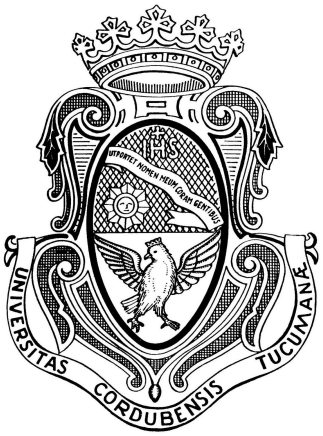
Facultad de Matemática, Astronomía, Física y Computación

Universidad Nacional de Córdoba



Verificación formal de protocolos distribuidos

TRABAJO ESPECIAL DE LICENCIATURA EN
CIENCIAS DE LA COMPUTACIÓN

Alejandro Naser Pastoriza

Dirigido por:

Dr. Alexey Gotsman

Colaborador:

Dr. Pedro D'Argenio

Resumen

En esta tesis probamos la correctitud de tres protocolos distribuidos. Primero, el algoritmo Single Decree Paxos que resuelve el problema de llegar a un acuerdo, o alcanzar consenso, entre un conjunto de procesos. Segundo, una optimización del protocolo de broadcast atómico de ZooKeeper que implementa un esquema primary-backup en el cual un proceso primario ejecuta las operaciones del cliente y utliza Zab para propagar los correspondientes cambios incrementales de estado a los procesos backup. Dicha optimización desarrollada por Alexey Gotsman, Joe Izraelevitz y Gregory Chockler, reduce la transferencia de estado durante la elección del líder. Por último, una variante del protocolo Vertical Paxos en las líneas de *Reconfigurable Atomic Transaction Commit*, Bravo & Gotsman, PODC'19, una clase de algoritmo Paxos en el cual reconfiguraciones pueden ocurrir durante el proceso de alcanzar acuerdo. Posteriormente, formalizamos la prueba de Single Decree Paxos en Dafny, un lenguaje de programación verifier-friendly basado en lógica de Floyd-Hoare que automatiza la verificación a través del SMT solver Z3.

Abstract

In this thesis we prove the correctness of three distributed protocols. First, the Single Decree Paxos algorithm which solves the problem of reaching agreement, or consensus, among a set of processes. Secondly, an optimization of the ZooKeeper atomic broadcast protocol which implements a primary-backup scheme in which a primary process executes client operations and uses Zab to propagate the corresponding incremental state changes to the backup processes. Such optimization developed by Alexey Gotsman, Joe Izraelevitz and Gregory Chockler, reduces state transfer during the leader election. Lastly, a variant of Vertical Paxos along the lines of *Reconfigurable Atomic Transaction Commit*, Bravo & Gotsman, PODC'19, a class of Paxos algorithm in which reconfigurations can occur in the middle of reaching agreement. Finally, we formalize the proof of Single Decree Paxos in Dafny, a verifier-friendly programming language based on Floyd-Hoare logic that automates verification via the Z3 SMT solver.

Agradecimientos

A Alexey Gotsman, por su confianza, y por ser una fuente de inspiración cada día.

A mis amigos, por más motivos de los que puedo enumerar, en especial a Mariano Mateuci, Luciano Cingolani, Giovanni Rescia, Ignacio Vidarte, Luis Ferroni, Diego Sulca, Alejandro Gadea, Felix Cuello, Nicolas Gomez y Nicolás Bella.

Un especial agradecimiento a Fedor Ryabinin por leer un manuscrito preliminar de este trabajo, cualquier error u omisión es también su culpa.

A Rosita Vitriu, Mary Almaraz y Mario Castillo, por introducirme al mundo de las matemáticas, y sin quiénes no estaría donde estoy hoy.

A Florin Dinu, por su invaluable ayuda.

A la familia Ledesma López, por ser también mi familia.

A Adriana Pastoriza, por ser una segunda madre.

A mi familia en general, en especial a Alfredo Naser, Teresita Pastoriza y Julián Naser.

A mi esposa, Noelia Sofía Ledesma López, por su amor incondicional.

A ustedes va dedicado mi trabajo.

Para Noelia Sofía.

Índice general

Introducción

Múltiples sistemas modernos comprenden un grupo de procesos. Un proceso abstrae una unidad de cómputo tal como una computadora física o virtual, un procesador, o un hilo de ejecución en un sistema concurrente. Un objetivo central en el desarrollo y la implementación de tales sistemas es lograr que los procesos colaboren en una tarea común.

El entorno, que varía desde una única unidad, a un centro de datos, e incluso a un sistema de escala planetaria, plantea una serie de desafíos: cómo idear un mecanismo que logre una forma robusta de cooperación a pesar de fallos en la red y en los procesos, e inclusive, en presencia de ataques adversarios. El desafío principal en sistemas distribuidos es precisamente construir algoritmos que equipen a cada proceso que permanece operativo con suficiente información, de manera tal que continúe cooperando correctamente en la tarea común en presencia de fallos parciales.

A fin de simplificar el diseño y la implementación de sistemas distribuidos, patrones recurrentes de cooperación en conjunto con sus complejidades inherentes, pueden ser encapsulados en abstracciones específicas sobre las cuales posteriormente se pueda construir sistemas más ricos. En esta tesis implementamos dos de estas abstracciones: consenso y broadcast atómico.

Los sistemas distribuidos tolerantes a fallos son ubicuos en nuestra vida diaria. Por lo tanto, errores en tales sistemas conllevan efectos dramáticos. A pesar de su amplio uso y la madurez del campo, estos sistemas continúan siendo conocidos por albergar errores sutiles y ser viciosamente difíciles de diseñar e implementar correctamente. En consecuencia, verificar safety de sistemas distribuidos es un esfuerzo continuo de investigación.

Fallos y demoras en la red, variaciones en los tiempos de ejecución, la pérdida, reordenamiento o duplicación de mensajes, los fallos en los procesos, los dominios no acotados, los parámetros desconocidos por el sistema, el número posiblemente infinito de schedules, y en general, el espacio de estados combinatoriamente grande, pueden disparar comportamientos que no fueron intencionados ni anticipados por los diseñadores y desarrolladores. Las técnicas de testing, en el mejor de los casos, rasguñan la superficie de todos los posibles comportamientos, model checking es incompleto y no escala, y en definitiva, las técnicas automáticas son limitadas por la indecidibilidad del problema subyacente. Un enfoque más apropiado, entonces, es verificar formalmente las propiedades deseadas a través de pruebas matemáticas.

En esta tesis probamos la correctitud de tres protocolos distribuidos que implementan las abstracciones anteriormente mencionadas. Primero, el algoritmo Single

Decree Paxos que resuelve el problema de alcanzar acuerdo, o consenso, entre un conjunto de procesos. Segundo, una optimización del protocolo de broadcast atómico de ZooKeeper que implementa un esquema primary-backup en el cual un proceso primario ejecuta las operaciones del cliente y utiliza Zab para propagar los correspondientes cambios incrementales de estado a los procesos backup. Dicha optimización desarrollada por Alexey Gotsman, Joe Izraelevitz y Gregory Chockler, reduce la transferencia de estado durante la elección del líder. Por último, una variante del protocolo Vertical Paxos en las líneas de *Reconfigurable Atomic Transaction Commit*, Bravo & Gotsman, PODC'19, una clase de algoritmo Paxos en el cual reconfiguraciones pueden ocurrir durante el proceso de alcanzar acuerdo.

Estas pruebas son igualmente complejas y pueden también albergar errores sutiles. Por lo tanto, para eliminar categóricamente errores de nuestros algoritmos, aspiramos a construir pruebas de verificación automatizadas chequeadas por máquina de sus propiedades de safety. Como un primer paso, hacemos esto para una de las pruebas en esta tesis, en particular, la prueba de correctitud de Single Decree Paxos, estableciendo de esta manera las bases para en el futuro formalizar las pruebas restantes.

Utilizamos Dafny, un lenguaje de programación verifier-friendly basado en lógica de Floyd-Hoare que automatiza la verificación vía el SMT solver Z3. Esto permite completar muchas pruebas de bajo nivel automáticamente. Sin embargo, múltiples clases de proposiciones son en general no decidibles, por lo que Z3 utiliza heurísticas. A fin de guiar las heurísticas del verificador hacia una prueba, proveemos invariantes que implican nuestras propiedades de safety, y delegamos en Dafny probar que son inductivos.

En resumen, esta tesis realiza las siguientes contribuciones:

1 El capítulo tres presenta un protocolo novedoso cuyo procedimiento de recuperación a fallos provee garantías bien definidas en términos de rendimiento y resistencia a fallos, y que podría ser de interés independiente.
2 El capítulo cuatro presenta la adaptación de un protocolo desarrollado inicialmente para *Transaction Certification Services* a un protocolo de broadcast atómico con un enfoque vertical, lo cual disminuye el número de réplicas necesarias para garantizar durabilidad en ciertos contextos.
3 Demostraciones matemáticas de que cada protocolo es una implementación correcta de su correspondiente abstracción.
4 El capítulo cinco presenta una verificación formal de Single Decree Paxos en Dafny.

Capítulo 1

Preliminares

1.1. Abstracciones

Para razonar sobre sistemas distribuidos debemos primero abstraer el sistema físico subyacente: identificar los componentes relevantes, describir sus propiedades centrales en una manera abstracta, y caracterizar la forma en que interactúan. Básicamente, el objetivo es abstraer todos aquellos detalles que no contribuyen hacia la comprensión fundamental de los algoritmos.

Una unidad capaz de realizar cómputo en un sistema distribuido es capturada por la noción de un *proceso*. Consideramos que un sistema está compuesto por un conjunto de procesos $\mathcal{P}$ que es estático y no varía, y cada proceso conoce la identidad de todos los procesos en $\mathcal{P}$. Asumimos que cada proceso ejecuta su propia copia de un *algoritmo*. La suma de estas copias constituye un *algoritmo distribuido*.

Cuando un proceso no se comporta de acuerdo con su algoritmo, decimos que un *fallo* ha ocurrido. Las distintas abstracciones de un proceso se caracterizan por la naturaleza de sus fallos. En esta tesis consideramos el fallo más elemental, en el cual un proceso deja de ejecutar su algoritmo y nunca reanuda la operación. Llamamos a esta una abstracción de proceso *crash-stop*. Decimos que un proceso es *faulty* si alguna vez sufre un fallo, y *correcto* de otro modo.

Consideramos sistemas distribuidos con paso de mensajes, donde los procesos se comunican intercambiando *mensajes* mediante *enlaces* de comunicación, que abstraen la red física que provee dicha comunicación entre procesos. Más aún, asumimos una topología completa, donde cada par de procesos se encuentra conectado por un enlace punto a punto bidireccional. Los mensajes en estos enlaces pueden ser perdidos, duplicados, reordenados, demorados arbitrariamente y sujetos a ataques maliciosos mientras se encuentren en tránsito.

Las distintas abstracciones de un enlace se caracterizan por el conjunto de propiedades que satisfacen. En el segundo capítulo especificamos las propiedades que se asumen sobre los enlaces. Los capítulos tres y cuatro consideran la abstracción *enlace confiable*. Esta abstracción se caracteriza por las siguientes propiedades: *entrega confiable* y *no duplicación*, que garantizan que cada mensaje enviado por un proceso emisor es entregado a un proceso receptor exactamente una vez, siempre y cuando el emisor y el receptor sean ambos correctos, y *no creación*, la cual asegu-

ra que ningún mensaje es creado o corrompido por la red. Puede parecer irrealista asumir una abstracción enlace confiable cuando es bien sabido que los enlaces físicos pueden perder, duplicar o modificar los mensajes. Esta suposición sólo captura el hecho de que tales problemas pueden ser abordados por algún protocolo de un nivel inferior. Esta funcionalidad se encuentra a menudo en protocolos estándar de transporte como TCP.

En el contexto de paso de mensajes, el comportamiento de cada proceso se caracteriza por el conjunto de mensajes que es capaz de recibir y enviar, el formato de cada uno de estos mensajes, y todas las sequencias validas de intercambios de mensajes. Estas reglas definen un *protocolo distribuido*.

Es crucial para el análisis de algoritmos distribuidos el especificar las suposiciones referidas al tiempo que hacemos sobre procesos y enlaces. Esto es, si se supone que cada proceso está equipado con un reloj físico, o si existen límites en los retrasos impuestos por el procesamiento o la comunicación. En esta tesis estudiamos sistemas distribuidos *asíncronos*, donde no realizamos ninguna suposición respecto del tiempo.

Una combinación de las abstracciones mencionadas anteriormente define un *modelo de sistema distribuido*. Razonar sobre algoritmos distribuidos involucra primero definir un modelo de sistema distribuido claro donde se supone que operan estos algoritmos. Dado un modelo de sistema, el siguiente paso es comprender cómo construir abstracciones que capturen patrones de interacción recurrentes en aplicaciones distribuidas.

1.2. Safety

Una propiedad de *safety* establece que un algoritmo no debe hacer nada incorrecto. Cuando una propiedad de safety es violada, debe serlo en un tiempo finito. Una vez que la propiedad ha sido violada, nunca puede ser satisfecha nuevamente. Más precisamente, una propiedad de safety es una propiedad tal que, cada vez que se viola en una ejecución σ de un algoritmo, existe una ejecución parcial σ' de σ tal que la propiedad es violada en cualquier extensión de σ'.

1.3. Pseudocódigo

Describimos nuestros algoritmos mostrando su pseudocódigo. Ellos corresponden a un modelo computacional reactivo en el cual un algoritmo proporciona una interfaz en forma de un conjunto de controladores de eventos. Cada uno de estos reacciona a los eventos entrantes y posiblemente desencadena nuevos eventos.

En este modelo asíncrono basado en eventos, cada proceso se construye como una máquina de estados cuyas transiciones se activan por los eventos correspondientes a la recepción de uno o más mensajes albergando información en uno o más atributos, o cuando alguna condición se cumple en las variables locales de un proceso.

Un controlador se formula en términos de una secuencia de instrucciones introducidas por **when received**, que describe un evento, seguido de pseudocódigo con

instrucciones para ejecutar, como se muestra en la Figura 1.1.

```
1 when received ACCEPT(b, k, m) from p_j
2     pre: status ∈ {LEADER, FOLLOWER} ∧ bal = b;
3     msg[k] ← m;
4     send ACCEPT_ACK(b, k) to p_j;
```

Figura 1.1: Ejemplo de un controlador

Cada proceso ejecuta el código disparado por los eventos de una manera mutuamente excluyente y atómica. Esto es, el mismo proceso no maneja dos eventos concurrentemente. Una vez que un proceso ha terminado de manejar un evento, este comprueba si se ha desencadenado algún otro evento. Esta verificación periódica se asume justa y se logra de manera implícita: no es visible en el pseudocódigo que mostramos.

Un evento también puede estar restringido por alguna condición en las variables locales, en forma de precondición.

```
1 pre: status ∈ {LEADER, FOLLOWER} ∧ bal = b;
```

Figura 1.2: Uso de precondiciones

Un controlador con una precondición como en la Figura 1.2 ejecuta sus intrucciones sólo cuando el evento correspondiente ha sido disparado y la condición se satisface. Un algoritmo que utiliza controladores de eventos condicionales se basa en que el sistema en tiempo de ejecución pueda almacenar los eventos externos hasta que se cumpla la condición en las variables internas.

1.4. Reducción

Dada una ejecución *fine-grained* de un sistema concurrente o distribuido real, podemos utilizar *reducción* para convertirla en una ejecución equivalente de pasos *coarse-grained*, simplificando la verificación. Específicamente, dos pasos en una ejecución pueden intercambiar lugares si hacerlo no tiene efecto en el resultado de la misma. Aplicar reducción requiere identificar todos los pasos en el sistema, probar relaciones de conmutatividad entre ellos, y aplicar esas relaciones para crear ejecuciones equivalentes con una forma más útil.

En el contexto de nuestro trabajo, utilizamos reducción para asumir de manera segura que no sólo cada proceso ejecuta un evento de manera mutuamente excluyente, sino que además es mutuamente excluyente entre procesos.

1.5. Invariantes inductivos

Una técnica para probar safety consiste en utilizar *invariantes inductivos*. Decimos que un invariante representado por una fórmula ϕ en alguna lógica es un invariante inductivo de un algoritmo $\mathcal{C}$, si las siguientes condiciones se cumplen:
(i) ϕ se cumple inicialmente, y
(ii) ϕ es preservada por $\mathcal{C}$.
Si ϕ implica una propiedad P, entonces $\mathcal{C}$ satisface P.

En términos de la lógica de Floyd-Hoare, (ii) significa que $\{\phi\}\ \mathcal{C}\ \{\phi\}$ es una terna de Hoare válida. Probar la validez de $\{\phi\}\ \mathcal{C}\ \{\phi\}$ se reduce a probar la insatisfacibilidad de la fórmula $\phi(V)\ \wedge\ \delta(V, V') \wedge \neg\phi(V')$, donde V es el conjunto de variables que ocurren en $\mathcal{C}$, $\phi(V)$ denota la aserción ϕ antes de $\mathcal{C}$, $\phi(V')$ denota la aserción ϕ luego de $\mathcal{C}$, y $\delta(V, V')$ es una fórmula que expresa la semántica de $\mathcal{C}$ como una relación entre los estados antes y luego de $\mathcal{C}$.

Sin embargo, no todos los invariantes son inductivos. Un invariante ϕ no es inductivo cuando asumir que ϕ se satisface antes de la ejecución de algún algoritmo no es suficiente para garantizar que ϕ se satisface luego de la ejecución del mismo, aún cuando se sabe que ϕ es un invariante del algoritmo. Esto sólo puede ocurrir cuando ϕ no excluye ciertos estados que no son alcanzables por el algoritmo.

Considérese, por ejemplo, el algoritmo en la Figura 1.3. Esta muestra un bucle simple para el cual la propiedad $\phi(x) \triangleq x \text{ mód } 2 = 0$ es un invariante.

```
1 x ← 0;
2 y ← 0;
3 while (∗) do
4     x ← x + y;
5     y ← y + 2;
```

Figura 1.3: Bucle simple

Sin embargo, ϕ no es un invariante inductivo para el bucle. Por ejemplo, si ejecutamos el bucle partiendo de $x = 0 \wedge y = 1$ arribamos a un estado en el cual $x = 1\ \wedge\ y = 3$. Es simple ver que para el cuerpo del bucle tenemos $\delta(V, V') \triangleq x' = x + y \wedge y' = y + 2$. De hecho, la fórmula $x \text{ mód } 2 = 0 \wedge x' = x + y \wedge y' = y + 2 \wedge x' \text{ mód } 2 \neq 0$ es satisfecha por $x = 0 \wedge y = 1 \wedge x' = 1 \wedge y' = 3$. Esto ocurre porque partimos de un estado que no es un estado válido del programa, es decir, alcanzable por el mismo, pero que satisface ϕ.

En otras palabras, asumir ϕ antes de una ejecución del cuerpo del bucle no es suficiente para garantizar que también se cumple luego de la misma. Para ser inductiva, la propiedad necesita ser fortalecida para incluir información sobre y, por ejemplo, de la siguiente manera: $\phi(x) \triangleq x \text{ mód } 2 = 0 \wedge y \text{ mód } 2 = 0$. Los invariantes inductivos son usualmente difíciles de encontrar tanto manualmente como utilizando análisis automático de programas.

1.6. Dafny

Dafny es un lenguaje de programación y un verificador de programas. El lenguaje incluye construcciones de especificación que permiten anotar su código para describir el comportamiento deseado y su verificador comprueba que el programa satisfaga su especificación. Por lo tanto, Dafny se puede utilizar para desarrollar código demostrablemente correcto.

El lenguaje Dafny es type-safe y secuencial. Este incluye construcciones halladas en lenguajes imperativos comúnes, como bucles y asignación dinámica de objetos, como así también en lenguajes funcionales, como funciones recursivas y tipos de datos inductivos y coinductivos. Su poder real proviene de la capacidad de especificar comportamiento a través de construcciones tales como precondiciones, postcondiciones, invariantes de bucles, etc, junto con el código que se supone implementa ese comportamiento.

El verificador comprueba que el código se comporte como se describe, pero también verifica terminación y la ausencia de errores en tiempo de ejecución, tales como indexar una arreglo fuera de sus límites. La correctitud de un programa es en definitiva reducida a la insatisfacibilidad de un conjunto de fórmulas, conocidas como *obligaciones de prueba*, que Dafny descarga en el SMT solver Z3. El verificador responde dando mensajes de error para las obligaciones de prueba que es no es capaz de probar.

```
method BinarySearch(a: array<int>, key: int) returns (r: int)
  requires ∀ i,j • 0 ≤ i < j < a.Length ⟹ a[i] ≤ a[j]
  ensures 0 ≤ r ⟹ r < a.Length ∧ a[r] = key
  ensures r < 0 ⟹ key ∉ a[..]
{
  var lo, hi := 0, a.Length;
  while lo < hi
    invariant 0 ≤ lo ≤ hi ≤ a.Length
    invariant key ∉ a[..lo] ∧ key ∉ a[hi..]
  {
    var mid := (lo + hi) / 2;
    if key < a[mid] {
      hi := mid;
    } else if a[mid] < key {
      lo := mid + 1;
    } else {
      return mid;
    }
  }
  return -1;
}
```

Figura 1.4: Ejemplo de un programa en Dafny

Ilustramos algunas de las características básicas de Dafny a través de un ejemplo, tomado de *Developing Verified Programs With Dafny*, por Rustan Leino, y mostrado en la Figura 1.4.

La figura muestra un *método*, que es un procedimiento de código. Las *postcondiciones* del método, introducidas a través de la palabra clave **ensures**, establecen todas aquellas condiciones que se deben cumplir luego de la ejecución del método.

El verificador comprueba que todos los posibles caminos de ejecución conducen a estados en los cuales las postcondiciones son satisfechas.

Las *precondiciones* del método, introducidas por la palabra clave **requires**, establecen qué condiciones deben ser garantizadas para invocar el método, que es validado por el verificador en todos los puntos de llamada, y por lo tanto, se puede asumir que estas condiciones se cumplen en la entrada al cuerpo del método.

El método también muestra un bucle. Para razonar sobre bucles, es necesario anotar el código con un *invariante de bucle*, una condición que se satisface al entrar al bucle, y que es preservada por el cuerpo del bucle. Dafny sabe entonces que luego de la ejecución del bucle, el invariante en conjunto con la negación de la guarda se satisface.

Un punto clave para entender cómo razonar sobre bucles es que no es posible considerar el comportamiento de un bucle mirando todos los posiblemente infinitos unrollings. En cambio, se debe idear un invariante que capture toda la información sobre las variables modificadas por el bucle. En otras palabras, el verificador no observa el cuerpo del bucle al razonar sobre cuáles estados del programa son alcanzables luego de un número arbitrario de iteraciones. En cambio, se basa en el invariante provisto por el programador para lograr una descripción lo suficientemente restringida de esos estados. Esta distinción es quizás la diferencia más significativa entre razonar sobre programas y simplemente ejecutarlos.

En algunas ocasiones, Dafny puede inferir automáticamente la prueba de terminación. Por ejemplo, en este caso, Dafny sabe que $hi - lo$ decrece y que se encuentra acotada inferiormente por 0, como se puede deducir del primer invariante. De otro modo, se necesitaría anotar el código con una función decreciente acotada inferiormente.

Una *función* en Dafny es una función matemática. Esta es definida no por código sino por una expresión. Las funciones pueden ser utilizadas en un estilo funcional de programación, y son frequentemente utilizadas en la especificación de programas posiblemente imperativos. Así como para los métodos, las precondiciones establecen cuando se permite que una función sea invocada. La Figura 1.5 muestra un ejemplo.

```
function abs(x: int): int {
  if x < 0 then -x else x
}
```

Figura 1.5: Ejemplo de una función en Dafny

Dafny también provee construcciones para asistir al verificador en obligaciones que no es capaz de probar, aún cuando son verdaderas. Esto ocurre como consecuencia de la indecidibilidad del problema subyacente. A medida que los programas se vuelven más sofisticados, tales situaciones surgen con mayor frecuencia.

El método `ComputePow2` en la Figura 1.6 computa la función `Pow2` con complejidad logarítmica. La correctitud de la segunda rama se sostiene en el hecho de que $2^n = (2^{n/2})^2$ cuando n es par. Este hecho es establecido como un lema. Más precisamente, el método *ghost* `Lemma`, que puede ser invocado cuando n es par, promete retornar en un estado en el cual la propiedad se cumple. Al invocar el lema,

```
method ComputePow2(n: nat) returns (p: nat)
  ensures p = pow2(n);
{
  if n = 0 {
    p := 1;
  } else if n %2 = 0 {
    p := ComputePow2(n / 2);
    p := p * p;
    Lemma(n);
  } else {
    p := ComputePow2(n-1);
    p := 2 * p;
  }
}

ghost method Lemma(n: nat)
  requires n %2 = 0;
  ensures pow2(n) = pow2(n/2) * pow2(n/2);
{
  if n ≠ 0 { Lemma(n-2); }
}
```

Figura 1.6: Una forma logarítmica de computar `Pow2`

`ComputePow2` obtiene de este modo la información que necesita. La prueba del lema es el cuerpo del método ghost, donde cada posible camino del código debe convencer al verificador que la postcondición se satisface. Puesto que `Lemma` es recursivo, la prueba corresponde a una por inducción.

Las entidades declaradas como ghost son sólo utilizadas durante la verificación, por lo que el compilador las omite del código ejecutable. En consecuencia, ellas pueden ser utilizadas para asistir a Dafny a completar una prueba sin afectar el costo de ejecución.

Estas son sólo algunas de las múltiples características incluidas en Dafny. Para mayor referencia, consúltese *Dafny Reference Manual* por Ford y Leino.

Capítulo 2

Single Decree Paxos

2.1. Introducción

Los algoritmos de *consenso* permiten a un grupo de procesos alcanzar acuerdo. Estos son un componente esencial de los servicios tolerantes a fallos implementados como sistemas distribuidos con paso de mensajes. En el contexto de *state machine replication*, por ejemplo, cada proceso contiene una réplica del estado del sistema, y ciertos procesos pueden proponer valores para el próximo estado del mismo. Puesto las réplicas deben ser mutuamente consistentes, para cada actualización del sistema, los procesos que no han fallado ejecutan una instancia de un protocolo de consenso para decidir uniformemente el resultado.

El algoritmo Paxos es el protocolo de consenso clásico, y su versión single-decree es un mecanismo para lograr acuerdo en un único valor sobre canales de comunicación no confiables.

2.2. Modelo de sistema

Consideramos un sistema asíncrono con paso de mensajes compuesto por un conjunto de procesos $\mathcal{P}$ crash-stop, es decir, fallan deteniendo permanentemente su ejecución. Un proceso es *correcto* si nunca falla. Asumimos que los procesos se encuentran conectados por enlaces en los que los mensajes pueden ser reordenados, duplicados o perdidos, pero no corrompidos. Existen $2f + 1$ procesos de los cuales a lo sumo f pueden fallar. Llamamos a un conjunto de al menos $f + 1$ procesos en $\mathcal{P}$ un *quórum*.

2.3. El problema

Consideramos el problema de implementar la abstracción consenso en el sistema descripto arriba. La abstracción provee dos primitivas:

`propose`(v): permite a un proceso proponer un valor v para consenso a todos los procesos en $\mathcal{P}$.

`decide`(v): permite a un proceso decidir que un valor v ha sido acordado y entregar este valor a su capa de aplicación.

La abstracción garantiza que a lo sumo uno de entre los valores propuestos es acordado. Un algoritmo es una implementación correcta de consenso si cada ejecución del mismo satisface:

Validez: Si un proceso decide un valor, entonces este fue propuesto.
Integridad: Ningún proceso decide diferentes valores.
Acuerdo: Ningunos dos procesos deciden de manera diferente.

2.4. El protocolo

```
1 bal ← 0 ∈ ℤ;
2 cbal ← 0 ∈ ℤ;
3 pv ← ⊥ ∈ {⊥} ∪ ℳ;
4 av ← ⊥ ∈ {⊥} ∪ ℳ;
5 status ← ELECTION ∈ {LEADER, FOLLOWER, ELECTION};
```

Figura 2.1: Variables locales de un proceso

Visión general y estado de los procesos. Esta sección provee una visión general del protocolo e introduce las variables de estado utilizadas por cada proceso, dadas en la Figura 2.1.

El protocolo utiliza un enfoque *basado en un líder*. Para proponer un valor, un proceso precisa primero ser electo como líder. Un líder gana una elección si recibe votos de una mayoría, obteniendo de esta manera el derecho a proponer valores. Con este fin, un proceso persiste el valor que desea proponer en la variable pv y envía una invitación a votarlo a cada proceso en el grupo.

Una vez electo, el líder ha comenzado un período de ejecución llamado *ballot*. En cualquier momento, cada proceso participa en un único ballot alojado en la variable bal. Una variable status registra si el proceso es un LEADER, un FOLLOWER o se encuentra en un estado especial ELECTION utilizado durante los cambios de líderes.

El protocolo se basa en una regla de mayoría simple: para que una propuesta sea *elegida*, el líder necesita convencer a un quórum de *aceptar* su propuesta. Al recibir una propuesta, los seguidores notifican la recepción al líder, aceptándola de este modo. Los procesos persisten su última propuesta aceptada en la variable av, y el ballot en que lo hicieron en una variable cbal.

Si una mayoría de seguidores ha aceptado una propuesta, el valor propuesto es acordado y puede ser decidido en los procesos participantes, esto es, entregado a la capa de aplicación en el mismo proceso.

Fase 1. En la primera fase del protocolo, un proceso intenta que un quórum participe de su ballot y reúne información acerca de propuestas ya aceptadas. Su código se muestra en la Figura 2.2.

Para proponer un valor v, un proceso ejecuta la función `propose` (línea 6). El proceso persiste el valor que quiere proponer en la variable pv, escoge un ballot que lidera mayor que bal e inicia una elección de líder. Con este propósito, envía el nuevo número de ballot en un mensaje `P1A` a todos los procesos, incluyéndose a sí mismo (línea 8).

Cuando un proceso se une a un ballot, promete ignorar cualquier invitación a un ballot no mayor. Por lo tanto, cuando un proceso recibe un mensaje `P1A`(b) (línea 9), verifica primero que b sea mayor a bal (línea 10). En este caso, actualiza su bal a b, uniéndose de esta manera al ballot, cambia su status a ELECTION y notifica la recepción al líder enviando un mensaje `P1B`. El mensaje `P1B` porta el número de ballot notificado, el último valor aceptado por el proceso y el ballot en el cual el proceso lo ha aceptado (línea 13).

```
6  function propose(v):
7      pv ← v;
8      send P1A(b | b > bal ∧ leader(b) = p_i) to all;
9  when received P1A(b) from p_j
10     pre: bal < b;
11     bal ← b;
12     status ← ELECTION;
13     send P1B(b, cbal, av) to p_j;
```

Figura 2.2: El protocolo en un proceso p_i: Fase 1

Fase 2. Explicamos a continuación la segunda fase del protocolo mostrada en la Figura 2.3, a través de la cual un proceso intenta que un quórum acepte su propuesta.

El futuro líder del ballot b espera hasta que reciba mensajes `P1B` para el ballot propuesto de un quórum de procesos (línea 14). Este sabe entonces que un quórum se ha unido a su ballot y se convierte en un líder establecido (línea 16). Si ningún proceso en el quórum reporta haber aceptado un valor (línea 17), el líder es libre de proponer su valor. En este caso, envía su propuesta en un mensaje `P2A` a todos los procesos, incluyéndose a sí mismo (línea 18). El mensaje porta el número de ballot y el valor propuesto. Si, por el contrario, algún proceso en el quórum reporta haber aceptado un valor, el líder escoge el valor aceptado en el máximo número de ballot de entre los reportados en las respuestas (línea 20), y lo envía en un mensaje `P2A` (línea 22).

Al recibir un mensaje `P2A` (línea 23), un proceso p_i comprueba que no corresponda a un ballot menor que bal (línea 24), para no violar su promesa de ignorar mensajes provenientes de ballots menores. En este caso, el proceso actualiza su último valor aceptado, el ballot en que lo ha aceptado y su ballot actual. Si el proceso no lidera el ballot entonces no se corresponde a su propuesta, por lo tanto establece su status a FOLLOWER. El proceso notifica entonces al líder del ballot b que ha aceptado su propuesta enviando un mensaje `P2B` (línea 30).

El líder del ballot b actúa una vez que recibe mensajes `P2B` para el ballot b y el

valor v de un quórum de procesos (línea 31), en cuyo caso sabe que se ha alcanzado acuerdo. Este notifica entonces a todos los procesos que el valor puede ser entregado de forma segura a través de un mensaje `DECISION` (línea 32).

Finalmente, al recibir un mensaje `DECISION` (línea 33), el proceso entrega el valor acordado a su capa de aplicación utilizando la primitiva `decide` (línea 34).

```
14 when received {P1B(b, cbal_j, av_j) | p_j ∈ Q} from a quorum Q
15     pre: bal = b ∧ status = ELECTION;
16     status ← LEADER;
17     if ∀p_j . p_j ∈ Q ⇒ av_j = ⊥ then
18         send P2A(b, pv) to all;
19     else
20         let j_0 be such that p_{j_0} ∈ Q, av_{j_0} ≠ ⊥ ∧
21               ∀p_j ∈ Q . av_j ≠ ⊥ ⇒ cbal_j ≤ cbal_{j_0};
22         send P2A(b, av_{j_0}) to all;
23 when received P2A(b, v) from p_j
24     pre: bal ≤ b;
25     av ← v;
26     bal ← b;
27     cbal ← b;
28     if p_i ≠ leader(b) then
29         status ← FOLLOWER;
30     send P2B(b, v) to p_j;
31 when received {P2B(b, v) | p_j ∈ Q} from a quorum Q
32     send DECISION(b, v) to all;
33 when received DECISION(b, v)
34     decide(v);
```

Figura 2.3: El protocolo en un proceso p_i: Fase 2

2.5. Correctitud

Un proceso sólo puede decidir un valor v en respuesta a un mensaje `DECISION`$(\_, v)$. Por lo tanto, para probar que ningún proceso decide diferentes valores, y que ningunos dos procesos deciden de manera diferente, es suficiente probar que si `DECISION`$(\_, v)$ y `DECISION`$(\_, v')$ son mensajes enviados, entonces $v = v'$.

Para construir un argumento inductivo, hay una sutileza que se debe tener en cuenta. No se puede asumir que si `DECISION`$(b, \_)$ y `DECISION`$(b', \_)$ son mensajes enviados con $b < b'$, entonces `DECISION`$(b, \_)$ fue enviado antes que `DECISION`$(b', \_)$. De hecho, como diferentes procesos pueden participar en diferentes ballots al mismo tiempo, es posible que existan mensajes siendo aceptados en el ballot b mientras un

ballot $b' > b$ sea operacional. En particular, un valor puede ser elegido en el ballot b' antes de que un valor sea elegido en el ballot $b < b'$. Por lo tanto, el protocolo debe garantizar que cualquier valor elegido en un ballot b' no contradice ningún valor potencialmente elegido en ballots menores a b'.

Decimos que un valor v es *choosable* en el ballot b si v ha sido elegido o puede aún ser elegido en el ballot b. Un valor v ha sido elegido o puede aún ser elegido en el ballot b si existe un quórum de procesos $\mathcal{Q}$ tal que cada proceso $q \in \mathcal{Q}$ ha recibido y notificado un mensaje $\texttt{P2A}(b, v)$ o aún puede hacerlo. Un proceso aún puede notificar un mensaje $\texttt{P2A}(b, v)$ sólo si su ballot es menor o igual que b. Por lo tanto,

$$\begin{aligned}\texttt{choosable}(b, v) \triangleq \exists \mathcal{Q}.\ &\mathcal{Q} \text{ es un quórum} \wedge \\ &(\forall q.\ q \in Q \Rightarrow q \text{ envió } \texttt{P2B}(b, v)\ \vee\ \mathsf{bal}_q \leq b)\end{aligned}$$

La Figura 2.4 muestra los invariantes principales mantenidos por el protocolo, de los cuales probamos los más interesantes a continuación.

1. Para cualquier mensajes $\texttt{P2A}(b, v)$ y $\texttt{P2A}(b, v')$, tenemos $v = v'$.
2. Para cualquier mensajes $\texttt{P2B}(b, \_)$ y $\texttt{P1B}(b', c', \_)$ enviados por el mismo proceso con $b' > b$, tenemos $c' \geq b$.
3. Para cualquier mensaje $\texttt{P1B}(\_, c, v)$ con $v \neq \bot$, tenemos que $\texttt{P2A}(c, v)$ es también un mensaje enviado.
4. Si $\texttt{choosable}(b, v)$ y $\texttt{P2A}(b', v')$ es un mensaje enviado con $b' > b$, entonces $v = v'$.
5. Para cualquier mensajes $\texttt{DECISION}(\_, v)$ y $\texttt{DECISION}(\_, v')$, tenemos $v = v'$.
6. a. Si en un proceso tenemos $\mathsf{status} = \text{LEADER}$, entonces $\mathsf{pv} \neq \bot$.
   b. Si en un proceso tenemos $\mathsf{av} \neq \bot$, entonces av fue propuesto.
   c. Para cualquier mensaje $\texttt{P1B}(\_, \_, v)$ con $v \neq \bot$, v fue propuesto.
   d. Para cualquier mensajes $\texttt{P2A}(\_, v)$, $\texttt{P2B}(\_, v)$ o $\texttt{DECISION}(\_, v)$, v fue propuesto.

Figura 2.4: Invariantes principales mantenidos por el protocolo.

Lema 1. *Si* $\texttt{choosable}(b, v)$ *se cumple luego de un paso del protocolo, entonces también se cumplía antes del mismo.*

Demostración. Si $\texttt{choosable}(b, v)$ se cumple entonces existe un quórum $\mathcal{Q}$ tal que cada proceso $q \in \mathcal{Q}$ ha recibido y notificado un mensaje $\texttt{P2A}(b, v)$ con $\texttt{P2B}(b, v)$ o q tiene un ballot menor o igual a b.

El número de ballot sólo puede ser afectado por las transiciones en las líneas 9 y 23. En todos los casos el número de ballot no decrece. En consecuencia, si es menor o igual que b luego de un paso del protocolo, así lo era antes del mismo.

La única transición en la cual un mensaje P2B es enviado y por lo tanto afecta la validez de la afirmación es aquella en la línea 23. Supóngase entonces que un proceso ha enviado un mensaje $\texttt{P2B}(b, v)$ durante la transición actual. La precondición en la línea 24 implica que su ballot era menor o igual que b antes del paso actual, y en consecuencia $\texttt{choosable}(b, v)$ se cumplía antes del mismo. □

Prueba del Invariante 1. Asumimos que dos procesos diferentes no pueden liderar el mismo número de ballot. Por lo tanto, si $\texttt{P2A}(b, \_)$ y $\texttt{P2A}(b, \_)$ son mensajes enviados, deben haber sido enviados por el mismo proceso p. Sin embargo, justo antes de enviar el primer mensaje P2A, p cambió su status a LEADER, y no puede haber enviado otro mensaje P2A hasta cambiar su status a ELECTION. Como esto sólo puede ocurrir durante la transición en la línea 9, esto significa que p también incrementó su número de ballot. En consecuencia, no pueden haber dos mensajes P2A diferentes enviados por el mismo proceso. □

Prueba del Invariante 2. Sean $\texttt{P2B}(b, \_)$ y $\texttt{P1B}(b', c', \_)$ mensajes enviados por un proceso p con $b' > b$. Supóngase que p envió $\texttt{P1B}(b', c', \_)$ antes de enviar $\texttt{P2B}(b, \_)$. Como los números de ballot son no decrecientes, debe ocurrir que $\mathsf{bal}_p \geq b' > b$ justo antes de la transición en la cual p envió el mensaje $\texttt{P2B}(b, \_)$, lo cual contradice la precondición en la línea 24. Por lo tanto, p debe haber enviado el mensaje $\texttt{P2B}(b, \_)$ antes de enviar el mensaje $\texttt{P1B}(b', c', \_)$.

La línea 27 implica que $\mathsf{cbal}_p = b$ justo antes de enviar $\texttt{P2B}(b, \_)$. Como cbal_p es también no decreciente, entonces $\mathsf{cbal}_p \geq b$ en el momento en que p envió el mensaje $\texttt{P1B}(b', c', \_)$, y por lo tanto $c' = \mathsf{cbal}_p \geq b$. □

Prueba del Invariante 3. Un mensaje $\texttt{P1B}(\_, c, v)$ sólo puede ser enviado por un proceso p durante la transición en la línea 9. Por ello, c y v son los valores de cbal_p y av_p, respectivamente, en el momento en que p envió el mensaje P1B.

Como $v \neq \perp$, y la única transición en la cual av_p y cbal_p son modificados es aquella en la línea 23, esto significa que p sólo puede haber actualizado el valor de av_p en respuesta a un mensaje $\texttt{P2A}(\mathsf{cbal}_p, \mathsf{av}_p) = \texttt{P2A}(c, v)$. □

Prueba del Invariante 4. Probamos el invariante por inducción en la longitud de la ejecución del protocolo. Inicialmente no existen mensajes P2A enviados, por lo que la afirmación es trivialmente verdadera. Para el paso inductivo, supóngase que $\texttt{choosable}(b, v)$ se cumple.

Por lo tanto, existe un quórum $\mathcal{Q}$ tal que para todo $p \in \mathcal{Q}$ o bien p envió un mensaje $\texttt{P2B}(b, v)$ o $\mathsf{bal}_p \leq b$. Sea $\texttt{P2A}(b', v')$ con $b' > b$, un mensaje enviado por un proceso q. Si $\texttt{P2A}(b', v')$ fue enviado antes del paso actual del protocolo, entonces el Lema 1 junto con la hipótesis inductiva implican que $v = v'$.

Sea entonces $\texttt{P2A}(b', v')$ un mensaje enviado durante la transición actual del protocolo. En consecuencia, como esto sólo puede ocurrir como resultado de la transición en la línea 14, entonces existe un quórum de procesos $\mathcal{Q}'$ tal que q recibió un mensaje $\texttt{P1B}(b', cbal_j, av_j)$ de cada $p_j \in \mathcal{Q}'$.

Dado que $\mathcal{Q}$ y $\mathcal{Q}'$ son quórums, existe $p \in \mathcal{Q} \cap \mathcal{Q}'$. Como p notificó el ballot b', esto es, participó en la elección del líder de b', y los números de ballot no decrecen,

entonces $\mathsf{bal}_p \geq b' > b$. Como $p \in \mathcal{Q}$ y $\mathsf{bal}_p > b$, entonces debe ser el caso que p envió un mensaje $\texttt{P2B}(b, v)$ y por ello, por el Invariante 2, $cbal_p \geq b$. Entonces, no puede ser el caso que para cada $p_j \in \mathcal{Q}'$, se tenga $av_j = \bot$. Por lo tanto, en la línea 20, j_0 es elegido tal que $v' = av_{j_0} \neq \bot$ y $cbal_{j_0} \geq cbal_j$ para cada $p_j \in \mathcal{Q}'$, en particular, $cbal_{j_0} \geq b$.

Como el mensaje $\texttt{P1B}(b', cbal_{j_0}, av_{j_0}) = \texttt{P1B}(b', cbal_{j_0}, v')$ es un mensaje enviado, el Invariante 3 implica que también lo es $\texttt{P2A}(cbal_{j_0}, v')$.

Si $cbal_{j_0} = b$, entonces $\texttt{P2A}(b, v')$ es un mensaje enviado. Como p envió un mensaje $\texttt{P2B}(b, v)$, debe ser el caso que $\texttt{P2A}(b, v)$ es también un mensaje enviado. Por lo tanto, por el Invariante 1, debemos tener $v = v'$. Supóngase entonces que $cbal_{j_0} > b$. El Lema 1 implica que $\texttt{choosable}(b, v)$ también se cumplía antes del paso actual del protocolo. En consecuencia, la hipótesis inductiva aplicada a $\texttt{P2A}(cbal_{j_0}, v')$ implica que $v = v'$. □

Prueba del Invariante 5. Sean b y b' números de ballot tales que $\texttt{DECISION}(b, v)$ y $\texttt{DECISION}(b', v')$ son mensajes enviados. Como un mensaje `DECISION` sólo puede ser enviado durante la transición en la línea 31, existen quórums $\mathcal{Q}$ y $\mathcal{Q}'$ tales que cada proceso $q \in \mathcal{Q}$ envió un mensaje $\texttt{P2B}(b, v)$, y cada proceso $q' \in \mathcal{Q}'$ envió un mensaje $\texttt{P2B}(b', v')$.

Supóngase $b = b'$. Como un mensaje `P2B` sólo puede ser enviado en respuesta a un mensaje `P2A` correspondiente durante la transición en la línea 23, entonces $\texttt{P2A}(b, v)$ y $\texttt{P2A}(b', v')$ deben ser mensajes enviados, en cuyo caso, por el Invariante 1, debe ser $v = v'$. Supóngase entonces sin pérdida de generalidad que $b < b'$. Nótese que $\texttt{choosable}(b, v)$ se cumple. Por lo tanto, como $\texttt{P2A}(b', v')$ es un mensaje enviado, el Invariante 4 implica que $v = v'$. □

Teorema 1. *El protocolo satisface Validez.*

Teorema 2. *El protocolo satisface Integridad y Acuerdo.*

Demostración. Sean v y v' valores decididos por el mismo o diferentes procesos. Como un proceso sólo puede decidir un valor en respuesta a un mensaje `DECISION`, entonces $\texttt{DECISION}(\_, v)$ y $\texttt{DECISION}(\_, v')$ son mensajes enviados. El Invariante 5 implica que $v = v'$, como queríamos demostrar. □

Capítulo 3

Broadcast atómico

3.1. Introducción

Un protocolo de *broadcast atómico* permite a un proceso transmitir un mensaje a un grupo de procesos. Estos protocolos garantizan que los procesos no sólo acuerdan el conjunto de mensajes entregados sino que también el orden en que esto ocurre, a pesar de fallos. Por ello, son un componente clave para implementar sistemas distribuidos tolerantes a fallos, pues permiten a un grupo de procesos aplicar cambios incrementales de estado de manera consistente, permaneciendo en sincronía.

Sin embargo, los protocolos de broadcast atómico son notoriamente costosos. Durante la recuperación a fallos, los procesos intercambian estado para acordar aquel a partir del cual se reanuda el funcionamiento, lo que en general implica grandes volúmenes de información. Presentamos un protocolo que reduce de manera significativa la cantidad de estado transferido durante este proceso.

3.2. Modelo de sistema

Consideramos un sistema asíncrono con paso de mensajes compuesto por un conjunto de procesos $\mathcal{P}$ crash-stop, es decir, fallan deteniendo permanentemente su ejecución. Un proceso es *correcto* si nunca falla. Asumimos que los procesos se encuentran conectados por enlaces confiables FIFO: los mensajes se entregan en orden FIFO, y se garantiza que los mensajes entre procesos correctos son eventualmente entregados. Existen $2f + 1$ procesos de los cuales a lo sumo f pueden fallar. Llamamos a un conjunto de al menos $f + 1$ procesos en $\mathcal{P}$ un quórum.

3.3. El problema

Consideramos el problema de implementar la abstracción broadcast atómico en el sistema descripto arriba. La abstracción provee dos primitivas:

`broadcast`(m): permite a un proceso enviar un mensaje de aplicación m a todos los procesos en $\mathcal{P}$.

`deliver`(m): permite a un proceso entregar un mensaje recibido m a su capa de aplicación.

Por simplicidad, asumimos que los mensajes enviados por los procesos son únicos. La abstracción garantiza que los procesos entregan un prefijo completo de una secuencia global común de mensajes. Un algoritmo es una implementación correcta de broadcast atómico si cada ejecución del mismo satisface:

Integridad: Cada mensaje entregado fue previamente enviado.
No duplicación: Ningún mensaje se entrega más de una vez al mismo proceso.
Orden confiable: Todos los procesos entregan un prefijo completo de un orden global común sobre los mensajes.

3.4. El protocolo

```
1 bal ← 0 ∈ ℤ;
2 cbal ← 0 ∈ ℤ;
3 next ← −1 ∈ ℤ;
4 last_delivered ← −1 ∈ ℤ;
5 msg[ ] ∈ ℕ → {⊥} ∪ ℳ;
6 status ∈ {LEADER, FOLLOWER, RECOVERING};
```

Figura 3.1: Variables locales de un proceso

Visión general y estado de los procesos. Esta sección provee una visión general del protocolo e introduce las variables de estado utilizadas por cada proceso, dadas en la Figura 3.1.

Nuestro protocolo utiliza un enfoque *basado en un líder*. Inicialmente, los procesos intentan elegir un líder. Un líder gana una elección si recibe votos de una mayoría, obteniendo de esta manera el derecho a proponer mensajes. Una vez electo, el líder ha comenzado un período de ejecución llamado *ballot*. En cualquier momento, cada proceso participa en un único ballot alojado en la variable bal.

Los procesos transmiten mensajes utilizando la primitiva `broadcast`. El líder recibe los mensajes entrantes y los coloca en el arreglo msg añadiendolos al final del mismo, y dictando de este modo el orden en que estos deben ser entregados. Con este fin, registra el último slot no vacío en el arreglo en una variable next. Para tolerancia a fallos, el ordenamiento en los mensajes precisa ser finalizado a través de consenso, que estipula una ronda de notificaciones desde los seguidores hacia el líder. Para ello, el líder propaga los mensajes a los seguidores.

Al recibir una propuesta, los seguidores la almacenan en su copia local del arreglo msg y notifican su recepción al líder, aceptándola de este modo. Si una mayoría de seguidores acepta el mensaje, se garantiza su supervivencia a cualquier fallo tolerado y puede ser entregado en los procesos participantes utilizando la primitiva `deliver`. Los procesos persisten la posición del último mensaje entregado en el arreglo msg en una variable last_delivered.

Cuando se sospecha el fallo de un líder, o un líder sospecha que ha perdido su quórum, los procesos ejecutan un protocolo de recuperación para acordar un estado consistente común antes de reanudar la operación normal, y también para establecer un nuevo líder capaz de transmitir nuevos mensajes. El propósito del protocolo de recuperación es elegir un líder que ha convencido un quórum de procesos de unirse a su ballot y cuyo estado es al menos tan actual como el de cualquiera de sus votantes. El segundo requerimiento garantiza que el nuevo líder preserva cualquier mensaje entregado en ballots previos, mientras previene la necesidad de transferir estado desde los seguidores al futuro líder. Una variable status registra si el proceso es un LEADER, un FOLLOWER o se encuentra en un estado especial RECOVERING utilizado durante los cambios de líderes.

Para garantizar un estado consistente, se requiere que los seguidores sincronicen su estado con el futuro líder antes de que empiecen a aceptar mensajes. Como pueden existir múltiples intentos fallidos de elección, utilizamos una variable de ballot adicional cbal para representar el último ballot en el cual un proceso ha sincronizado su estado con aquel del futuro líder.

```
7  when received broadcast(m)
8      pre: status = LEADER;
9      if ∀k. msg[k] ≠ m then
10         next ← next + 1;
11         msg[next] ← m;
12         send ACCEPT(bal, next, m) to P;

13 when received ACCEPT(b, k, m) from p_j
14     pre: status ∈ {LEADER, FOLLOWER} ∧ bal = b;
15     msg[k] ← m;
16     send ACCEPT_ACK(b, k) to p_j;

17 when received a quorum of ACCEPT_ACK(b, k)
18     pre: status = LEADER ∧ bal = b;
19     send COMMIT(b, k, msg[k]) to P;

20 when received COMMIT(b, k, m)
21     pre: status ∈ {LEADER, FOLLOWER} ∧ bal = b ∧ k = last_delivered + 1;
22     last_delivered ← k;
23     msg[k] ← m;
24     deliver(msg[k]);
```

Figura 3.2: El protocolo en un proceso p_i: caso libre de fallos

Caso libre de fallos. En el modo normal del protocolo, un único líder propone una secuencia de mensajes a sus seguidores con el objetivo de replicarlos y garantizar la durabilidad de los mensajes entregados y su orden. Su código se muestra en la Figura 3.2.

Para transmitir un mensaje m, un proceso lo envía en un mensaje `broadcast`. Un proceso p_i opera sobre el mensaje sólo cuando es el líder (línea 8). Al recibir `broadcast`(m), el líder p_i comprueba si ha recibido anteriormente este mensaje con el propósito de evitar entregar el mismo mensaje más de una vez (línea 9). En caso contrario, este añade el mensaje a la secuencia ordenada de mensajes msg y realiza un análogo a la "fase 2" de Paxos, intentando convencer al grupo de procesos de aceptar su propuesta. Con este fin, envía un mensaje `ACCEPT` al grupo, incluyéndose a sí mismo por uniformidad, que es análogo al mensaje `P2A` de Paxos (línea 12). El mensaje porta el ballot del líder, la posición del mensaje en el arreglo msg y el mensaje m.

Un proceso opera sobre el mensaje `ACCEPT` sólo si participa en el ballot correspondiente (línea 14). Este almacena el mensaje en su copia local del arreglo msg y luego envía un mensaje `ACCEPT_ACK` al líder de b, análogo al mensaje `P2B` de Paxos (línea 16). El mensaje porta el ballot y el número de slot correspondiente al mensaje que está siendo notificado. Esto confirma que el proceso ha aceptado el mensaje.

El líder del ballot b actúa una vez que recibe mensajes `ACCEPT_ACK` para el ballot b y slot k de cada proceso en un quórum (línea 17), en cuyo caso el mensaje ha estado presente en el arreglo de una mayoría en el mismo ballot y número de slot, y por lo tanto sobrevivirá cualquier fallo tolerado. En este caso, el líder notifica a los seguidores que el mensaje puede ser entregado de forma segura a través de un mensaje `COMMIT` (línea 19).

Al recibir un mensaje `COMMIT` (línea 20), el proceso actualiza last_delivered para registrar el prefijo de la secuencia global común de mensajes que ha entregado y actualiza su arreglo. Los mensajes son entregados en orden creciente de número de slot sin huecos como lo implica la línea 21. Finalmente, el proceso entrega el mensaje a su capa de aplicación (línea 24).

Caso de recuperación. Explicamos a continuación cómo el protocolo trata los fallos ejecutando el protocolo de recuperación en la Figura 3.3.

El objetivo del protocolo de recuperación es doble: primero, elegir un líder quien convenza a un quórum de participar de su ballot y cuyo estado *domine* al quórum, esto es, su estado es al menos tan actual como el de cualquiera de sus votantes; y segundo, garantizar que antes de que un seguidor empiece a aceptar propuestas del nuevo líder, este ha sincronizado su estado con aquel del líder, asegurando de esta manera que los procesos permanecen en sincronía. Nótese que como el estado del líder electo domina el estado de cada proceso en el quórum, no es necesario transferir estado desde los seguidores hacia el líder.

Describimos ahora el procedimiento de recuperación en detalle. Cuando un seguidor p_i sospecha que su líder ha fallado, o un líder p_i sospecha que ha perdido su quórum, este intenta convertirse en un líder ejecutando la función `recover` (línea 25). El proceso escoge un ballot que lidera mayor a bal y cualquier número de ballot que haya utilizado en intentos previos de convertirse en líder, y lo envía en un mensaje `NEW_LEADER` a todos los procesos, incluyéndose a sí mismo, en conjunto con su cbal actual y la longitud de su arreglo msg (línea 27); este mensaje es análogo al mensaje `P1A` en Paxos.

Cuando un proceso se une a un ballot, promete ignorar cualquier invitación a un ballot no mayor, como en Paxos. Por ello, cuando un proceso recibe un mensaje NEW_LEADER$(b, cbal, msg_len)$ (línea 28), comprueba primero que el ballot propuesto es mayor al suyo, asegurándose no violar su promesa, y que el estado del proponente domina el suyo: o bien ha participado en un ballot mayor, o ha aceptado más mensajes en su mismo ballot (línea 30, donde $(a, b) \geq (c, d) \triangleq a > c \vee (a = c \wedge b \geq d)$).

```
25 function recover():
26     let b be a ballot number such that leader(b) = p_i, b > bal, and b is greater
         than any ballot number used by p_i in previous attempts to become a leader;
27     send NEW_LEADER(b, cbal, len(msg)) to P;

28 when received NEW_LEADER(b, cbal, msg_len) from p_j
29     pre: b > bal;
30     if (cbal, msg_len) ≥ (cbal, len(msg)) then
31         bal ← b;
32         status ← RECOVERING;
33         send NEW_LEADER_ACK(bal) to p_j;
34     else
35         let b' be a ballot number such that leader(b') = p_i, b' > b, and b' is
             greater than any ballot number used by p_i in previous attempts to
             become a leader;
36         send NEW_LEADER(b', cbal, len(msg)) to P;

37 when received {NEW_LEADER_ACK(b) | p_j ∈ Q} from a quorum Q
38     pre: status = RECOVERING ∧ bal = b;
39     cbal ← b;
40     send NEW_STATE(b, msg) to P \ {p_i};

41 when received NEW_STATE(b, msg) from p_j
42     pre: status = RECOVERING ∧ bal = b;
43     cbal ← b;
44     msg ← msg;
45     status ← FOLLOWER;
46     send NEW_STATE_ACK(b) to p_j;

47 when received NEW_STATE_ACK(b) from a set that together with p_i forms a
     quorum
48     pre: status = RECOVERING ∧ bal = b;
49     status ← LEADER;
50     next ← max{k | msg[k] ≠ ⊥};
51     forall {k | msg[k] ≠ ⊥} do send COMMIT(b, k, msg[k]) to P;
```

Figura 3.3: El protocolo en un proceso p_i: Caso de recuperación.

En este caso, establece su ballot a b, uniéndose así al ballot y prometiendo no aceptar ninguna propuesta de un ballot menor a b, y cambia su status a RECOVERING, lo cual ocasiona que el proceso deje de operar sobre cualquier mensaje broadcast,

ACCEPT, ACCEPT_ACK y COMMIT. Luego responde al futuro líder con un mensaje NEW_LEADER_ACK, análogo al mensaje P1B de Paxos (línea 33). Si la condición en la línea 30 no se satisface, el receptor sabe que tiene un estado más actualizado, e intenta convertirse en un líder él mismo con un número de ballot mayor al recibido (línea 36).

El futuro líder del ballot b espera hasta que reciba mensajes NEW_LEADER_ACK para el ballot propuesto de un quórum de procesos (línea 37). Este sabe entonces que un quórum se ha unido a su ballot y que su arreglo msg es al menos tan actual como el de cualquiera de sus votantes. El futuro líder establece entonces su cbal a b. Para finalizar el procedimiento de recuperación, necesita sincronizar el estado de sus votantes con el propio. Con este propósito, envía un mensaje NEW_STATE a todos los procesos con excepción de sí mismo el cual contiene el ballot y el nuevo estado (línea 40).

Al recibir un mensaje NEW_STATE (línea 41), un proceso sobreescribe su estado con el provisto por el líder, cambia su status a FOLLOWER, y establece cbal a b, registrando de este modo el hecho de que ha sincronizado su estado con el del líder de b. Nótese que el proceso no aceptará mensajes del nuevo líder hasta que reciba el mensaje NEW_STATE. El seguidor notifica entonces la recepción del nuevo estado al líder enviando un mensaje NEW_STATE_ACK (línea 46).

Al recibir mensajes NEW_STATE_ACK de un conjunto de procesos que junto con p_i forma un quórum (línea 47), $p_i = \texttt{leader}(b)$ sabe que un quórum de procesos comparte su estado en el ballot b y se establece entonces como su líder (línea 49), establece next al último slot no vacío en el arreglo msg, y notifica que todos los mensajes en msg pueden ser entregados de forma segura (línea 51), puesto que estos mensajes se encuentran presentes en una mayoría en el ballot b, y serán por lo tanto preservados a lo largo de ballots mayores.

3.5. Correctitud

El protocolo garantiza que el k-ésimo mensaje en la secuencia de mensajes entregados por un proceso puede sólo ser entregado en respuesta a un mensaje $\texttt{COMMIT}(\_, k, \_)$. Por lo tanto, para probar que todos los procesos entregan un prefijo completo de un orden global común, es suficiente probar que si $\texttt{COMMIT}(\_, k, m)$ y $\texttt{COMMIT}(\_, k, m')$ son mensajes enviados, entonces $m = m'$.

Para construir un argumento inductivo, hay una sutileza que se debe tener en cuenta. No se puede asumir que si $\texttt{COMMIT}(b, \_, \_)$ y $\texttt{COMMIT}(b', \_, \_)$ son mensajes enviados con $b < b'$, entonces $\texttt{COMMIT}(b, \_, \_)$ fue enviado antes que $\texttt{COMMIT}(b', \_, \_)$. De hecho, como diferentes procesos pueden participar en diferentes ballots al mismo tiempo, es posible que existan mensajes siendo aceptados en el ballot b mientras un ballot $b' > b$ sea operacional. En particular, un mensaje $\texttt{COMMIT}(b', \_, \_)$ puede ser enviado antes de que el mensaje $\texttt{COMMIT}(b, \_, \_)$ sea enviado, para ballots $b' > b$. Por lo tanto, nuestro protocolo debe garantizar que cualquier mensaje COMMIT en un ballot b' para un slot k no contradice ningún potencial COMMIT en ballots menores a b' para el mismo slot.

Decimos que un mensaje m es *commitable* en el ballot b para un slot k, si $\texttt{COMMIT}(b, k, m)$ es un mensaje enviado o aún puede ser enviado. Un mensaje COMMIT puede ser enviado durante el caso libre de fallos (línea 19) o durante el caso de recuperación (línea 51) del protocolo.

Un mensaje $\texttt{COMMIT}(b, k, m)$ es un mensaje enviado o aún puede ser enviado durante el caso libre de fallos del protocolo si existe un quórum de procesos $\mathcal{Q}$ tal que cada proceso $q \in \mathcal{Q}$ ha recibido y notificado un mensaje $\texttt{ACCEPT}(b, k, m)$ o aún puede hacerlo. Un proceso aún puede notificar un mensaje $\texttt{ACCEPT}(b, k, m)$ sólo si su ballot es menor o igual que b. Por lo tanto,

$$\begin{aligned}\texttt{acceptable}(b, k, m) \triangleq\ & \exists \mathcal{Q}.\ \mathcal{Q} \text{ es un quórum} \wedge \\ & (\forall q.\ q \in Q \Rightarrow q \text{ notificó } \texttt{ACCEPT}(b, k, m)\ \vee\ \mathsf{bal}_q \leq b)\end{aligned}$$

Un mensaje $\texttt{COMMIT}(b, k, m)$ es un mensaje enviado o aún puede ser enviado durante el caso de recuperación del protocolo si existe un quórum de procesos $\mathcal{Q}$ tal que cada proceso $q \in \mathcal{Q}$ es o bien el líder del ballot b, o ha recibido y notificado un mensaje $\texttt{NEW\_STATE}(b, msg)$ con $msg[k] = m$ o aún puede hacerlo. Un proceso aún puede notificar un mensaje $\texttt{NEW\_STATE}(b, msg)$ sólo si su ballot es menor o igual que b. Por lo tanto,

$$\begin{aligned}&\texttt{recoverable}(b, k, m) \triangleq \exists \mathcal{Q}, msg.\ \mathcal{Q} \text{ es un quórum} \wedge\ msg[k] = m \wedge \\ &\quad (\forall q.\ q \in Q \Rightarrow q = \texttt{leader}(b) \vee q \text{ notificó } \texttt{NEW\_STATE}(b, msg) \vee \mathsf{bal}_q \leq b)\end{aligned}$$

Finalmente, definimos

$$\texttt{commitable}(b, k, m) \triangleq \texttt{acceptable}(b, k, m)\ \vee\ \texttt{recoverable}(b, k, m)$$

Un paso clave en la prueba de corrección será demostrar que si se cumple $\texttt{commitable}(b, k, m)$ y $\texttt{COMMIT}(b', k, m')$ es un mensaje enviado con $b' > b$, entonces $m = m'$. Este hecho es formalizado por el Invariante 6.c.. Los ítems restantes fortalecen la hipótesis para hacer al invariante inductivo.

La Figura 3.4 muestra los invariantes principales mantenidos por nuestro protocolo, de los cuales probamos los más interesantes a continuación.

Prueba del Invariante 4. Por el Invariante 2 existe msg tal que msg es un prefijo de msg y $\texttt{NEW\_STATE}(b, msg)$ es un mensaje enviado. El Invariante 3 aplicado al líder del ballot b, y las líneas 50 y 10 implican que cualquier mensaje enviado $\texttt{ACCEPT}(b, k, \_)$ debe tener $k > \texttt{len}(msg)$. Por lo tanto, $\texttt{len}(msg) < k \leq \texttt{len}(\mathsf{msg})$. Entonces, por el Invariante 2, $\texttt{ACCEPT}(b, k, \mathsf{msg}[k])$ es un mensaje enviado, en cuyo caso el Invariante 1 implica que $\mathsf{msg}[k] = m$, como queríamos. □

Prueba del Invariante 5. La línea 27 y el hecho de que no pueden existir dos mensajes diferentes NEW_LEADER con el mismo número de ballot implican que cuando p_i envió el mensaje $\texttt{NEW\_LEADER}(b, cbal, msg\_len)$, este tenía $\mathsf{cbal} = cbal$ y $\texttt{len}(\mathsf{msg}) = msg\_len$. Para que p_i tenga $\mathsf{bal} = b$ y $\mathsf{status} = \text{RECOVERING}$, p_i debe haber recibido y notificado este mensaje.

1. a. Para cualquier mensajes $\texttt{ACCEPT}(b, k, m_1)$ y $\texttt{ACCEPT}(b, k, m_2)$, tenemos $m_1 = m_2$.
   b. Para cualquier mensajes $\texttt{NEW\_STATE}(b, m_1)$ y $\texttt{NEW\_STATE}(b, m_2)$, tenemos $m_1 = m_2$.
   c. Para cualquier mensajes $\texttt{COMMIT}(b, k, m_1)$ y $\texttt{COMMIT}(b, k, m_2)$, tenemos $m_1 = m_2$.
2. Si en un proceso tenemos $\mathsf{cbal} = b$, entonces existe msg tal que msg es un prefijo de msg, $\texttt{NEW\_STATE}(b, msg)$ es un mensaje enviado, y para cada $\texttt{len}(msg) < k \leq \texttt{len}(\mathsf{msg})$, $\texttt{ACCEPT}(b, k, \mathsf{msg}[k])$ es un mensaje enviado.
3. Si en un proceso tenemos $\mathsf{cbal} = b$ y $\texttt{NEW\_STATE}(b, msg)$ es un mensaje enviado, entonces msg es un prefijo de msg.
4. Si en un proceso tenemos $\mathsf{cbal} = b$, $\texttt{len}(\mathsf{msg}) \geq k$ y $\texttt{ACCEPT}(b, k, m)$ es un mensaje enviado, entonces $\mathsf{msg}[k] = m$.
5. Si un proceso envió $\texttt{NEW\_LEADER}(b, cbal, msg\_len)$, $\mathsf{bal} = b$ y $\mathsf{status} = \text{RECOVERING}$, entonces $\mathsf{cbal} \geq cbal$ y $\texttt{len}(\mathsf{msg}) = msg\_len$.
6. Si $\texttt{commitable}(b, k, m)$ se cumple y $b' > b$, entonces
   a. Si en un proceso tenemos $\mathsf{cbal} = b'$, entonces $\mathsf{msg}[k] = m$.
   b. Si en un proceso tenemos $\mathsf{cbal} = b'$ y $\mathsf{status} = \text{LEADER}$, entonces $\mathsf{next} \geq k$.
   c. Para cualquier mensaje $\texttt{COMMIT}(b', k, m')$, tenemos $m' = m$.
   d. Para cualquier mensaje $\texttt{ACCEPT\_ACK}(b', k')$, tenemos $k' > k$.
   e. Para cualquier mensaje $\texttt{ACCEPT}(b', k', \_)$, tenemos $k' > k$.
   f. Para cualquier mensaje $\texttt{NEW\_STATE}(b', msg)$, tenemos $msg[k] = m$.
7. Para cualquier mensajes $\texttt{COMMIT}(\_, k, m_1)$ y $\texttt{COMMIT}(\_, k, m_2)$, tenemos $m_1 = m_2$.
8. Si m es el k-ésimo mensaje (0-indexado) entregado por algún proceso, entonces existe un ballot b tal que $\texttt{COMMIT}(b, k, m)$ es un mensaje enviado.
9. Si en un proceso tenemos $\mathsf{msg}[k] = m$ para algún $m \neq \bot$, o $\texttt{ACCEPT}(\_, \_, m)$ es un mensaje enviado, o $\texttt{COMMIT}(\_, \_, m)$ es un mensaje enviado, o $\texttt{NEW\_STATE}(\_, msg)$ es un mensaje enviado con $msg[k] = m$ para algún $m \neq \bot$, entonces m fue previamente enviado.
10. En cualquier proceso, todos los mensajes en el arreglo msg son distintos.

Figura 3.4: Invariantes principales mantenidos por nuestro protocolo. Sea $\texttt{len}(msg) \triangleq max\{k \mid msg[k] \neq \bot\}$.

En ese punto, este debe haber tenido $(\mathsf{cbal}, \texttt{len}(\mathsf{msg})) \geq (cbal, msg\_len)$, pero la línea 30 asegura que $(\mathsf{cbal}, \texttt{len}(\mathsf{msg})) \leq (cbal, msg\_len)$ y por lo tanto $\mathsf{cbal} = cbal$ y $\texttt{len}(\mathsf{msg}) = msg\_len$ justo luego de notificar el mensaje. Como p_i establece su status

a RECOVERING, esto ocasiona que el proceso deje de operar sobre cualquier mensaje `broadcast`, `ACCEPT` y `COMMIT` y, por lo tanto, msg permanece sin modificaciones. Como los números de cbal son no decrecientes, entonces cbal $\geq cbal$. □

Prueba del Invariante 6. Probamos el invariante por inducción en la longitud de la ejecución del protocolo. Inicialmente cbal $= 0$ para todos los procesos y no existen mensajes enviados, por lo que la afirmación es trivialmente verdadera. Para el paso inductivo, supóngase que `commitable`(b, k, m) se cumple. El resultado sigue trivialmente de la hipótesis inductiva para todas las transiciones excepto aquellas en las líneas 37 y 47.

Considérese entonces la transición en la línea 37 por un proceso p_i la cual gestiona la recepción de mensajes `NEW_LEADER_ACK`(b') de un quórum $\mathcal{Q}$ en respuesta a un mensaje `NEW_LEADER`$(b', cbal, msg\_len)$. La hipótesis implica que o bien `acceptable`(b, k, m) o `recoverable`(b, k, m) se cumple.

Si `acceptable`(b, k, m) se cumple, entonces existe un quórum $\mathcal{Q}'$ tal que cada proceso ha recibido un mensaje `ACCEPT`(b, k, m) y ha respondido con `ACCEPT_ACK`(b, k) o este tiene bal $\leq b$. Sea $q \in \mathcal{Q} \cap \mathcal{Q}'$. Dado que $q \in \mathcal{Q}'$ ha recibido y notificado `NEW_LEADER`$(b', cbal, msg\_len)$, este tiene bal $\geq b' > b$. Entonces, q debe haber enviado su mensaje `ACCEPT_ACK` antes del mensaje `NEW_LEADER_ACK`. Por la precondición en la línea 14 y el hecho de que si status $\in \{$LEADER, FOLLOWER$\}$ entonces bal $=$ cbal, cuando q envió el mensaje `ACCEPT_ACK`, este debe haber tenido cbal $= b$. Entonces, cuando q envió el mensaje `NEW_LEADER_ACK` este tenía cbal $\geq b$. En consecuencia, $cbal \geq b$.

Si, por otro lado, `recoverable`(b, k, m) se cumple, entonces existe un quórum de procesos $\mathcal{Q}'$ cuyos procesos p son tales que $p =$ `leader`(b), p ha recibido y notificado `NEW_STATE`(b, msg) con $msg[k] = m$, o este tiene bal $\leq b$. Sea $q \in \mathcal{Q} \cap \mathcal{Q}'$. Puesto que $q \in \mathcal{Q}'$ ha recibido y notificado `NEW_LEADER`$(b', cbal, msg\_len)$, este tiene bal $\geq b' > b$. Entonces, q debe haber enviado su mensaje `NEW_STATE` o `NEW_STATE_ACK` antes del mensaje `NEW_LEADER_ACK`. Las líneas 39 y 43 implican que cbal $= b$ cuando q envió este mensaje. Entonces, cuando q envió el mensaje `NEW_LEADER_ACK` este tenía cbal $\geq b$. En consecuencia, $cbal \geq b$.

Puesto que p_i envió `NEW_LEADER`$(b', cbal, msg\_len)$, y por la línea 38, bal $= b'$ y status $=$ RECOVERING, el Invariante 5 implica que en p_i tenemos `len`(msg) $= msg\_len$ y cbal $\geq cbal \geq b$.

Si cbal $> b$, la hipótesis inductiva implica que msg$[k] = m$. Supóngase entonces que cbal $= b$. Por la línea 30, cuando q notificó `NEW_LEADER`$(b', cbal, msg\_len)$ este debe haber tenido `len`(msg) $\leq msg\_len$. En el caso en que `acceptable`(b, k, m) se cumple, q ha recibido y notificado `ACCEPT`(b, k, m). Entonces, cuando q envió el mensaje `NEW_LEADER_ACK` este tenía `len`(msg) $\geq k$ y por lo tanto $msg\_len \geq k$. Entonces, por el Invariante 4, debemos tener msg$[k] = m$. Si, por otro lado, `recoverable`(b, k, m) se cumple, entonces existe msg tal que $msg[k] = m$ y `NEW_STATE`(b, msg) es un mensaje enviado, en cuyo caso, por el Invariante 3, msg es un prefijo de msg y por lo tanto msg$[k] = m$.

En consecuencia, el mensaje `NEW_STATE`(b', msg') enviado durante la transición actual tiene $msg'[k] =$ msg$[k] = m$. Ningún mensaje `COMMIT`, `ACCEPT_ACK` o `ACCEPT`

es enviado durante la transición actual.

Considérese ahora la transición en la línea 47 por un proceso p_i la cual gestiona la recepción de mensajes NEW_STATE_ACK(b') de un conjunto que junto con p_i forman un quórum en respuesta a un mensaje NEW_STATE(b', msg), y por la cual p_i se convierte en el líder del ballot b'. Como se ha mostrado arriba, msg$[k] = m$ justo luego de la transición en la línea 37. Como p_i no modificó su status = RECOVERING, desde ese punto hasta que p_i recibió los mensajes NEW_STATE_ACK, msg permanece sin modificaciones. Por lo tanto, la línea 50 implica que next $\geq k$ y que el mensaje COMMIT(b', k, m') enviado durante la transición actual tiene $m' = m$. Ningún mensaje ACCEPT_ACK, ACCEPT o NEW_STATE es enviado durante la transición actual. □

Prueba del Invariante 7. Sean COMMIT(b_1, k, m_1) y COMMIT(b_2, k, m_2) mensajes enviados. Supóngase sin pérdida de generalidad que $b_1 \leq b_2$. Si $b_1 = b_2$, entonces por el Invariante 1, debemos tener $m_1 = m_2$. Supóngase ahora que $b_1 < b_2$. Nótese que si COMMIT(b_1, k, m_1) es un mensaje enviado, entonces commitable(b_1, k, m_1) se cumple. Por lo tanto, por el Invariante 6, debemos tener $m_1 = m_2$. □

Teorema 3. *El algoritmo satisface Integridad.*

Teorema 4. *El algoritmo satisface No duplicación.*

Probamos a continuación que nuestro protocolo satisface Orden confiable: existe un orden total $\preceq$ en el conjunto de todos los mensajes que han sido entregados en una ejecución por cualquier proceso, tal que si un proceso p_i entrega un mensaje m, entonces para todos los mensajes m', $m' \prec m$ implica que p_i entrega m' antes que m. En otras palabras, todos los procesos entregan los mensajes en el mismo orden sin huecos.

Fíjese una ejecución r del algoritmo y sea $\mathcal{M}$ el conjunto de mensajes entregados en r. Defínase un orden total $\preceq_i$ en el conjunto $\mathcal{M}_i$ de mensajes entregados por un proceso p_i en r tal que para cada $m, m' \in \mathcal{M}_i$, $m \preceq_i m'$ si y sólo si p_i entrega m antes que m'. Escribiremos μ_i para denotar la secuencia sobre $\mathcal{M}_i$ inducida por $\preceq_i$ y $\mu_i \lfloor k$ para denotar el prefijo de μ_i de longitud k.

Lema 2. *Para cualquier procesos p_i y p_j, y prefijos μ'_j de μ_j, si $|\mu'_j| \leq |\mu_i|$, entonces μ'_j es un prefijo de μ_i.*

Demostración. Sea $0 \leq k < |\mu'_j| \leq |\mu_i|$. El Invariante 8 implica que existen ballots b, b' tales que COMMIT($b, k, \mu'_j[k]$) y COMMIT($b', k, \mu_i[k]$) son mensajes enviados. Por el Invariante 7, debe ser $\mu'_j[k] = \mu_i[k]$. □

Probamos a continuación que el Lema 2 implica la existencia de la relación de orden total estipulada por la propiedad de Orden confiable. Sea $\preceq \triangleq \bigcup_{p_i} \preceq_i$.

Lema 3. *La relación $\preceq$ es antisimétrica.*

Demostración. Asúmase por contradicción que existen mensajes m y m' tales que $m \preceq m'$, $m' \preceq m$, y $m \neq m'$. Entonces, existen procesos p_i y p_j tales que $m \prec_i m'$ y $m' \prec_j m$. Sea ℓ el índice de m' en μ_i. Entonces, por el Lema 2, o bien $|\mu_j| < \ell$

o $\mu_j\lfloor\ell = \mu_i\lfloor\ell$. Si $|\mu_j| < \ell$, entonces por el Lema 2, $m' \prec_j m$ implica que $m' \prec_i m$ contradiciendo la hipótesis $m \prec_i m'$. Si $\mu_j\lfloor\ell = \mu_i\lfloor\ell$, entonces $\mu_j[\ell] = m'$. Dado que $m \prec_i m'$, esto implica que $m \prec_j m'$ contradiciendo la hipótesis $m' \prec_j m$. Por ello, concluimos que $\preceq$ es antisimétrica. □

Lema 4. *La relación $\preceq$ es transitiva.*

Demostración. Asúmase por contradicción que existen mensajes m, m' y m'' en $\mathcal{M}$ tales que $m \prec m'$, $m' \prec m''$ y $m'' \preceq m$. Entonces, existen procesos p_i, p_j y p_k, tales que p_i entrega m y m', p_j entrega m' y m'', y p_k entrega m y m'', y se cumple

$$m \prec_i m' \tag{3.1}$$

$$m' \prec_j m'' \tag{3.2}$$

$$m = m'' \vee m'' \prec_k m \tag{3.3}$$

Por el Lema 2, 3.1 y 3.2 implican que m' tiene el mismo índice en μ_i y μ_j. Por ello, el índice de m en μ_i es menor que el índice de m'' en μ_j, que por el Lema 2 implica que $m \prec_j m''$. En consecuencia, $m \neq m''$, y por 3.3, $m'' \prec_k m$. Sin embargo, puesto que $m \prec_j m''$, el Lema 2 implica que $m \prec_k m''$, lo cual es una contradicción. □

Lema 5. *La relación $\preceq$ es conexa.*

Demostración. Considérese $m, m' \in \mathcal{M}$. Sea p_i un proceso que entrega m y p_j un proceso que entrega m'. Si el índice de m en μ_i es menor que el índice de m' en μ_j, y por el Lema 2, m tiene el mismo índice en μ_i y μ_j, obtenemos que $m \prec_j m'$. Por lo tanto, $m \preceq m'$. De otro modo, por el mismo argumento, obtenemos que $m' \prec_i m$, y en consecuencia $m' \preceq m$. □

Lema 6. *La relación $\preceq$ es un orden total sobre $\mathcal{M}$.*

Teorema 5. *El algoritmo satisface Orden confiable.*

Demostración. Se ha mostrado que los procesos entregan los mensajes de acuerdo a $\preceq$. Necesitamos ahora probar que si un proceso p_i entrega un mensaje m, entonces para todo m' tal que $m' \prec m$, p_i entrega m' y $m' \prec_i m$. Considérense entonces mensajes m y m' tales que p_i entrega m y $m' \prec m$. Entonces, existe un proceso p_j tal que p_j entrega m y m', y $m' \prec_j m$. Por el Lema 2, esto implica que p_i entrega m' y $m' \prec_i m$, como queríamos demostrar. □

Capítulo 4

Vertical Paxos

4.1. Introducción

La mayoría de las soluciones por replicación requieren $2f + 1$ procesos para tolerar f fallos crash-stop. Esto es costoso: si la información se persiste en todos los procesos, sólo $f + 1$ son necesarios para garantizar durabilidad. Puesto que en este caso, el fallo de sólo un proceso bloquearía el procesamiento de mensajes, para recuperarse se requiere reconfigurar el sistema, esto es, cambiar la membresía para reemplazar los procesos fallidos por procesos frescos.

Los procesos necesitan acordar la próxima configuración que se reduce a consenso, lo cual nuevamente requiere $2f + 1$ procesos. Una solución consiste en utilizar un servicio de configuración independiente con $2f+1$ procesos únicamente para realizar consenso en la configuración y sólo $f + 1$ para replicar la información relevante.

4.2. Modelo de sistema

Consideramos un sistema asíncrono con paso de mensajes compuesto por un conjunto de procesos $\mathcal{P}$ crash-stop, es decir, fallan deteniendo permanentemente su ejecución. Un proceso es *correcto* si nunca falla. Asumimos que los procesos se encuentran conectados por enlaces confiables FIFO: los mensajes se entregan en orden FIFO, y se garantiza que los mensajes entre procesos correctos son eventualmente entregados. Existe un grupo de procesos cuya membresía puede cambiar a lo largo del tiempo moviéndose a través de una secuencia de *configuraciones*. Las configuraciones se almacenan en un *configuration service* (CS) externo que por simplicidad asumimos ser un proceso confiable.

4.3. El problema

Consideramos el problema de implementar la abstracción broadcast atómico en el sistema descripto arriba. La abstracción provee dos primitivas:

`broadcast`(m): permite a un proceso enviar un mensaje de aplicación m a todos los procesos en $\mathcal{P}$.

`deliver`(m): permite a un proceso entregar un mensaje recibido m a su capa de aplicación.

Por simplicidad, asumimos que los mensajes enviados por los procesos son únicos. La abstracción garantiza que los procesos entregan un prefijo completo de una secuencia global común de mensajes. Un algoritmo es una implementación correcta de broadcast atómico si cada ejecución del mismo satisface:

Integridad: Cada mensaje entregado fue previamente enviado.

No duplicación: Ningún mensaje se entrega más de una vez al mismo proceso.

Orden confiable: Todos los procesos entregan un prefijo completo de un orden global común sobre los mensajes.

4.4. El protocolo

```
1  epoch ← 0 ∈ ℤ;
2  cepoch ← 0 ∈ ℤ;
3  next ← −1 ∈ ℤ;
4  last_delivered ← −1 ∈ ℤ;
5  members ∈ 2^𝒫;
6  msg[ ] ∈ ℕ → {⊥} ∪ ℳ;
7  status ∈ {LEADER, FOLLOWER, RECONFIGURING};
8  pmembers ∈ 2^𝒫;
9  pepoch, new_epoch ∈ ℤ;
10 probing, initialized ∈ {TRUE, FALSE};
```

Figura 4.1: Variables locales de un proceso

Visión general y estado de los procesos. Esta sección provee una visión general del protocolo e introduce las variables de estado utilizadas por cada proceso, dadas en la Figura 4.1.

Consideramos un grupo de procesos cuya membresía puede cambiar a lo largo del tiempo moviéndose a través de una secuencia de configuraciones. Reconfigurar es el proceso de cambiar la configuración de este grupo. En nuestro protocolo, una reconfiguración es iniciada por un proceso cuando sospecha que otro proceso ha fallado. Una configuración es entonces un par $\langle e, \mathcal{M} \rangle$ donde e es la época que identifica la configuración y $\mathcal{M} \in 2^{\mathcal{P}}$ es el conjunto de procesos que pertenecen al grupo. En la práctica, el configuration service puede ser implementado utilizando replicación Paxos-like. El configuration service provee tres operaciones. La operación `compare_and_swap`($e, \langle e', \mathcal{M} \rangle$) es exitosa si la época de la última configuración almacenada es e; en este caso, este almacena la configuración provista con $e' > e$. Las operaciones `get_last`() y `get`(e) retornan la última configuración del grupo y la configuración del grupo asociada a una época dada e, respectivamente.

Nuestro protocolo utiliza un enfoque *basado en un líder*. Inicialmente, los procesos intentan elegir un líder. Un líder gana una elección si su estado contiene todos los mensajes previamente aceptados por los miembros de su configuración, los cuales persiste en una variable `members`, obteniendo de esta manera el derecho a proponer mensajes. Una vez electo, el líder ha comenzado un período de ejecución llamado *época*. En cualquier momento, cada proceso participa en una única época alojada en la variable epoch.

Los procesos transmiten mensajes utilizando la primitiva `broadcast`. El líder recibe los mensajes entrantes y los coloca en el arreglo msg, añadiendolos al final del mismo, y dictando de este modo el orden en que estos deben ser entregados. Con este fin, registra el último slot no vacío en el arreglo en una variable next. Para tolerancia a fallos, el ordenamiento en los mensajes necesita ser replicado por cada miembro de la configuración, que estipula una ronda de notificaciones desde los seguidores hacia el líder. Para ello, el líder propaga los mensajes a los seguidores.

Al recibir una propuesta, los seguidores la almacenan en su copia local del arreglo msg y notifican su recepción al líder, aceptándola de este modo. Si todos los seguidores en la configuración aceptan el mensaje, se garantiza su supervivencia a cualquier fallo tolerado, y puede ser entregado en los procesos participantes utilizando la primitiva `deliver`. Los procesos persisten la posición del último mensaje entregado en el arreglo msg en una variable last_delivered.

Cuando se sospecha que un miembro de la configuración ha fallado, los procesos ejecutan un protocolo de reconfiguración para reemplazar los procesos fallidos por procesos frescos, acordar un estado consistente común antes de reanudar la operación normal, y también para establecer un nuevo líder capaz de transmitir nuevos mensajes. El propósito del protocolo de recuperación es elegir un líder cuyo estado contiene todos los mensajes aceptados en épocas previas. Una variable status registra si el proceso es un LEADER, un FOLLOWER o se encuentra en un estado especial RECONFIGURING utilizado durante reconfiguraciones.

Un proceso realizando una reconfiguración primero *prueba* las configuraciones previas para determinar qué procesos aún se encuentran vivos y para hallar un proceso cuyo estado contenga todos los mensajes previamente aceptados, que servirá como el nuevo líder. Una variable probing $\in \{\texttt{TRUE}, \texttt{FALSE}\}$ indica este estado. La época siendo probada y los miembros de su configuración son almacenados en las variables pepoch y pmembers, respectivamente.

Para garantizar un estado consistente, se requiere que los seguidores sincronicen su estado con el futuro líder antes de que empiecen a aceptar mensajes. Cuando esto ocurre, decimos que el proceso ha sido *inicializado*. Una variable `initialized` $\in \{\texttt{TRUE}, \texttt{FALSE}\}$ en un proceso registra si el proceso ha sido inicializado alguna vez. Nuestro protocolo garantiza que una configuración es operacional, esto es, comienza a aceptar mensajes, sólo luego de que todos sus miembros han sido inicializados.

La fase de prueba es complicada por el hecho de que puede existir una serie de intentos fallidos de reconfiguración, donde un fallo ocurre antes de que un futuro líder inicialice todos sus seguidores. Por lo tanto, la prueba requiere atravesar las épocas desde la actual hacia atrás, ignorando épocas que no fueron operativas. La fase de prueba selecciona como el líder de la nueva época, indicada por la varia-

ble new_epoch, el primer proceso inicializado que encuentra durante este recorrido; podemos demostrar que este proceso garantiza las propiedades deseadas.

Como pueden existir múltiples intentos fallidos de reconfiguración, utilizamos una variable de época adicional cepoch para representar la última época en la cual un proceso ha sincronizado su estado con aquel del futuro líder.

```
11 when received broadcast(m)
12     pre: status = LEADER;
13     if ∀k. msg[k] ≠ m then
14         next ← next + 1;
15         msg[next] ← m;
16         send ACCEPT(epoch, next, m) to members;
17 when received ACCEPT(e, k, m) from p_j
18     pre: status ∈ {LEADER, FOLLOWER} ∧ epoch = e;
19     msg[k] ← m;
20     send ACCEPT_ACK(e, k) to p_j;
21 when received ACCEPT_ACK(e, k) from every p_j ∈ members
22     pre: status = LEADER ∧ epoch = e;
23     send COMMIT(e, k, msg[k]) to members;
24 when received COMMIT(e, k, m)
25     pre: status ∈ {LEADER, FOLLOWER} ∧ epoch = e ∧ k = last_delivered + 1;
26     last_delivered ← k;
27     msg[k] ← m;
28     deliver(msg[k]);
```

Figura 4.2: El protocolo en un proceso p_i: caso libre de fallos

Caso libre de fallos. En el modo normal del protocolo, un único líder propone una secuencia de mensajes a sus seguidores con el objetivo de replicarlos y garantizar la durabilidad de los mensajes entregados y su orden. Su código se muestra en la Figura 4.2.

Para transmitir un mensaje m, un proceso lo envía en un mensaje `broadcast`. Un proceso p_i opera sobre el mensaje sólo cuando es el líder (línea 12). Al recibir `broadcast`(m), el líder p_i comprueba si ha recibido anteriormente este mensaje con el propósito de evitar entregar el mismo mensaje más de una vez (línea 13). En caso contrario, este añade el mensaje a la secuencia ordenada de mensajes msg e intenta replicarlo en todos sus seguidores. Con este fin, envía un mensaje `ACCEPT` al grupo, incluyéndose a sí mismo por uniformidad (línea 16). El mensaje porta la época del líder, la posición del mensaje en el arreglo msg y el mensaje m.

Un proceso opera sobre el mensaje `ACCEPT` sólo si participa en la época correspondiente (línea 18). Este almacena el mensaje en su copia local del arreglo msg y luego envía un mensaje `ACCEPT_ACK` al líder de e (línea 20). El mensaje porta la

época y el número de slot correspondiente al mensaje que está siendo notificado. Esto confirma que el proceso ha aceptado el mensaje.

El líder de la época e actúa una vez que recibe mensajes `ACCEPT_ACK` para la época e y slot k de todos los miembros de su configuración (línea 21), en cuyo caso el mensaje ha estado presente en el arreglo de todos los miembros de una época en el mismo slot, y por lo tanto sobrevivirá cualquier fallo tolerado. En este caso, el líder notifica a todos los miembros de su configuración que el mensaje puede ser entregado de forma segura a través de un mensaje `COMMIT` (línea 23).

Al recibir un mensaje `COMMIT` (línea 24), el proceso actualiza last_delivered para registrar el prefijo de la secuencia global común de mensajes que ha entregado y actualiza su arreglo. Los mensajes son entregados en orden creciente de número de slot sin huecos como lo implica la línea 25. Finalmente, el proceso entrega el mensaje a su capa de aplicación (línea 28).

Reconfiguración. Explicamos a continuación cómo el protocolo trata los fallos ejecutando el protocolo de reconfiguración en la Figura 4.3.

El objetivo del protocolo de reconfiguración es doble: primero, elegir un líder quien convenza a todos los miembros de una configuración de participar en su época y cuyo estado contiene todos los mensajes aceptados, y segundo, garantizar que antes de que un seguidor empiece a aceptar propuestas del nuevo líder, este ha sincronizado su estado con aquel del líder, asegurando de esta manera que los procesos permanecen en sincronía.

Describimos ahora el procedimiento de reconfiguración en detalle. Cuando un proceso p_i sospecha que un miembro de su configuración ha fallado, este inicia una reconfiguración ejecutando la función `reconfigure`() (línea 29). El proceso escoge un número de época new_epoch mayor que la época almacenada en el configuration service y luego inicia la fase de prueba, como lo indica el flag probing. El proceso p_i registra la época siendo probada en pepoch y la membresía de esta época en pmembers. El proceso inicializa estas variables cuando obtiene la información sobre la época actual del configuration service (línea 32). Este luego envía un mensaje `PROBE` a los miembros de la configuración siendo actualmente probada, solicitándoles se unan a la nueva época new_epoch (línea 34).

Al recibir un mensaje `PROBE`(e) (línea 35), un proceso comprueba primero que la época propuesta sea mayor a la mayor época a la que se le haya solicitado unirse, que el proceso almacena en epoch. En este caso, el proceso establece epoch a e uniéndose así a la nueva época y cambia su status a RECONFIGURING, lo cual ocasiona que el proceso deje de operar sobre cualquier mensaje `broadcast`, `ACCEPT`, `ACCEPT_ACK` y `COMMIT`. Luego responde a p_j con un mensaje `PROBE_ACK`, el cual indica si ha sido previamente inicializado (línea 39).

Si p_i halla un proceso que ha sido previamente inicializado (línea 40) y puede en consecuencia servir como el nuevo líder, p_i termina de probar (línea 42). A continuación, el proceso p_i computa la membresía de la nueva configuración utilizando la función `compute_membership` (línea 43) provista por el configuration service. No prescribimos una implementación particular para esta función, excepto que la nueva membresía debe contener al futuro líder p_j y sólo puede contener procesos que

```
29 function reconfigure():
30     pre: probing = FALSE;
31     probing ← TRUE;
32     ⟨pepoch, pmembers⟩ ← get_last() at CS;
33     new_epoch ← pepoch + 1;
34     send PROBE(new_epoch) to pmembers;
35 when received PROBE(e) from p_j
36     pre: e > epoch;
37     epoch ← e;
38     status ← RECONFIGURING;
39     send PROBE_ACK(initialized, e) to p_j;
40 when received PROBE_ACK(TRUE, new_epoch) from p_j
41     pre: probing = TRUE;
42     probing ← FALSE;
43     var M ← compute_membership();
44     if compare_and_swap(new_epoch − 1, ⟨new_epoch, M⟩) at CS then
45         send NEW_CONFIG(new_epoch, M) to p_j;
46 non-deterministically when received PROBE_ACK(FALSE, new_epoch) from
     p_j ∈ pmembers and no PROBE_ACK(TRUE, new_epoch)
47     pre: probing = TRUE;
48     pepoch ← pepoch − 1;
49     pmembers ← get(pepoch) at CS;
50     send PROBE(new_epoch) to pmembers;
51 when received NEW_CONFIG(e, M) from p_j
52     pre: status = RECONFIGURING ∧ epoch = e;
53     cepoch ← e;
54     members ← M;
55     send NEW_STATE(e, msg) to members \ {p_i};
56 when received NEW_STATE(e, msg) from p_j
57     pre: e ≥ epoch;
58     epoch ← e;
59     cepoch ← e;
60     msg ← msg;
61     initialized ← TRUE;
62     status ← FOLLOWER;
63     send NEW_STATE_ACK(e) to p_j;
64 when received NEW_STATE_ACK(e) from members \ {p_i}
65     pre: status = RECONFIGURING ∧ epoch = e;
66     status ← LEADER;
67     next ← max{k | msg[k] ≠ ⊥};
68     forall {k | msg[k] ≠ ⊥} do send COMMIT(e, k, msg[k]) to members;
```

Figura 4.3: El protocolo en un proceso p_i: reconfiguración

respondieron a la prueba o procesos frescos. Estos últimos se pueden agregar para alcanzar el nivel deseado de tolerancia a fallos. Una vez que la nueva configuración es computada, p_i intenta almacenarla en el configuration service utilizando la operación `compare_and_swap` (línea 44). La operación es exitosa sólo si la época actual en el configuration service es la época desde la cual p_i comenzó a probar, lo cual implica que ninguna reconfiguración concurrente ocurrió mientras p_i se encontraba probando. En este caso, p_i envía un mensaje con la nueva configuración al futuro líder (línea 45).

Si p_i no halla un tal proceso en la época pepoch y recibe al menos una respuesta `PROBE_ACK` de un proceso que no ha sido inicializado (línea 46), p_i puede concluir que la época pepoch no es operacional y nunca lo será, pues ha convencido al menos uno de sus miembros de unirse a la nueva época. En este caso, p_i comienza a probar la época precedente (línea 48). Puesto que ningún mensaje puede haber sido aceptado por todos en la época pepoch, escoger un nuevo líder de una época anterior no omitirá ningún mensaje aceptado.

Cuando el futuro líder recibe el mensaje `NEW_CONFIG` (línea 51), este establece cepoch a e y persiste los miembros de la configuración actual. Para finalizar el procedimiento de reconfiguración, necesita sincronizar el estado de los miembros con el propio. Con este propósito, envía un mensaje `NEW_STATE` a todos los procesos con excepción de sí mismo, el cual contiene la época y el nuevo estado (línea 55).

Al recibir un mensaje `NEW_STATE` (línea 56), un proceso sobreescribe su estado con el provisto por el líder, cambia su status a FOLLOWER, y establece initialized a `TRUE`. Como parte de la actualización del estado, el proceso también actualiza su epoch y cepoch a la nueva época, registrando de este modo el hecho de que ha sincronizado su estado con el del líder de e. Nótese que el proceso no aceptará mensajes del nuevo líder hasta que reciba el mensaje `NEW_STATE`. El seguidor notifica entonces la recepción del nuevo estado al líder enviando un mensaje `NEW_STATE_ACK` (línea 63).

Al recibir mensajes `NEW_STATE_ACK` de todos los miembros de su configuración excepto de sí mismo, $p_i = \texttt{leader}(e)$ sabe que todos los procesos comparten su estado en la época e y se establece entonces como su líder (línea 66), establece next al último slot no vacío en el arreglo msg, y notifica que todos los mensajes en msg pueden ser entregados de forma segura (línea 68), puesto que estos mensajes se encuentran presentes en todos los miembros de la configuración en la época e, y serán por lo tanto preservados a lo largo de épocas mayores.

4.5. Correctitud

La Figura 4.4 muestra los invariantes principales mantenidos por nuestro protocolo, de los cuales probamos los más interesantes a continuación.

Prueba del Invariante 2. Cuando p_i recibe y notifica $\texttt{PROBE}(e)$ este establece epoch a e. Por lo tanto, como el protocolo garantiza trivialmente que epoch nunca decrece, a partir de ese punto siempre tendrá epoch $\geq e$. Por las comprobaciones en las líneas 18, 52 y 57, p_i nunca será capaz de procesar mensajes $\texttt{ACCEPT}(e', \_, \_)$, $\texttt{NEW\_CONFIG}(e', \_)$ o $\texttt{NEW\_STATE}(e', \_)$ con $e' < e$, y por ello, nunca enviará ningún

1. a. Para cualquier mensajes $\texttt{ACCEPT}(e, k, m_1)$ y $\texttt{ACCEPT}(e, k, m_2)$, tenemos $m_1 = m_2$.

   b. Para cualquier mensajes $\texttt{COMMIT}(e, k, m_1)$ y $\texttt{COMMIT}(e, k, m_2)$, tenemos $m_1 = m_2$.

2. Luego de que un proceso recibe $\texttt{PROBE}(e)$ y responde con $\texttt{PROBE\_ACK}(\_, e)$, este nunca enviará $\texttt{ACCEPT\_ACK}(e', \_)$, $\texttt{NEW\_STATE}(e', \_)$ o $\texttt{NEW\_STATE\_ACK}(e')$ con $e' < e$.

3. Asúmase que todos los miembros en e tienen $\mathsf{cepoch} = e$. Sea p_i un miembro de e' tal que $e' < e$. Si p_i no es un miembro de e entonces p_i no puede ser un miembro de ningúna época $e'' > e$.

4. Si p_i ha enviado $\texttt{NEW\_STATE}(e, msg)$ con $msg[k] = m$, o $\texttt{ACCEPT}(e, k, m)$, o ha recibido y notificado algunos de estos mensajes, entonces luego de esto y mientras $\mathsf{cepoch} = e$ en p_i, tenemos $\mathsf{msg}[k] = m$.

5. Supóngase que $\texttt{COMMIT}(e, k, m)$ es un mensaje enviado. Si en un proceso p_i tenemos $\mathsf{cepoch} = e' > e$, entonces $\mathsf{msg}[k] = m$.

6. Para cualquier mensajes $\texttt{COMMIT}(\_, k, m_1)$ y $\texttt{COMMIT}(\_, k, m_2)$, tenemos $m_1 = m_2$.

7. Si m es el k-ésimo mensaje (0-indexado) entregado por algún proceso, entonces existe una época e tal que $\texttt{COMMIT}(e, k, m)$ es un mensaje enviado.

8. Si en un proceso tenemos $\mathsf{msg}[k] = m$ para algún $m \neq \perp$, o $\texttt{ACCEPT}(\_, \_, m)$ es un mensaje enviado, o $\texttt{COMMIT}(\_, \_, m)$ es un mensaje enviado, o $\texttt{NEW\_STATE}(\_, msg)$ es un mensaje enviado con $msg[k] = m$ para algún $m \neq \perp$, entonces m fue previamente enviado.

9. En cualquier proceso, todos los mensajes en el arreglo msg son distintos.

Figura 4.4: Invariantes principales mantenidos por nuestro protocolo.

mensaje $\texttt{ACCEPT\_ACK}(e', \_)$, $\texttt{NEW\_STATE}(e', \_)$ o $\texttt{NEW\_STATE\_ACK}(e')$ con $e' < e$ luego de enviar $\texttt{PROBE\_ACK}(\_, e)$. □

Prueba del Invariante 3. Probamos el invariante por inducción en e''. Asúmase que el invariante se cumple para cada $e'' < e^*$. Lo probamos ahora para $e'' = e^*$. Los miembros de e^* se computan en la línea 43 por un proceso p_r utilizando la función `compute_membership`, que retorna o bien procesos frescos o procesos que han respondido a la prueba de p_r. Puesto que p_i era un miembro de $e' < e^*$, este no es fresco. El proceso p_r comienza probando la época $e^* - 1$ y termina al recibir un mensaje $\texttt{PROBE\_ACK}(\texttt{TRUE}, e^*)$. Por la hipótesis inductiva, p_i no es miembro de ninguna época desde $e^* - 1$ hasta $e + 1$. Por lo tanto, si la prueba se detiene antes de alcanzar e, entonces p_i no será un miembro de e^*. Asúmase ahora que la prueba alcanza e. Por el Invariante 2, cada seguidor en e debe tener $\mathsf{cepoch} = e$ antes de recibir $\texttt{PROBE}(e^*)$. Entonces, cualquier miembro de e que recibe $\texttt{PROBE}(e^*)$ tendrá $\mathsf{initialized} = \texttt{TRUE}$. Por ello, si cualquier miembro en e responde, será con $\texttt{PROBE\_ACK}(\texttt{TRUE}, e^*)$. Dado

que el proceso p_r no se desplazará a la época precedente hasta que al menos un proceso responda con PROBE_ACK, esto significa que la prueba nunca puede ir mas allá de e. Como p_i no es un miembro de e, entonces no puede ser incluido como miembro de e^*. □

Prueba del Invariante 5. Probamos el invariante por inducción en e'. Asúmase que el invariante se cumple para cada $e' < e''$. Probamos ahora que se cumple para $e' = e''$ por inducción en la longitud de la ejecución del protocolo.

Considérese la transición en la línea 51, cuando un futuro líder p_i se incorpora a la época e''. Mostramos que luego de esta transición tenemos $\mathsf{msg}[k] = m$. Puesto que p_i fue elegido como el futuro líder de la época e'', este proceso ha respondido con PROBE_ACK(TRUE, e'') a PROBE(e''). Por lo tanto, p_i era miembro de una configuración en una época $e^* < e''$ que estaba siendo probada. La prueba finaliza cuando al menos un proceso envía un mensaje PROBE_ACK(TRUE, e''). La hipótesis de que COMMIT(e, k, m) es un mensaje enviado impica que en algún punto NEW_STATE(e, msg) es un mensaje enviado con $msg[k] = m$ y este mensaje es notificado por todos los miembros de e excepto su líder, o todos los miembros en e respondieron con ACCEPT_ACK a ACCEPT(e, k, m). Esto significa que todos los miembros en e se encuentran inicializados. Esto, junto con el Invariante 2, implican que la fase de prueba no puede haber ido más allá de e. Por lo tanto, $e \leq e^* < e''$.

Sea e_0 el valor de cepoch en p_i justo antes de la transición en la línea 51. Si $e_0 < e$ entonces p_i no sería un miembro de e, pues todos los miembros de e se encuentran inicializados y tienen $\mathsf{cepoch} = e$, y en consecuencia, por el Invariante 3, no podría haber sido elegido como el futuro líder de la época e''. En consecuencia, $e_0 \geq e$. Por otro lado, como todos los miembros de la época e^* siendo probada tienen $\mathsf{cepoch} \leq e^*$, debemos tener $e_0 \leq e^* < e''$. En consecuencia, $e \leq e_0 < e''$.

Si $e < e_0$, entonces por la hipótesis inductiva, tenemos $\mathsf{msg}[k] = m$ justo luego de la transición en la línea 51, como queríamos. Asúmase ahora que $e_0 = e$. Como COMMIT(e, k, m) es un mensaje enviado, entonces cada proceso en e recibe NEW_STATE(e, msg) con $msg[k] = m$ y responde con NEW_STATE_ACK(e) o cada proceso en e recibe ACCEPT(e, k, m) y responde con ACCEPT_ACK(e, k). El Invariante 2 implica que p_i envió su mensaje NEW_STATE, NEW_STATE_ACK, ACCEPT o ACCEPT_ACK antes de enviar PROBE_ACK(TRUE, e''). Entonces lo requerido sigue del Invariante 4.

Esto implica, en particular, que cualquier mensaje NEW_STATE(e'', msg) enviado con $e'' > e$ debe tener $\mathsf{msg}[k] = m$, por lo que el resultado se cumple luego de la transición en la línea 56. Las líneas 67 y 14 aseguran que el valor no será sobreescrito durante el caso libre de fallos del protocolo. El resultado es trivial para el resto de las transiciones. □

Prueba del Invariante 6. Sean COMMIT(e_1, k, m_1) y COMMIT(e_2, k, m_2) mensajes enviados. Asúmase sin pérdida de generalidad que $e_1 \leq e_2$. Si $e_1 = e_2$, entonces por el Invariante 1, debemos tener $m_1 = m_2$. Asúmase ahora que $e_1 < e_2$. El Invariante 5 implica que el líder de e_2 tenía $msg[k] = m_1$ justo luego de convertirse en líder en la línea 66. Por lo tanto, el mensaje COMMIT(e_2, k, m_2) enviado durante la reconfiguración debe tener $m_1 = m_2$.

Esto implica que, en la línea 67, el líder de e_2 debe haber tenido $\mathsf{next} \geq k$. Por lo tanto, todos los mensajes $\texttt{COMMIT}(e_2, k', \_)$ enviados durante el caso libre de fallos de e_2 deben tener $k' > k$. □

Teorema 6. *El algoritmo satisface Integridad.*

Teorema 7. *El algoritmo satisface No duplicación.*

Probamos a continuación que nuestro protocolo satisface Orden confiable: existe un orden total $\preceq$ en el conjunto de todos los mensajes que han sido entregados en una ejecución por cualquier proceso tal que si un proceso p_i entrega un mensaje m, entonces para todos los mensajes m', $m' \prec m$ implica que p_i entrega m' antes que m. En otras palabras, todos los procesos entregan los mensajes en el mismo orden sin huecos.

Fíjese una ejecución r del algoritmo y sea $\mathcal{M}$ el conjunto de mensajes entregados en r. Defínase un orden total $\preceq_i$ en el conjunto $\mathcal{M}_i$ de mensajes entregados por un proceso p_i en r tal que para cada $m, m' \in \mathcal{M}_i$, $m \preceq_i m'$ si y sólo si p_i entrega m antes que m'. Escribiremos μ_i para denotar la secuencia sobre $\mathcal{M}_i$ inducida por $\preceq_i$ y $\mu_i \lfloor k$ para denotar el prefijo de μ_i de longitud k.

Lema 7. *Para cualquier procesos p_i y p_j, y prefijos μ'_j de μ_j, si $|\mu'_j| \leq |\mu_i|$, entonces μ'_j es un prefijo de μ_i.*

Demostración. Sea $0 \leq k < |\mu'_j| \leq |\mu_i|$. El Invariante 7 implica que existen épocas e, e' tales que $\texttt{COMMIT}(e, k, \mu'_j[k])$ y $\texttt{COMMIT}(e', k, \mu_i[k])$ son mensajes enviados. Por el Invariante 6, debe ser $\mu'_j[k] = \mu_i[k]$. □

Probamos a continuación que el Lema 7 implica la existencia de la relación de orden total estipulada por la propiedad de Orden confiable. Sea $\preceq \triangleq \bigcup_{p_i} \preceq_i$.

Lema 8. *La relación $\preceq$ es antisimétrica.*

Demostración. Asúmase por contradicción que existen mensajes m y m' tales que $m \preceq m'$, $m' \preceq m$, y $m \neq m'$. Entonces, existen procesos p_i y p_j tales que $m \prec_i m'$ y $m' \prec_j m$. Sea ℓ el índice de m' en μ_i. Entonces, por el Lema 7, o bien $|\mu_j| < \ell$ o $\mu_j \lfloor \ell = \mu_i \lfloor \ell$. Si $|\mu_j| < \ell$, entonces por el Lema 7, $m' \prec_j m$ implica que $m' \prec_i m$ contradiciendo la hipótesis $m \prec_i m'$. Si $\mu_j \lfloor \ell = \mu_i \lfloor \ell$, entonces $\mu_j[\ell] = m'$. Dado que $m \prec_i m'$, esto implica que $m \prec_j m'$ contradiciendo la hipótesis $m' \prec_j m$. Por ello, concluimos que $\preceq$ es antisimétrica. □

Lema 9. *La relación $\preceq$ es transitiva.*

Demostración. Asúmase por contradicción que existen mensajes m, m' y m'' en $\mathcal{M}$ tales que $m \prec m'$, $m' \prec m''$ y $m'' \preceq m$. Entonces, existen procesos p_i, p_j y p_k, tales que p_i entrega m y m', p_j entrega m' y m'', y p_k entrega m y m'', y se cumple

$$m \prec_i m' \tag{4.1}$$

$$m' \prec_j m'' \tag{4.2}$$

$$m = m'' \vee m'' \prec_k m \tag{4.3}$$

Por el Lema 7, 4.1 y 4.2 implican que m' tiene el mismo índice en μ_i y μ_j. Por ello, el índice de m en μ_i es menor que el índice de m'' en μ_j, que por el Lema 7 implica que $m \prec_j m''$. En consecuencia, $m \neq m''$, y por 4.3, $m'' \prec_k m$. Sin embargo, puesto que $m \prec_j m''$, el Lema 7 implica que $m \prec_k m''$, lo cual es una contradicción. □

Lema 10. *La relación $\preceq$ es conexa.*

Demostración. Considérese $m, m' \in \mathcal{M}$. Sea p_i un proceso que entrega m y p_j un proceso que entrega m'. Si el índice de m en μ_i es menor que el índice de m' en μ_j, y por el Lema 7, m tiene el mismo índice en μ_i y μ_j, obtenemos que $m \prec_j m'$. Por lo tanto, $m \preceq m'$. De otro modo, por el mismo argumento, obtenemos que $m' \prec_i m$, y en consecuencia $m' \preceq m$. □

Lema 11. *La relación $\preceq$ es un orden total sobre $\mathcal{M}$.*

Teorema 8. *El algoritmo satisface Orden confiable.*

Demostración. Se ha mostrado que los procesos entregan los mensajes de acuerdo a $\preceq$. Necesitamos ahora probar que si un proceso p_i entrega un mensaje m, entonces para todo m' tal que $m' \prec m$, p_i entrega m' y $m' \prec_i m$. Considérense entonces mensajes m y m' tales que p_i entrega m y $m' \prec m$. Entonces, existe un proceso p_j tal que p_j entrega m y m', y $m' \prec_j m$. Por el Lema 7, esto implica que p_i entrega m' y $m' \prec_i m$, como queríamos demostrar. □

Capítulo 5

Single Decree Paxos en Dafny

Para formalizar Single Decree Paxos en Dafny, debemos codificar los procesos, los mensajes y la red subyacente. Para comenzar, necesitamos nombres para los `status` de los procesos en nuestro pseudocódigo. Los definimos en un *enumeration type* que llamamos `St`:

```
datatype St = E | F | L
```

correspondientes a ELECTION, FOLLOWER y LEADER, respectivamente. Los mensajes se modelan como un tipo en el cual tenemos un constructor por cada posible mensaje:

```
datatype Msg = P1A(b: int) |
               P1B(b: int, c: int, v: int, s: int) |
               P2A(b: int, v: int) |
               P2B(b: int, v: int, s: int)
```

Cada constructor es parametrizado por los atributos del mensaje. Existe una diferencia entre los constructores y los mensajes en nuestro pseudocódigo. Aquí, los mensajes `P1B` y `P2B` poseen un argumento extra s que corresponde al emisor del mensaje.

Puesto que Dafny no es capaz de razonar sobre conjuntos, lo asistimos probando que cualesquiera dos quórums se intersecan. Con este fin, tenemos un lema el cual establece que si un conjunto de enteros A se compone de elementos en el rango $[0, N)$, entonces su cardinalidad debe ser menor o igual a N.

```
lemma in_range(A: set<int>, N: int)
    requires ∀ i • i in A ⟹ 0 ≤ i < N
    requires N ≥ 0
    ensures |A| ≤ N
    decreases N
{
    if (N = 0) {
        ∀ i | i in A
            ensures false
        {}
        assert A = {};
    } else if N − 1 in A {
        in_range(A − {N − 1}, N − 1);
    } else {
        in_range(A, N − 1);
    }
}
```

Como se puede ver, la prueba procede recursivamente eliminando elementos hasta alcanzar un conjunto vacío, que debe tener cardinalidad 0. Esto corresponde a una

prueba por inducción en N. Para probar que cualesquiera dos quórums se intersecan procedemos de la siguiente manera: Asúmase que $A, A' \subseteq S$ son quórums disjuntos en algún universo S con cardinalidad N. Como $A \cap A' = \varnothing$, debemos tener $|A \cup A'| = |A| + |A'| > N/2 + N/2 = N$. Por otro lado, como $A \cup A' \subseteq S$, debemos tener también $|A \cup A'| \leq |S| = N$, lo cual es una contradicción. En consecuencia, debemos tener $A \cap A' \neq \varnothing$. Este argumento es formalizado en el siguiente lema, donde el universo es el conjunto de enteros $\texttt{ps} = \{0, 1, ..., N-1\}$.

```
lemma quorums_intersect(ps: set<int>, N: int)
    requires N > 0 ∧ |ps| = N
    requires ∀ p • p in ps ⟹ 0 ≤ p < N
    ensures ∀ A, A' • A ≤ ps ∧ A' ≤ ps ∧ N < 2 * |A| ∧ N < 2 * |A'| ⟹ A * A' ≠ {}
{
    ∀ A, A' | A ≤ ps ∧ A' ≤ ps ∧ 2 * |A| > N ∧ 2 * |A'| > N
        ensures A * A' ≠ {}
    {
        if A * A' = {} {
            in_range(A + A', N);
            assert false;
        }
    }
}
```

Disponemos además de un método auxiliar para implementar el mecanismo descripto en la línea 20, donde tenemos:

$$\begin{aligned}&\textbf{let } j_0 \text{ be such that } p_{j_0} \in \mathcal{Q},\ av_{j_0} \neq \perp \wedge \\ &\qquad \forall p_j \in \mathcal{Q}\ .\ av_j \neq \perp \Rightarrow cbal_j \leq cbal_{j_0}\end{aligned}$$

Aquí -1 representa $\perp$. Las precondiciones establecen que `p1bs` debe ser un conjunto que consiste únicamente de mensajes `P1B`, pues al ser type-safe, Dafny requiere pruebas de que puede aplicar los correspondientes destructores; y, existe al menos un mensaje donde el último valor aceptado no es $\perp$.

```
method pick_with_max_cbal(p1bs: set<Msg>) returns (m: Msg)
    requires ∀ m • m in p1bs ⟹ m.P1B?
    requires ∃ m • m in p1bs ∧ m.v ≠ −1
    ensures m in p1bs ∧ m.P1B? ∧ m.v ≠ −1
    ensures ∀ m' • m' in p1bs ∧ m'.P1B? ∧ m'.v ≠ −1 ⟹ m'.c ≤ m.c
{
    m :| m in p1bs ∧ m.P1B? ∧ m.v ≠ −1;
    var q', q := {m}, p1bs − {m};
    while (q ≠ {})
        decreases q
        invariant q + q' = p1bs
        invariant m in p1bs ∧ m.P1B? ∧ m.v ≠ −1
        invariant ∀ m' • m' in q' ∧ m'.P1B? ∧ m'.v ≠ −1 ⟹ m'.c ≤ m.c
    {
        var y :| y in q;
        q, q' := q − {y}, q' + {y};
        if (y.P1B? ∧ y.v ≠ −1 ∧ m.c < y.c) {
            m := y;
        }
    }
}
```

Las postcondiciones establecen que un mensaje es escogido del conjunto de manera tal que este tiene el mayor *cbal* de entre los mensajes con $av \neq \perp$, representados aquí por los atributos c y v, respectivamente.

En la descripción del algoritmo, mencionamos que los procesos escogen ballots que lideran, pero no prescribimos una implementación particular para ello. En el modelo en Dafny, precisamos este hecho teniendo N procesos con identificadores en el conjunto $\{0, ..., N-1\}$, donde un proceso p lidera todos los ballots b tales que b mód $N = p$.

Por lo tanto, contamos con un método auxiliar para implementar el mecanismo descripto en la línea 8, donde tenemos:

$$\textbf{let } b \text{ be such that } b > \mathsf{bal} \wedge \texttt{leader}(b) = p_i$$

Este método escoge el siguiente número de ballot perteneciente al proceso p mayor a su ballot actual b.

```
method new_bal(b: int, p: int, N: int) returns (b': int)
    requires b ≥ −1 ∧ N > 0
    ensures b' > b ∧ b' %N = p
    decreases *
{
    b' := b + 1;
    while (b' %N ≠ p)
        decreases *
        invariant b' > b
    {
        b' := b' + 1;
    }
}
```

Introducimos a continuación dos predicados auxiliares que establecen cuándo un valor v es *choosable* o elegido en un ballot b, respectivamente, por un quórum A. Las variables `bal`, `ios` y `ps` serán introducidas más adelante en este capítulo.

```
predicate choosable(A: set<int>, b: int, v: int,
        bal: map<int, int>, ios: set<Msg>, ps: set<int>)
    requires bal.Keys = ps
{
    2 * |A| > |ps| ∧ A ≤ ps ∧
    (∀ p • p in A ⟹ (P2B(b, v, p) in ios ∨ bal[p] ≤ b))
}

predicate chosen(A: set<int>, b: int, v: int, ios: set<Msg>, ps: set<int>)
{
    2 * |A| > |ps| ∧ A ≤ ps ∧
    (∀ p • p in A ⟹ P2B(b, v, p) in ios)
}
```

Representar el protocolo requiere estado para cada proceso. En particular, cada proceso tiene un status, su último valor aceptado av, su ballot bal, el ballot en el cual ha aceptado su último valor cbal, y un buffer para persistir todos los mensajes `P1B` de su elección de líder actual, si así lo hubiera. Para este propósito utilizamos mapas desde los procesos a los respectivos valores, es decir, existe un mapa cuyo dominio es el conjunto de procesos y cuyo codominio es el tipo del valor correspondiente para cada uno de los atributos descriptos arriba.

Con el fin de disponer de la información necesaria en el scope, modelamos el protocolo como un bucle infinito encapsulado en un sólo método. Como los procesos son independientes, la reducción garantiza que es seguro asumir que los eventos se manejan de manera mutuamente excluyente entre procesos, de manera similar a las

pruebas manuales en este trabajo. Por lo tanto, en cada paso del bucle, escogemos de manera no determinística un proceso y un mensaje, y el proceso opera sobre ese mensaje si las condiciones requeridas se cumplen.

Modelamos la red simplemente como un conjunto. Para escoger un mensaje, lo hacemos del conjunto de mensajes enviados `ios`. Este simple modelo satisface propiedades deseables: escoger un mensaje de manera no determinística de un conjunto implica que un mensaje puede nunca ser elegido (el mensaje ha sido perdido en la red), o el mensaje se escoge más de una vez (el mensaje es duplicado), o que se escoge en un orden arbitrario (los mensajes son reordenados).

En algunas ocasiones, resulta necesario conocer el estado de los ballots y la red justo antes de ingresar al paso actual del bucle, que nos permitirá hacer uso de la hipótesis inductiva. Por ello, utilizamos dos variables adicionales para persistir este estado: `ios'` y `bal'`.

El método principal es entonces:

```
method sd_paxos(ps: set<int>, N: int)
    requires N > 0 ∧ |ps| = N
    requires ∀ p • p in ps ⟹ 0 ≤ p < N
    decreases *
{
    var st:   map<int, St>       := map p | p in ps •  F;
    var av:   map<int, int>      := map p | p in ps • −1;
    var bal:  map<int, int>      := map p | p in ps • −1;
    var cbal: map<int, int>      := map p | p in ps • −1;
    var p1bs: map<int, set<Msg⟩  := map p | p in ps • {};
    var ios: set<Msg> := {};
    ghost var ios' := ios;
    ghost var bal' := bal;
    quorums_intersect(ps, N);

    while true
        decreases *
        // INDUCTIVE INVARIANT
    {
        bal' := bal; ios' := ios;

        var p       :| p in ps;
        var propose :| propose in {false, true};

        // CODE IMPLEMENTING EACH TRANSITION
    }
}
```

Las precondiciones establecen que el universo está compuesto de un conjunto de $N > 0$ procesos con identificadores en $\{0, 1, ..., N-1\}$. La anotación `decreases *` indica a Dafny que no requerimos prueba de terminación, pues el bucle se ejecutará indefinidamente. Luego de declarar el estado requerido, introducimos en el scope la información de que dos cualesquiera quórums se intersecan invocando el lema correspondiente.

Por simplicidad omititmos la existencia de los mensajes `propose` y en su lugar asumimos de manera equivalente que un proceso decide de manera no determinística cuándo proponer un valor. El operador `:|` significa "escoger un valor tal que".

Por lo tanto las líneas

```
        var p       :| p in ps;
        var propose :| propose in {false, true};
```

logran el funcionamiento deseado; esto es, escoger cualquier proceso del conjunto y decidir si este desea proponer un valor o no en el paso actual del protocolo. A continuación mostramos la codificación de las transiciones:

```
if propose ∨ ios = {} {
    var b := new_bal(bal[p], p, N);
    ios := ios + { P1A(b) };
} else {
    var msg :| msg in ios;

    if (msg.P1A? ∧ bal[p] < msg.b) {
        st := st[p := E];
        p1bs := p1bs[p := {}];
        bal := bal[p := msg.b];
        ios := ios + { P1B(bal[p], cbal[p], av[p], p) };
    }

    if (msg.P1B? ∧ bal[p] = msg.b ∧ st[p] = E ∧ bal[p] %N = p) {
        p1bs   := p1bs[p := p1bs[p] + {  msg  }];
        var A' := set m | m in p1bs[p] • m.s;

        if (2 * |A'| > N) {
            st := st[p := L];

            if (∀ m • m in p1bs[p] ⟹ m.v = −1) {
                var v :| v ≠ −1;
                ios   := ios + { P2A(bal[p], v) };
            } else {
                var m := pick_with_max_cbal(p1bs[p]);
                ios   := ios + { P2A(bal[p], m.v) };
            }
        }
    }

    if (msg.P2A? ∧ bal[p] ≤ msg.b) {
        av   := av  [p := msg.v];
        bal  := bal [p := msg.b];
        cbal := cbal[p := msg.b];
        if (msg.b %N ≠ p) {
            st := st[p := F];
        }
        ios  := ios + { P2B(bal[p], av[p], p) };
    }
}
```

No proveemos mayores detalles del código pues es casi una traducción directa del pseudocódigo en las Figuras 2.2 y 2.3. Nuestro objetivo ahora es probar

$$\mathcal{P} \triangleq \forall b, b', v, v'.\ \texttt{chosen}(b, v) \land \texttt{chosen}(b', v') \Rightarrow v = v'$$

Nuestra estrategia será entonces encontrar un invariante que implique $\mathcal{P}$ y delegaremos en Dafny probar que efectivamente es preservado por el protocolo, en cuyo caso, habremos probado que el protocolo satisface $\mathcal{P}$. Con este propósito fortalecemos los invariantes en la Figura 2.4 de la siguiente manera:

1. Si en un proceso p_i tenemos $\mathsf{status} = \textsc{election}$, entonces todos los mensajes en el buffer `p1bs` tienen ballot igual a bal_{p_i}.
2. En cualquier proceso siempre se tiene $\mathsf{cbal} \leq \mathsf{bal}$.
3. Si un proceso p_i ha enviado un mensaje $\texttt{P1B}(b, \_, \_)$, entonces $\mathsf{bal}_{p_i} \geq b$.
4. Si un proceso p_i ha enviado un mensaje $\texttt{P2B}(b, \_)$, entonces $\mathsf{cbal}_{p_i} \geq b$.
5. Si un proceso p_i ha enviado mensajes $\texttt{P2B}(b, \_)$ y $\texttt{P1B}(b', c', \_)$ con $b' > b$, entonces $c' \geq b$.

6. Si un proceso p_i ha enviado un mensaje $\texttt{P2A}(b, \_)$, entonces $\mathsf{bal}_{p_i} \geq b$.
7. Si un proceso p_i ha enviado un mensaje $\texttt{P2A}(b, \_)$ y tiene $\mathsf{status} = \text{ELECTION}$, entonces $\mathsf{bal}_{p_i} > b$.
8. Para cualquier mensajes enviados $\texttt{P2A}(b, v)$ y $\texttt{P2A}(b, v')$, tenemos $v = v'$.
9. Si $\texttt{P2A}(\_, v)$ es un mensaje enviado, entonces $v \neq \bot$.
10. Si $\texttt{P2B}(b, v)$ es un mensaje enviado, entonces también lo es $\texttt{P2A}(b, v)$.
11. Si en un proceso p_i tenemos $\mathsf{cbal} \neq 0$, entonces $\texttt{P2A}(\mathsf{cbal}_{p_i}, \mathsf{av}_{p_i})$ es un mensaje enviado.
12. Si $\texttt{P1B}(\_, c, v)$ es un mensaje enviado con $c \neq 0$, entonces también lo es $\texttt{P2A}(c, v)$.
13. Si un proceso p_i ha enviado un mensaje $\texttt{P2A}(b, \_)$ y tiene $\mathsf{bal} = b$, entonces $\mathsf{status}_{p_i} = \text{LEADER}$.
14. Si $\texttt{chosen}(b, v)$, entonces $\texttt{choosable}(b, v)$.
15. Si $\texttt{choosable}(b, v)$ luego de un paso del protocolo, entonces $\texttt{choosable}(b, v)$ también antes del mismo.
16. Si $\texttt{choosable}(b, v)$ y $\texttt{P2A}(b', v')$ es un mensaje enviado con $b' > b$, entonces $v' = v$.
17. Si $\texttt{chosen}(b, v)$ y $\texttt{chosen}(b', v')$, entonces $v' = v$.

Estos invariantes son codificados en Dafny del siguiente modo:

```
invariant st.Keys = av.Keys = bal.Keys = cbal.Keys = p1bs.Keys = bal'.Keys = ps
invariant ∀ p, m • p in ps ∧ m in p1bs[p] ⟹ m.P1B?
invariant ∀ p, m • p in ps ∧ m in p1bs[p] ⟹ m = P1B(m.b, m.c, m.v, m.s)
invariant ∀ p, m • p in ps ∧ m in p1bs[p] ⟹ m in ios
invariant ∀ p, m • p in ps ∧ st[p] = E ∧ m in p1bs[p] ⟹ m.b = bal[p]
invariant ∀ p, b, c, v • P1B(b, c, v, p) in ios ⟹ p in ps
invariant ∀ p, b, v • P2B(b, v, p) in ios ⟹ p in ps
invariant ∀ p • p in ps ⟹ bal[p] ≥ cbal[p] ≥ −1
invariant ∀ p, b, c, v • P1B(b, c, v, p) in ios ⟹ b ≤ bal[p]
invariant ∀ p, b, v • P2B(b, v, p) in ios ⟹ b ≤ cbal[p]
invariant ∀ p, b, v, b', c', v' • P2B(b, v, p) in ios ∧ P1B(b', c', v', p) in ios
    ∧ b' > b ⟹ c' ≥ b
invariant ∀ b, v • P2A(b, v) in ios ⟹ b %N in ps
invariant ∀ b, v • P2A(b, v) in ios ⟹ b ≤ bal[b %N]
invariant ∀ b, v • P2A(b, v) in ios ∧ st[b %N] = E ⟹ b < bal[b %N]
invariant ∀ b, v, v' • P2A(b, v) in ios ∧ P2A(b, v') in ios ⟹ v' = v
invariant ∀ b, v • P2A(b, v) in ios ⟹ v ≠ −1
invariant ∀ p, b, v • P2B(b, v, p) in ios ⟹ P2A(b, v) in ios
invariant ∀ p • p in ps ⟹ cbal[p] ≠ −1 ⟹ P2A(cbal[p], av[p]) in ios
invariant ∀ p, b, c, v • P1B(b, c, v, p) in ios ∧ c ≠ −1 ⟹ P2A(c, v) in ios
invariant ∀ p, b, v • p in ps ∧ P2A(b, v) in ios ∧ bal[p] = b ∧
    b %N = p ⟹ st[p] = L
invariant ∀ A, b, v • chosen(A, b, v, ios, ps) ⟹ choosable(A, b, v, bal, ios, ps)
invariant ∀ A, b, v • choosable(A, b, v, bal, ios, ps) ⟹ choosable(A, b, v, bal', ios', ps)
invariant ∀ A, b, v, b', v' • choosable(A, b, v, bal, ios, ps) ∧ P2A(b', v') in ios
    ∧ b' > b > −1 ⟹ v' = v
invariant ∀ A, A', b, v, b', v' • chosen(A, b, v, ios, ps) ∧ chosen(A', b', v', ios, ps) ∧
    b' > b > −1 ⟹ v' = v
```

Estos incluyen además ciertos invariantes específicos a Dafny para asegurar que puede aplicar los respectivos destructores en la implementación de ciertas transiciones. Este invariante implica trivialmente nuestra propiedad pues la contiene, luego si Dafny puede probar que el invariante es inductivo hemos cumplido nuestro objetivo. En efecto, al ejecutar el verificador en el código descripto obtenemos el siguiente resultado:

```
alex@tuko:~/Documentos/Dafny$ time dafny -compile:0 -trace sd-paxos.dfy
[TRACE] Using prover: /home/alex/Documentos/Dafny/dafny-source/z3/bin/z3
Dafny 2.2.0.10923
Parsing sd-paxos.dfy
Coalescing blocks...
Inlining...

Running abstract interpretation...
  [0,045867 s]
[TRACE] Using prover: /home/alex/Documentos/Dafny/dafny-source/z3/bin/z3

Verifying Impl$$_module.__default.in__range ...
  [0,400 s, 13 proof obligations]  verified

Verifying Impl$$_module.__default.quorums__intersect ...
  [0,078 s, 6 proof obligations]  verified

Verifying CheckWellformed$$_module.__default.pick__with__max__cbal ...
  [0,074 s, 5 proof obligations]  verified

Verifying Impl$$_module.__default.pick__with__max__cbal ...
  [0,113 s, 25 proof obligations]  verified

Verifying CheckWellformed$$_module.__default.new__bal ...
  [0,074 s, 1 proof obligation]  verified

Verifying Impl$$_module.__default.new__bal ...
  [0,077 s, 5 proof obligations]  verified

Verifying CheckWellformed$$_module.__default.choosable ...
  [0,080 s, 1 proof obligation]  verified

Verifying Impl$$_module.__default.sd__paxos ...
  [33,114 s, 126 proof obligations]  verified

Dafny program verifier finished with 8 verified, 0 errors

real    0m35,329s
user    0m35,178s
sys     0m0,325s
```

Aquí se pueden observar las obligaciones de prueba generadas por cada procedimiento, y cómo Dafny ha podido probar exitosamente cada una de ellas.

Bibliografía

[1] Manuel Bravo y Alexey Gotsman. "Reconfigurable Atomic Transaction Commit". En: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. PODC '19. Toronto ON, Canada: ACM, 2019, págs. 399-408. ISBN: 978-1-4503-6217-7. DOI: 10.1145/3293611.3331590. URL: http://doi.acm.org/10.1145/3293611.3331590.

[2] Christian Cachin, Rachid Guerraoui y Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. 2nd. Springer Publishing Company, Incorporated, 2011. ISBN: 3642152597, 9783642152597.

[3] Gregory V. Chockler y Alexey Gotsman. "Multi-Shot Distributed Transaction Commit (Extended Version)". En: *CoRR* abs/1808.00688 (2018). arXiv: 1808.00688. URL: http://arxiv.org/abs/1808.00688.

[4] Chris Hawblitzel y col. "IronFleet: Proving Practical Distributed Systems Correct". En: *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM - Association for Computing Machinery, 2015. URL: https://www.microsoft.com/en-us/research/publication/ironfleet-proving-practical-distributed-systems-correct/.

[5] Chris Hawblitzel y col. "IronFleet: Proving Safety and Liveness of Practical Distributed Systems". En: *Communications of the ACM* 60 (2017). URL: https://www.microsoft.com/en-us/research/publication/ironfleet-proving-safety-liveness-practical-distributed-systems/.

[6] Luke Herbert, Rustan Leino y Jose Quaresma. "Using Dafny, an Automatic Program Verifier". En: *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Ed. por Bertrand Meyer y Martin Nordio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, págs. 156-181. ISBN: 978-3-642-35746-6. DOI: 10.1007/978-3-642-35746-6_6. URL: https://doi.org/10.1007/978-3-642-35746-6_6.

[7] Flavio P. Junqueira, Benjamin C. Reed y Marco Serafini. *Dissecting Zab*. Inf. téc. YL-2010-0007. http://labs.yahoo.com/files/YL-2010-007.pdf. Yahoo! Research, 2010.

[8] Flavio P. Junqueira, Benjamin C. Reed y Marco Serafini. "Zab: High-performance broadcast for primary-backup systems". En: *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*. 2011, págs. 245-256. DOI: 10.1109/DSN.2011.5958223.

[9] Jason Koenig y Rustan Leino. "Getting Started with Dafny: A Guide". En: 2012. URL: https://www.microsoft.com/en-us/research/publication/getting-started-dafny-guide/.

[10] Leslie Lamport. "Paxos Made Simple". En: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), págs. 51-58. URL: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/.

[11] Rustan Leino. "Accessible Software Verification with Dafny". En: *IEEE Software* 34.6 (2017), págs. 94-97. DOI: 10.1109/MS.2017.4121212.

[12] Rustan Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". En: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. por Edmund M. Clarke y Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, págs. 348-370. ISBN: 978-3-642-17511-4.

[13] Rustan Leino. "Developing Verified Programs with Dafny". En: *Verified Software: Theories, Tools, Experiments*. Ed. por Rajeev Joshi, Peter Müller y Andreas Podelski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, págs. 82-82. ISBN: 978-3-642-27705-4.

[14] Rustan Leino. "Modeling Concurrency in Dafny". En: *Engineering Trustworthy Software Systems*. Ed. por Jonathan P. Bowen, Zhiming Liu y Zili Zhang. Cham: Springer International Publishing, 2018, págs. 115-142. ISBN: 978-3-030-02928-9.

[15] Rustan Leino y Valentin Wüstholz. "The Dafny Integrated Development Environment". En: *Electronic Proceedings in Theoretical Computer Science* 149 (abr. de 2014). DOI: 10.4204/EPTCS.149.2.

[16] Paqui Lucio. "A Tutorial on Using Dafny to Construct Verified Software". En: *Electronic Proceedings in Theoretical Computer Science* 237 (2017), 1–19. ISSN: 2075-2180. DOI: 10.4204/eptcs.237.1. URL: http://dx.doi.org/10.4204/EPTCS.237.1.

[17] André Medeiros. "ZooKeeper's atomic broadcast protocol: Theory and practice". En: 2012.

[18] Robbert Van Renesse y Deniz Altinbuken. "Paxos Made Moderately Complex". En: *ACM Comput. Surv.* 47.3 (feb. de 2015), 42:1-42:36. ISSN: 0360-0300. DOI: 10.1145/2673577. URL: http://doi.acm.org/10.1145/2673577.