no
$ultimaPromesa = N$
PROMESA(N, I_c, V_c)
E11
$ultimaPromesa = N$
RECHAZO(N_j)
PROMESA(N, I_c, V_c)
E10
E11

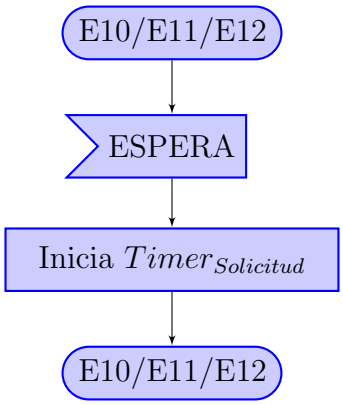

E11
ACEPTAR(N, I_c, V_c)
N = $ultimaPromesa$
no
Ignoro solicitud
E11
sí
Si $\exists I_c$ en mi registro
no
sí
$N_j = N$
ACEPTADO(N_j, I_c, V_c)
E12
Registra (N, I_c, V_c)
Quito I_c de I_{pend}
Calculo $apuntador$
¿Me faltaba I_c?
si
Quito I_c de I_{falt}
no
ACEPTADO(N, I_c, V_c)
E12

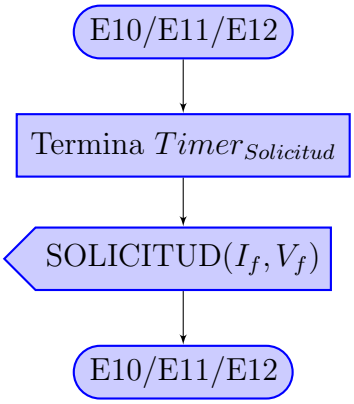

E12
ACEPTAR(N, I_c, V_c)
Registra(N, I_c, V_c)
ACEPTADO(N, I_c, V_c)
E12

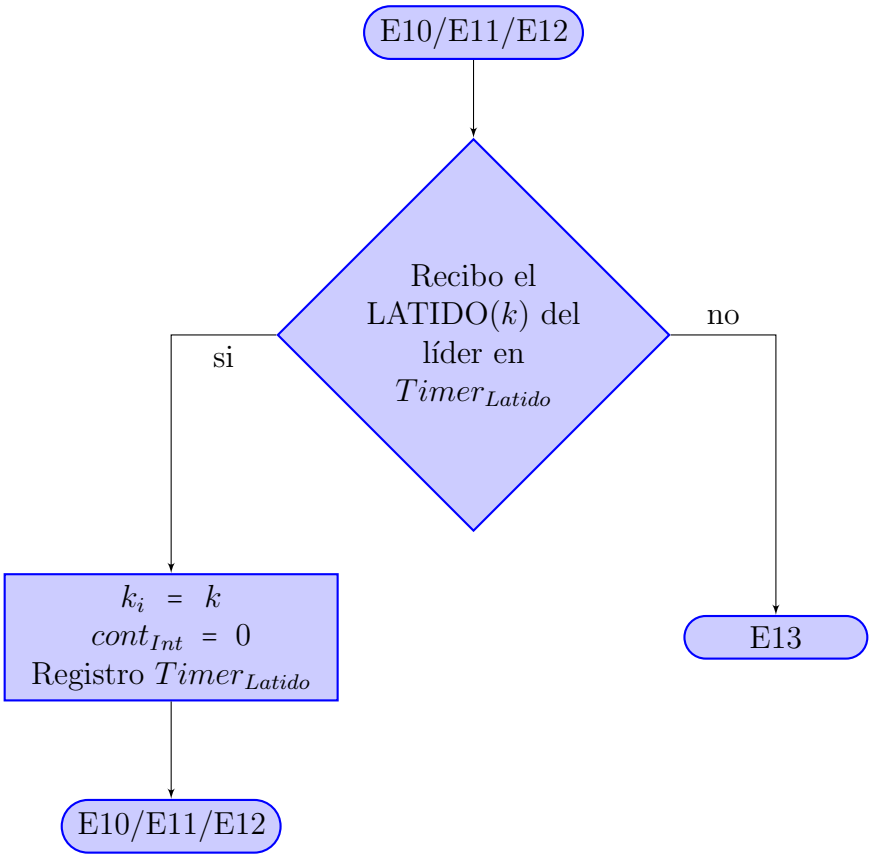

E12
Se recibe cualquier mensaje de un proceso que no se reconoce como líder
Ignora el mensaje
E12

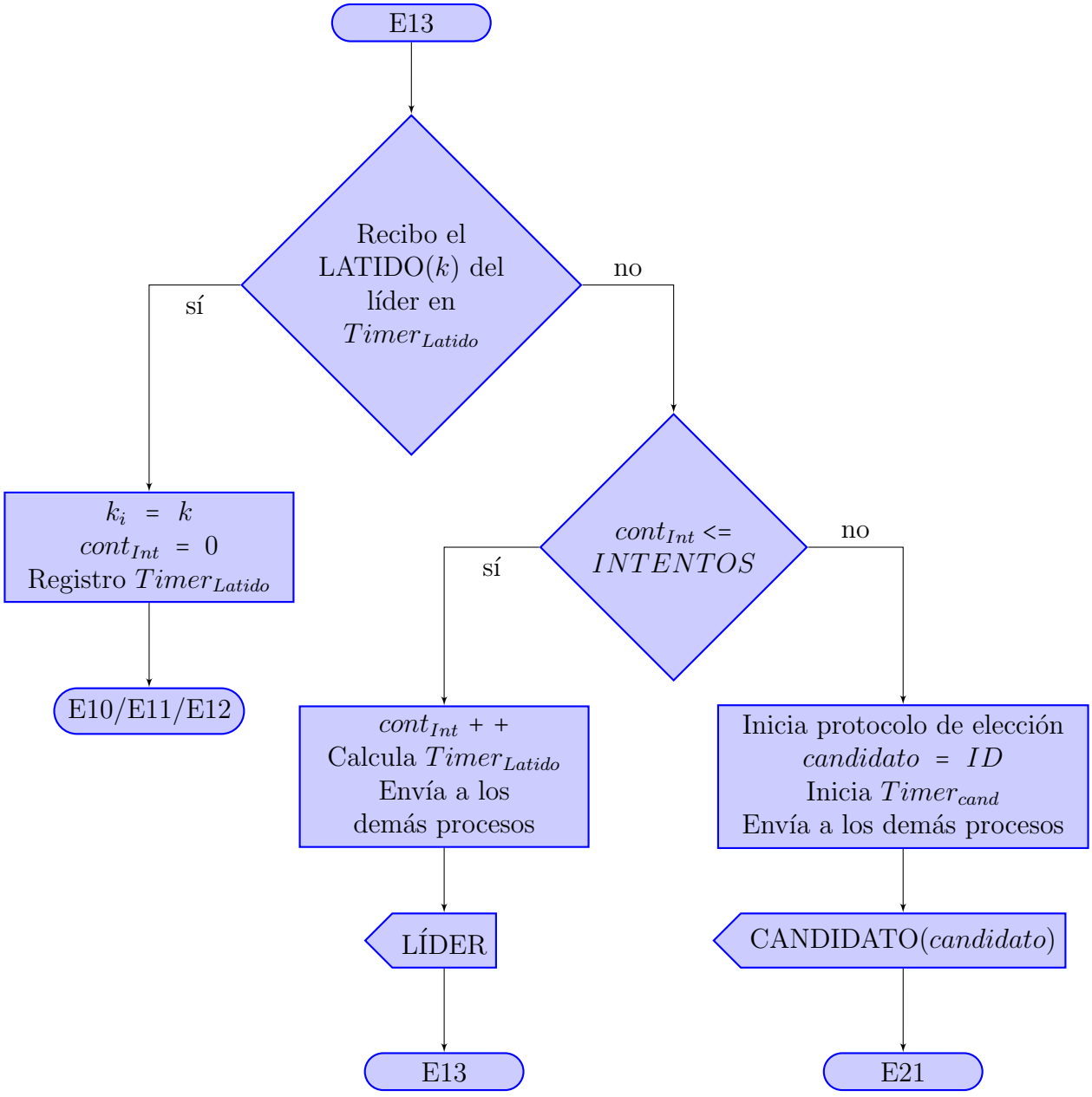

E0X
RECHAZO(N_i, I_c, V_c)
PREPARA($N_i + 1, I_c, V_c$)
E06

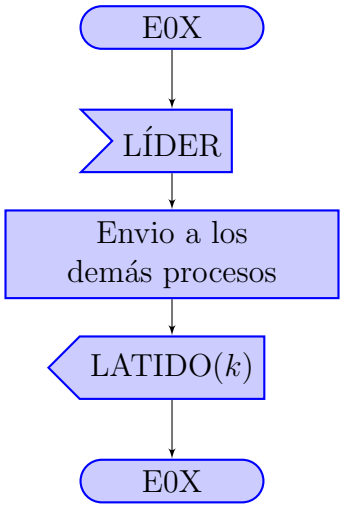

E10/E11/E12
RESPUESTA(N, I_c, V_c)
Quito I_c de I_{falt}
Registra (N, I_c, V_c)
E10/E11/E12

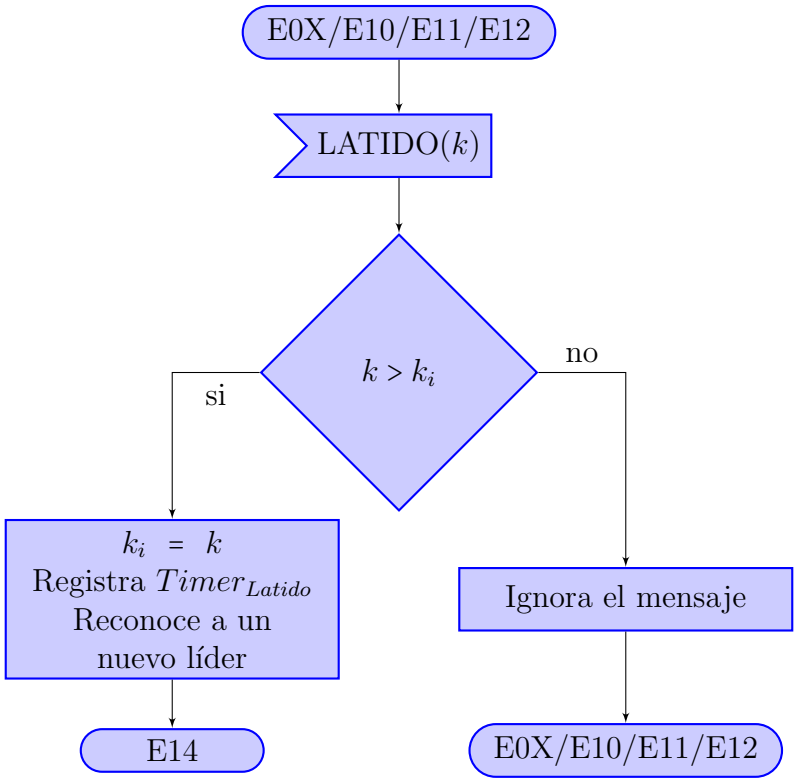

E10/E11/E12
ESPERA
Inicia $Timer_{Solicitud}$
E10/E11/E12

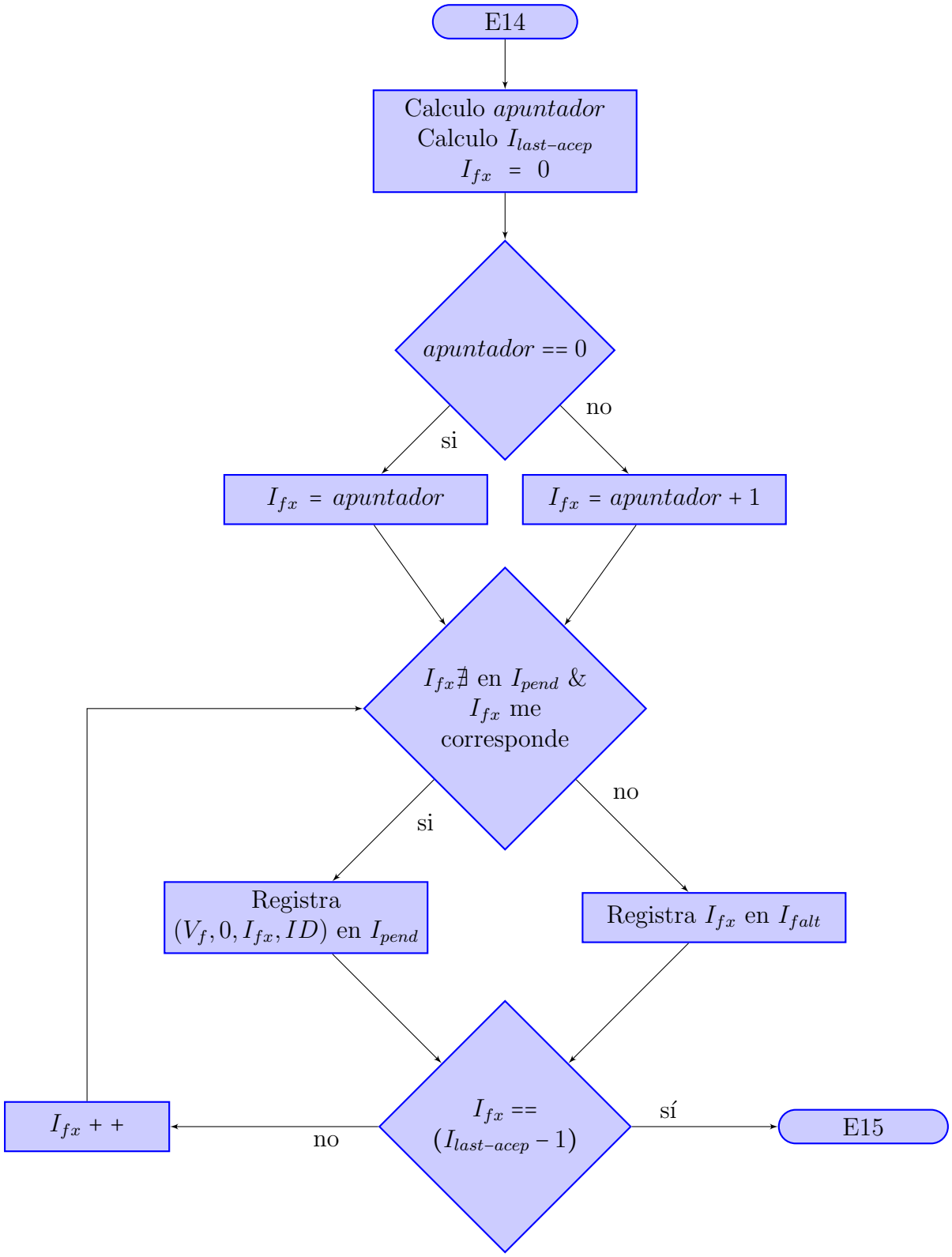

E10/E11/E12
Termina $Timer_{Solicitud}$
SOLICITUD(I_f, V_f)
E10/E11/E12

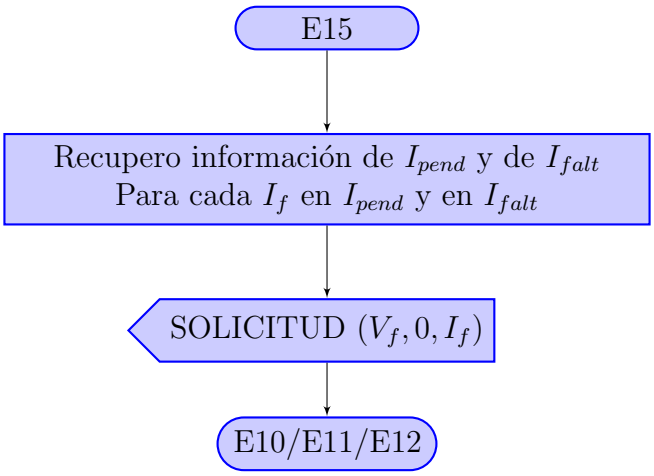

E10/E11/E12
Recibo el LATIDO(k) del líder en $Timer_{Latido}$
si
no
$k_i = k$
$cont_{Int} = 0$
Registro $Timer_{Latido}$
E13
E10/E11/E12

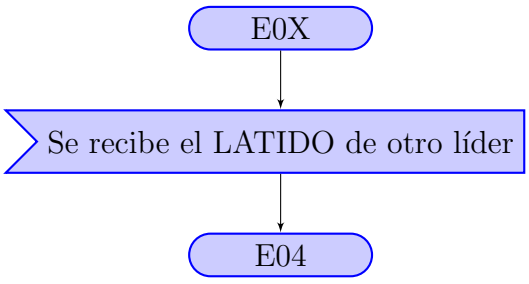

E13
Recibo el LATIDO(k) del líder en $Timer_{Latido}$
sí
no
$k_i = k$
$cont_{Int} = 0$
Registro $Timer_{Latido}$
E10/E11/E12
$cont_{Int}$ <= $INTENTOS$
sí
no
$cont_{Int}$ + +
Calcula $Timer_{Latido}$
Envía a los demás procesos
Inicia protocolo de elección
$candidato = ID$
Inicia $Timer_{cand}$
Envía a los demás procesos
LÍDER
CANDIDATO($candidato$)
E13
E21

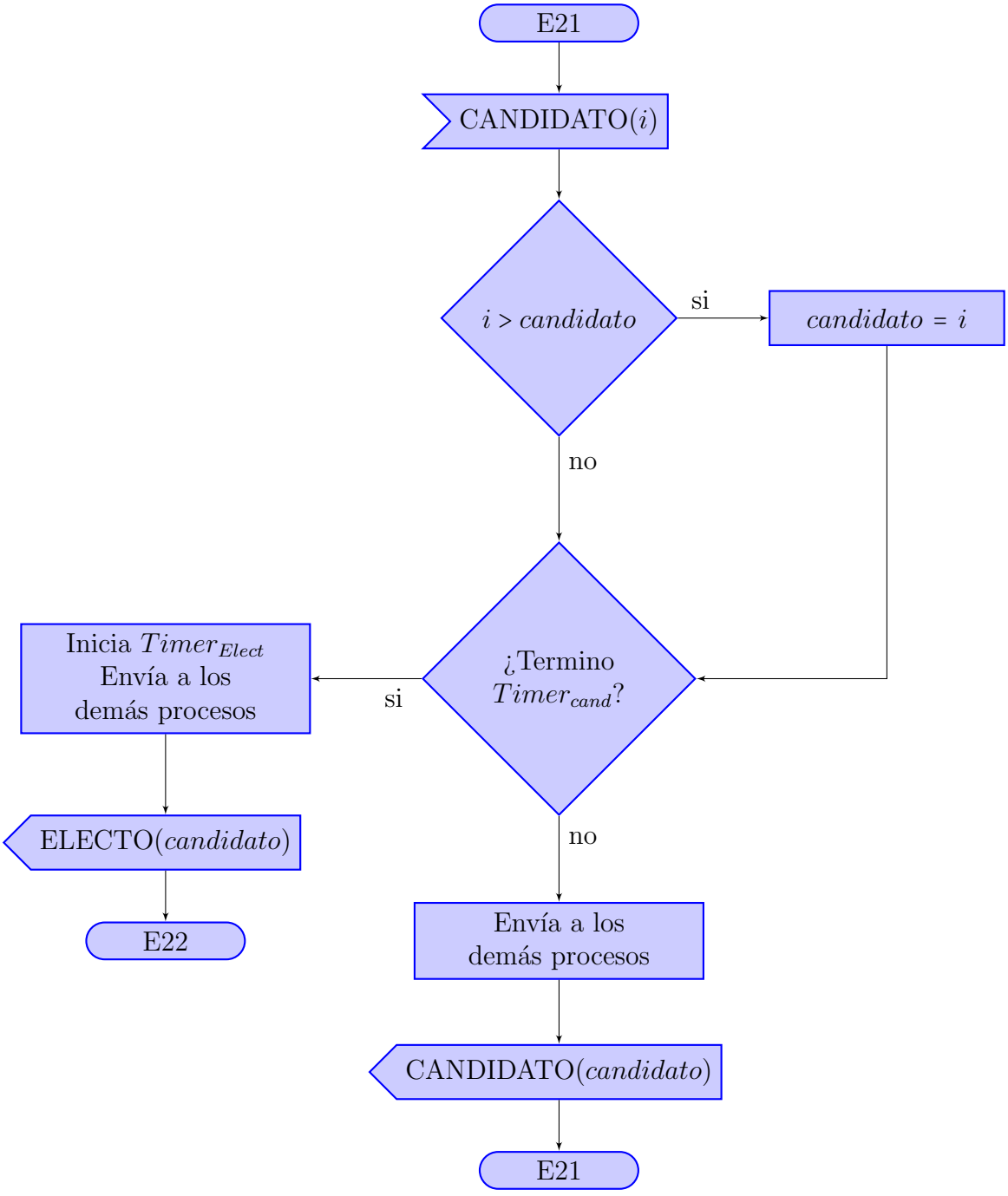

E0X
LÍDER
Envio a los
demás procesos
LATIDO(k)
E0X

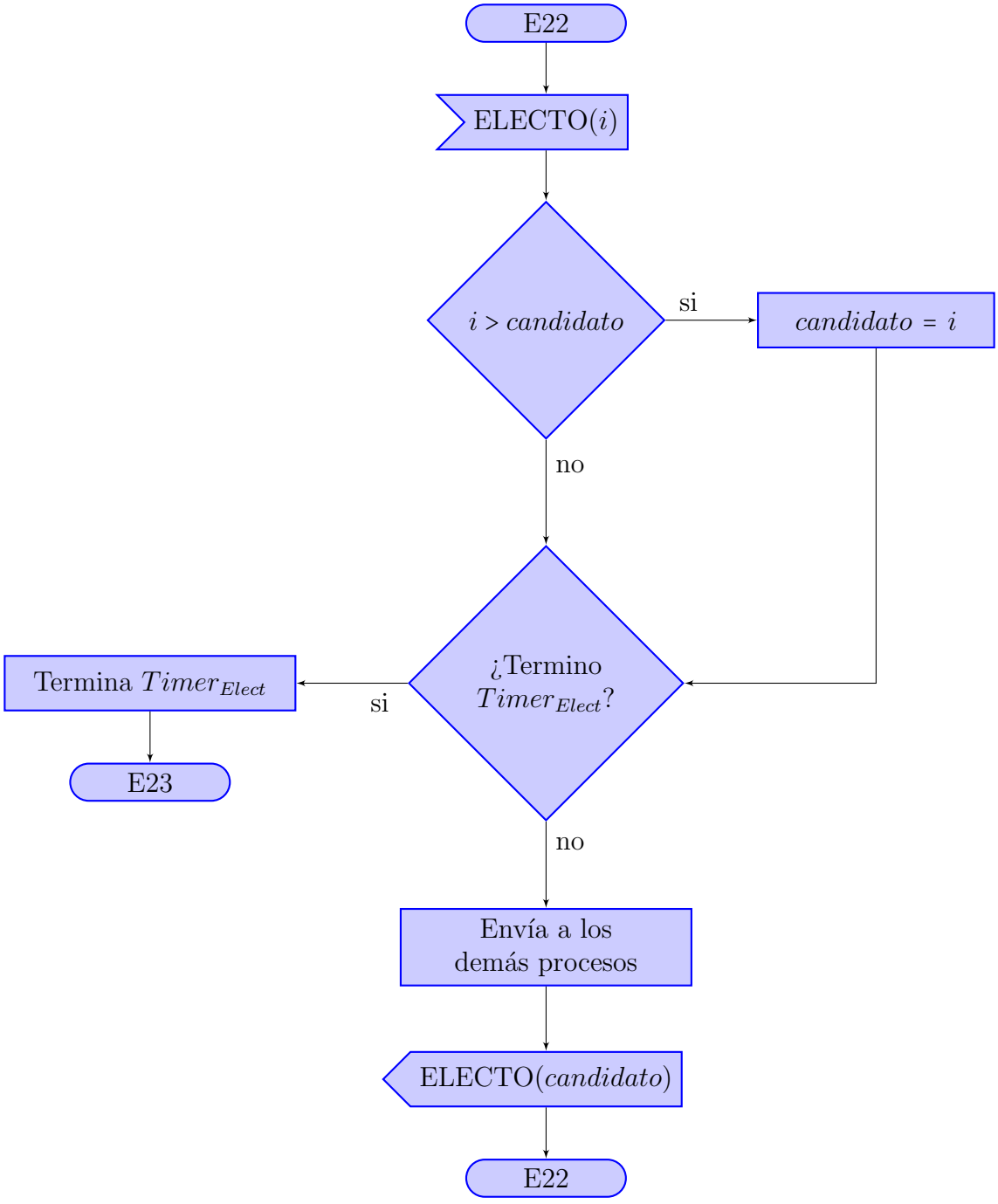

E0X/E10/E11/E12
LATIDO(k)
$k > k_i$
si
no
$k_i = k$
Registra $Timer_{Latido}$
Reconoce a un
nuevo líder
E14
Ignora el mensaje
E0X/E10/E11/E12

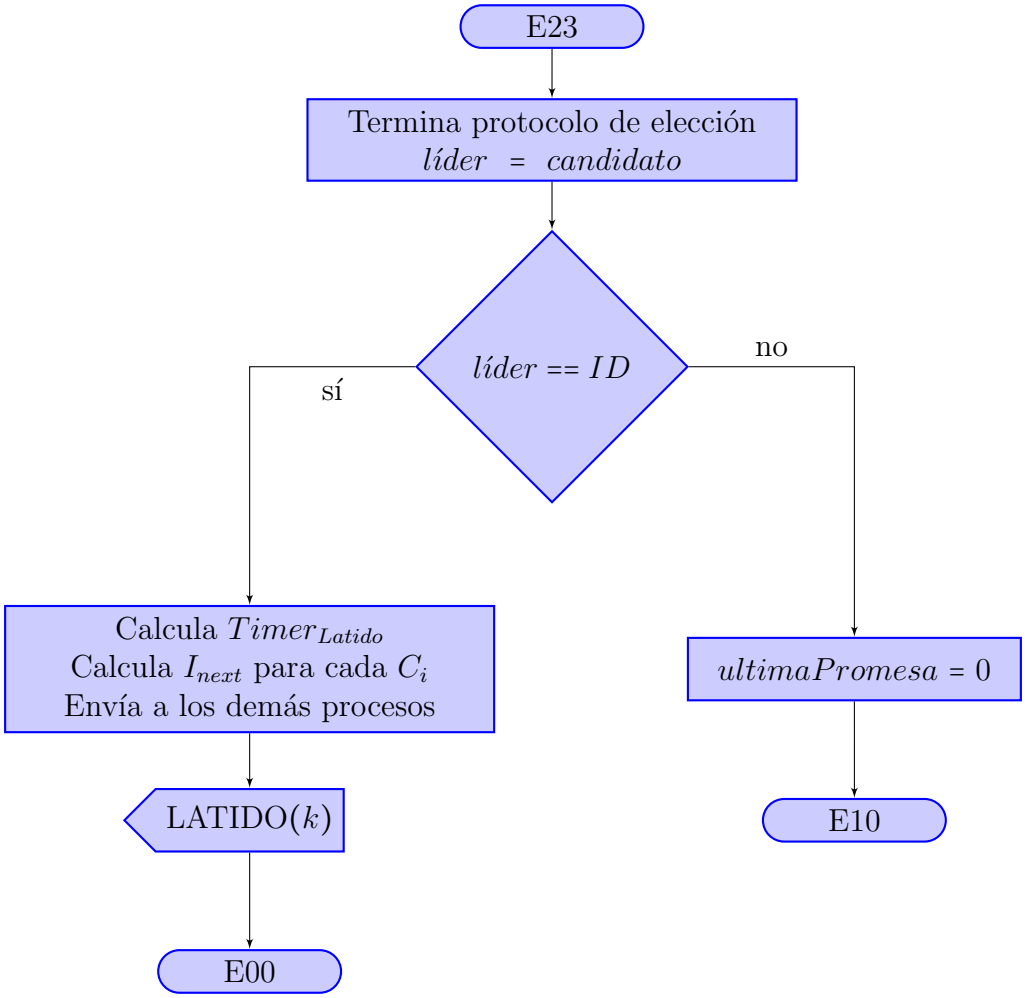

E14
Calculo *apuntador*
Calculo $I_{last-acep}$
$I_{fx} = 0$
apuntador == 0
si
no
I_{fx} = *apuntador*
I_{fx} = *apuntador* + 1
$I_{fx} \nexists$ en I_{pend} &
I_{fx} me
corresponde
si
no
Registra
$(V_f, 0, I_{fx}, ID)$ en I_{pend}
Registra I_{fx} en I_{falt}
I_{fx} ==
$(I_{last-acep} - 1)$
sí
no
I_{fx} + +
E15


E15
Recupero información de I_{pend} y de I_{falt}
Para cada I_f en I_{pend} y en I_{falt}
SOLICITUD $(V_f, 0, I_f)$
E10/E11/E12


E0X
Se recibe el LATIDO de otro líder
E04


E21
CANDIDATO(i)
$i > candidato$
si
$candidato = i$
no
¿Termino
$Timer_{cand}$?
si
Inicia $Timer_{Elect}$
Envía a los
demás procesos
ELECTO($candidato$)
E22
no
Envía a los
demás procesos
CANDIDATO($candidato$)
E21


E22
ELECTO(i)
$i > candidato$
si
$candidato = i$
no
¿Termino $Timer_{Elect}$?
si
Termina $Timer_{Elect}$
E23
no
Envía a los demás procesos
ELECTO($candidato$)
E22


E23
Termina protocolo de elección
líder = candidato
líder == ID
sí
no
Calcula $Timer_{Latido}$
Calcula I_{next} para cada C_i
Envía a los demás procesos
LATIDO(k)
E00
ultimaPromesa = 0
E10

REFERENCIAS

[1] J. L. González, J. Pérez-Carretero, V. Sosa-Sosa, J. F. Rodriguez-Cardoso, and R. Marcelín-Jiménez. An approach for constructing private storage services as a unified fault-tolerant system. J. Syst. Softw., 86(7):1907–1922, July 2013.

[2] P. N. Yianilos and S. Sobti. The evolving field of distributed storage. Internet Computing, IEEE, 5(5):35–39, Sep 2001.

[3] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. J. ACM, 32(2):374–382, April 1985.

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

[5] H. Attiya and J. Welch. Distributed Computing: Fundamentals, Simulations and Advanced Topics. John Wiley & Sons, 2004.

[6] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. J. ACM, 43(2):225–267, March 1996.

[7] L. Lamport. The part-time parliament. ACM Trans. Comput. Syst., 16(2):133–169, May 1998.

[8] Sitio oficial de python. `http://www.python.org`.

[9] L. Lamport. Paxos made simple. ACM SIGACT News, 32(4):18–25, 2001.

[10] R. De Prisco, B. W. Lampson, and N. A. Lynch. Revisiting the paxos algorithm. Theor. Comput. Sci., 243(1-2):35–91, 2000.

[11] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Deconstructing paxos. SIGACT News, 34(1):47–67, March 2003.

[12] B. Lampson. The ABCD's of paxos. In PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing, New York, NY, USA, 2001. ACM Press.

[13] M. Castro and B. Liskov. Practical byzantine fault tolerance. In Proceedings of the Third Symposium on Operating Systems Design and Implementation, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[14] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. ACM Trans. Program. Lang. Syst., 4(3):382–401, July 1982.

[15] E. Gafni and L. Lamport. Disk paxos. Distrib. Comput., 16(1):1–20, February 2003.

[16] H.C. Li, A. Clement, A.S. Aiyer, and L. Alvisi. The paxos register. In Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on, pages 114–126, Oct 2007.

[17] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[18] S. Ghemawat, H. Gobioff, and S. T. Leung. The google file system. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[19] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[20] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[21] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS VII, pages 84–92, New York, NY, USA, 1996. ACM.

[22] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6, OSDI'04, pages 8–8, Berkeley, CA, USA, 2004. USENIX Association.

[23] J. MacCormick, C. A. Thekkath, M. Jager, K. Roomp, L. Zhou, and R. Peterson. Niobe: A practical replication protocol. Trans. Storage, 3(4):1:1–1:43, February 2008.

[24] J. Kirsch and Y. Amir. Paxos for system builders: An overview. In Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware, LADIS '08, pages 3:1–3:6, New York, NY, USA, 2008. ACM.

[25] B. Li, Y. He, and K. Xu. Distributed metadata management scheme in cloud computing. In Proceedings of 2011 6th International Conference on Pervasive Computing and Applications(ICPCA), pages 32–38, Port Elizabeth, South Africa, 2011.

[26] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[27] D. Stamatakis, N. Tsikoudis, O. Smyrnaki, and K. Magoutis. Scalability of replicated metadata services in distributed file systems. In KarlMichael Göschka and Seif Haridi, editors, Distributed Applications and Interoperable Systems, volume 7272 of Lecture Notes in Computer Science, pages 31–44. Springer Berlin Heidelberg, 2012.

[28] S. Mu, K. Chen, Y. Wu, and W. Zheng. When paxos meets erasure code: Reduce network and storage cost in state machine replication. In Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing, HPDC '14, pages 61–72, New York, NY, USA, 2014. ACM.

[29] W.K. Lin, D.M. Chiu, and Y.B. Lee. Erasure code replication revisited. In Peer-to-Peer Computing, 2004. Proceedings. Proceedings. Fourth International Conference on, pages 90–97, Aug 2004.

[30] L. Rizzo. Effective erasure codes for reliable computer communication protocols. SIGCOMM Comput. Commun. Rev., 27(2):24–36, April 1997.

[31] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[32] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.

[33] B. W. Lampson and H. E. Sturgis. Crash recovery in a distributed data storage system, 1979.

[34] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the harp file system. In Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles, SOSP '91, pages 226–238, New York, NY, USA, 1991. ACM.

[35] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks, DSN '11, pages 245–256, Washington, DC, USA, 2011. IEEE Computer Society.

[36] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[37] R. Van-Renesse, N. Schiper, and F. B. Schneider. Vive la différence: Paxos vs. viewstamped replication vs. zab. CoRR, abs/1309.5671, 2013.

[38] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In Proceedings of the 2nd International Conference on Software Engineering, ICSE '76, pages 562–570, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[39] L. Lamport. Time, clocks, and the ordering of events in a distributed system. Commun. ACM, 21(7):558–565, July 1978.

[40] Z. Chen, J. Xiong, and D. Meng. Replication-based highly available metadata management for cluster file systems. In Cluster Computing (CLUSTER), 2010 IEEE International Conference on, pages 292–301, Sept 2010.

[41] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[42] I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults - a tutorial. In Rogério de Lemos, Taisy Silva Weber, and João Batista Camargo Jr., editors, LADC, volume 2847 of Lecture Notes in Computer Science, pages 366–368. Springer, 2003.

ÍNDICE ALFABÉTICO