En busca de un algoritmo de consenso comprensible (Versión extendida)

Diego Ongaro y John Ousterhout
Universidad Stanford

Resumen

Raft es un algoritmo de consenso para administrar un registro replicado. Produce un resultado equivalente a (multi-)Paxos, y es tan eficiente como Paxos, pero su estructura es diferente de Paxos; esto hace que Raft sea más comprensible que Paxos y también proporciona una mejor base para construir sistemas prácticos. Para mejorar la comprensión, Raft separa los elementos clave del consenso, como la elección del líder, la replicación de registros y la seguridad, y aplica un mayor grado de coherencia para reducir la cantidad de estados que se deben considerar. Los resultados de un estudio de usuarios demuestran que Raft es más fácil de aprender para los estudiantes que Paxos. Raft también incluye un nuevo mecanismo para cambiar la membresía del clúster, que utiliza mayorías superpuestas para garantizar la seguridad.

1. Introducción

Los algoritmos de consenso permiten que una colección de máquinas funcione como un grupo coherente que puede sobrevivir a las fallas de algunos de sus miembros. Debido a esto, juegan un papel clave en la construcción de sistemas de software confiables a gran escala. Paxos [15, 16] ha dominado la discusión de los algoritmos de consenso durante la última década: la mayoría de las implementaciones de consenso se basan en Paxos o están influenciadas por él, y Paxos se ha convertido en el vehículo principal utilizado para enseñar a los estudiantes sobre el consenso.

Desafortunadamente, Paxos es bastante difícil de entender, a pesar de los numerosos intentos de hacerlo más accesible. Además, su arquitectura requiere cambios complejos para admitir sistemas prácticos. Como resultado, tanto los desarrolladores de sistemas como los estudiantes luchan con Paxos.

Después de luchar con Paxos nosotros mismos, nos dispusimos a encontrar un nuevo algoritmo de consenso que pudiera proporcionar una mejor base para la educación y la creación de sistemas. Nuestro enfoque era inusual en el sentido de que nuestro objetivo principal era *comprensibilidad*: ¿podríamos definir un algoritmo de consenso para sistemas prácticos y describirlo de una manera que sea significativamente más fácil de aprender que Paxos? Además, queríamos que el algoritmo facilitara el desarrollo de intuiciones que son esenciales para los constructores de sistemas. Era importante no solo que el algoritmo funcionara, sino que fuera obvio por qué funciona.

El resultado de este trabajo es un algoritmo de consenso llamado Raft. Al diseñar Raft, aplicamos técnicas específicas para mejorar la comprensión, incluida la descomposición (Raft separa la elección del líder, la replicación de registros y la seguridad) y reducción del espacio de estado (en relación con Paxos, Raft reduce el grado de no determinismo y las formas en que los servidores pueden ser inconsistentes entre sí). Un estudio de usuarios con 43 estudiantes en dos universidades muestra que Raft es significativamente más fácil de entender que Paxos: después de aprender ambos algoritmos, 33 de estos estudiantes pudieron responder preguntas sobre Raft mejor que preguntas sobre Paxos.

Raft es similar en muchos aspectos a los algoritmos de consenso existentes (en particular, Viewstamped Replication de Oki y Liskov [29, 22]), pero tiene varias características novedosas:

- Líder fuerte:Raft utiliza una forma de liderazgo más fuerte que otros algoritmos de consenso. Por ejemplo, las entradas de registro solo fluyen desde el líder a otros servidores. Esto simplifica la gestión del registro replicado y facilita la comprensión de Raft.
- Elección de líder:Raft usa cronómetros aleatorios para elegir líderes. Esto agrega solo una pequeña cantidad de mecanismo a los latidos del corazón que ya se requieren para cualquier algoritmo de consenso, mientras resuelve los conflictos de manera simple y rápida.
- Cambios de membresía:El mecanismo de Raft para cambiar el conjunto de servidores en el clúster utiliza un nuevo *consenso conjunto*enfoque donde la mayoría de dos configuraciones diferentes se superponen durante las transiciones. Esto permite que el clúster siga funcionando con normalidad durante los cambios de configuración.

Creemos que Raft es superior a Paxos y otros algoritmos de consenso, tanto con fines educativos como base para la implementación. Es más simple y comprensible que otros algoritmos; se describe lo suficientemente completo como para satisfacer las necesidades de un sistema práctico; tiene varias implementaciones de código abierto y es utilizado por varias empresas; sus propiedades de seguridad han sido formalmente especificadas y probadas; y su eficiencia es comparable a otros algoritmos.

El resto del documento presenta el problema de la máquina de estado replicada (Sección 2), analiza las fortalezas y debilidades de Paxos (Sección 3), describe nuestro enfoque general de comprensibilidad (Sección 4), presenta el algoritmo de consenso de Raft (Secciones 5–8) , evalúa Raft (Sección 9) y analiza el trabajo relacionado (Sección 10).

2 máquinas de estado replicadas

Los algoritmos de consenso suelen surgir en el contexto de *máquinas de estado replicadas*[37]. En este enfoque, las máquinas de estado en una colección de servidores calculan copias idénticas del mismo estado y pueden continuar operando incluso si algunos de los servidores están inactivos. Las máquinas de estado replicadas son

Este informe técnico es una versión ampliada de [32]; el material adicional se indica con una barra gris en el margen. Publicado el 20 de mayo de 2014.

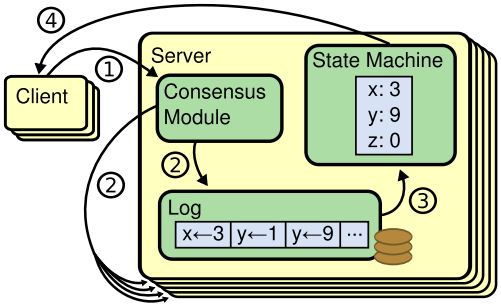

Figura 1:Arquitectura de máquina de estado replicada. El algoritmo de consenso administra un registro replicado que contiene comandos de máquina de estado de los clientes. Las máquinas de estado procesan secuencias idénticas de comandos de los registros, por lo que producen los mismos resultados.

Se utiliza para resolver una variedad de problemas de tolerancia a fallas en sistemas distribuidos. Por ejemplo, los sistemas a gran escala que tienen un solo líder de clúster, como GFS [8], HDFS [38] y RAMCloud [33], generalmente usan una máquina de estado replicada separada para administrar la elección del líder y almacenar la información de configuración que debe sobrevivir. el líder se estrella. Los ejemplos de máquinas de estado replicadas incluyen Chubby [2] y ZooKeeper [11].

Las máquinas de estado replicadas generalmente se implementan mediante un registro replicado, como se muestra en la Figura 1. Cada servidor almacena un registro que contiene una serie de comandos, que su máquina de estado ejecuta en orden. Cada registro contiene los mismos comandos en el mismo orden, por lo que cada máquina de estado procesa la misma secuencia de comandos. Dado que las máquinas de estado son deterministas, cada una calcula el mismo estado y la misma secuencia de salidas.

Mantener la coherencia del registro replicado es el trabajo del algoritmo de consenso. El módulo de consenso en un servidor recibe comandos de los clientes y los agrega a su registro. Se comunica con los módulos de consenso en otros servidores para garantizar que cada registro finalmente contenga las mismas solicitudes en el mismo orden, incluso si algunos servidores fallan. Una vez que los comandos se replican correctamente, la máquina de estado de cada servidor los procesa en orden de registro y los resultados se devuelven a los clientes. Como resultado, los servidores parecen formar una sola máquina de estado altamente confiable.

Los algoritmos de consenso para sistemas prácticos suelen tener las siguientes propiedades:

- aseguran*la seguridad*(nunca devuelve un resultado incorrecto) en todas las condiciones no bizantinas, incluidas las demoras de la red, las particiones y la pérdida, duplicación y reordenación de paquetes.
- Son completamente funcionales (*disponible*) siempre que la mayoría de los servidores estén operativos y puedan comunicarse entre sí y con los clientes. Por lo tanto, un clúster típico de cinco servidores puede tolerar la falla de dos servidores cualesquiera. Se supone que los servidores fallan al detenerse; más tarde pueden recuperarse del estado en almacenamiento estable y volver a unirse al clúster.
- No dependen del tiempo para asegurar la consistencia. Tendencia de los registros: los relojes defectuosos y los retrasos extremos en los mensajes pueden, en el peor de los casos, causar problemas de disponibilidad.
- En el caso común, un comando puede completarse tan pronto como la mayoría del clúster haya respondido a una sola ronda de llamadas a procedimientos remotos; una minoría de servidores lentos no tiene por qué afectar el rendimiento general del sistema.

3 ¿Qué tiene de malo Paxos?

En los últimos diez años, el protocolo Paxos de Leslie Lamport [15] se ha convertido casi en sinónimo de consenso: es el protocolo que se enseña con mayor frecuencia en los cursos y la mayoría de las implementaciones de consenso lo utilizan como punto de partida. Paxos primero define un protocolo capaz de llegar a un acuerdo sobre una única decisión, como una única entrada de registro replicada. Nos referimos a este subconjunto como*Paxos de un solo decreto*. Luego, Paxos combina varias instancias de este protocolo para facilitar una serie de decisiones, como un registro (*multipaxos*). Paxos garantiza tanto la seguridad como la vida, y admite cambios en la membresía del clúster. Su corrección ha sido probada, y es eficiente en el caso normal.

Desafortunadamente, Paxos tiene dos inconvenientes importantes. El primer inconveniente es que Paxos es excepcionalmente difícil de entender. La explicación completa [15] es notoriamente opaca; pocas personas logran comprenderlo, y solo con un gran esfuerzo. Como resultado, ha habido varios intentos de explicar Paxos en términos más simples [16, 20, 21]. Estas explicaciones se centran en el subconjunto de un solo decreto, pero siguen siendo un desafío. En una encuesta informal de asistentes a NSDI 2012, encontramos pocas personas que se sintieran cómodas con Paxos, incluso entre investigadores experimentados. Luchamos con Paxos nosotros mismos; no pudimos entender el protocolo completo hasta después de leer varias explicaciones simplificadas y diseñar nuestro propio protocolo alternativo, un proceso que tomó casi un año.

Nuestra hipótesis es que la opacidad de Paxos se deriva de su elección del subconjunto de decreto único como base. Paxos de un solo decreto es denso y sutil: se divide en dos etapas que no tienen explicaciones intuitivas simples y no se pueden entender de forma independiente. Debido a esto, es difícil desarrollar intuiciones acerca de por qué funciona el protocolo de decreto único. Las reglas de composición para multi-Paxos agregan una complejidad y sutileza adicionales significativas. Creemos que el problema general de llegar a un consenso sobre múltiples decisiones (es decir, un registro en lugar de una sola entrada) se puede descomponer de otras formas que son más directas y obvias.

El segundo problema con Paxos es que no proporciona una buena base para construir implementaciones prácticas. Una razón es que no existe un algoritmo ampliamente aceptado para múltiples Paxos. Las descripciones de Lamport se refieren principalmente a Paxos de un solo decreto; esbozó posibles enfoques para múltiples Paxos, pero faltan muchos detalles. Ha habido varios intentos de desarrollar y optimizar Paxos, como [26], [39] y [13], pero estos difieren

unos de otros y de los bocetos de Lamport. Sistemas como Chubby [4] han implementado algoritmos similares a Paxos, pero en la mayoría de los casos sus detalles no han sido publicados.

Además, la arquitectura de Paxos es pobre para construir sistemas prácticos; esta es otra consecuencia de la descomposición en decreto único. Por ejemplo, hay poco beneficio en elegir una colección de entradas de registro de forma independiente y luego fusionarlas en un registro secuencial; esto solo agrega complejidad. Es más simple y eficiente diseñar un sistema en torno a un registro, donde las nuevas entradas se agregan secuencialmente en un orden restringido. Otro problema es que Paxos utiliza un enfoque simétrico de igual a igual en su núcleo (aunque finalmente sugiere una forma débil de liderazgo como una optimización del rendimiento). Esto tiene sentido en un mundo simplificado donde solo se tomará una decisión, pero pocos sistemas prácticos utilizan este enfoque. Si se deben tomar una serie de decisiones, es más simple y rápido elegir primero un líder,

Como resultado, los sistemas prácticos se parecen poco a Paxos. Cada implementación comienza con Paxos, descubre las dificultades para implementarlo y luego desarrolla una arquitectura significativamente diferente. Esto lleva mucho tiempo y es propenso a errores, y las dificultades para comprender Paxos exacerban el problema. La formulación de Paxos puede ser buena para probar teoremas sobre su corrección, pero las implementaciones reales son tan diferentes de Paxos que las pruebas tienen poco valor. El siguiente comentario de los implementadores de Chubby es típico:

> Existen brechas significativas entre la descripción del algoritmo Paxos y las necesidades de un sistema del mundo real. . . . el sistema final se basará en un protocolo no probado [4].

Debido a estos problemas, llegamos a la conclusión de que Paxos no proporciona una buena base ni para la creación de sistemas ni para la educación. Dada la importancia del consenso en los sistemas de software a gran escala, decidimos ver si podíamos diseñar un algoritmo de consenso alternativo con mejores propiedades que Paxos. Raft es el resultado de ese experimento.

4 Diseño para la comprensibilidad

Teníamos varios objetivos al diseñar Raft: debe proporcionar una base completa y práctica para la construcción del sistema, de modo que reduzca significativamente la cantidad de trabajo de diseño requerido por los desarrolladores; debe ser seguro en todas las condiciones y disponible en condiciones de funcionamiento típicas; y debe ser eficiente para operaciones comunes. Pero nuestro objetivo más importante, y el desafío más difícil, era *comprensibilidad*. Debe ser posible que una gran audiencia entienda el algoritmo cómodamente. Además, debe ser posible desarrollar intuiciones sobre el algoritmo, de modo que los creadores de sistemas puedan realizar las extensiones que son inevitables en las implementaciones del mundo real.

Hubo numerosos puntos en el diseño de Raft en los que tuvimos que elegir entre enfoques alternativos. En estas situaciones, evaluamos las alternativas basándonos en la comprensibilidad: ¿qué tan difícil es explicar cada alternativa (por ejemplo, qué tan complejo es su espacio de estado y tiene implicaciones sutiles?) y qué tan fácil será para un lector comprender completamente entender el enfoque y sus implicaciones?

Reconocemos que existe un alto grado de subjetividad en dicho análisis; sin embargo, utilizamos dos técnicas que son de aplicación general. La primera técnica es el conocido enfoque de descomposición de problemas: siempre que sea posible, dividimos los problemas en partes separadas que podrían resolverse, explicarse y comprenderse de forma relativamente independiente. Por ejemplo, en Raft separamos la elección de líder, la replicación de registros, la seguridad y los cambios de membresía.

Nuestro segundo enfoque fue simplificar el espacio de estados reduciendo el número de estados a considerar, haciendo que el sistema sea más coherente y eliminando el no determinismo donde sea posible. Específicamente, no se permite que los registros tengan agujeros, y Raft limita las formas en que los registros pueden volverse inconsistentes entre sí. Aunque en la mayoría de los casos intentamos eliminar el no determinismo, hay algunas situaciones en las que el no determinismo mejora la comprensión. En particular, los enfoques aleatorios introducen el no determinismo, pero tienden a reducir el espacio de estado al manejar todas las opciones posibles de manera similar ("elija cualquiera; no importa"). Utilizamos la aleatorización para simplificar el algoritmo de elección del líder de Raft.

5 El algoritmo de consenso Raft

Raft es un algoritmo para administrar un registro replicado de la forma descrita en la Sección 2. La Figura 2 resume el algoritmo en forma resumida como referencia, y la Figura 3 enumera las propiedades clave del algoritmo; los elementos de estas figuras se discuten por partes en el resto de esta sección.

Raft implementa el consenso eligiendo primero a un distinguido *líder*, luego le da al líder la responsabilidad completa de administrar el registro replicado. El líder acepta las entradas de registro de los clientes, las replica en otros servidores y les dice a los servidores cuándo es seguro aplicar las entradas de registro a sus máquinas de estado. Tener un líder simplifica la gestión del registro replicado. Por ejemplo, el líder puede decidir dónde colocar nuevas entradas en el registro sin consultar a otros servidores, y los datos fluyen de forma sencilla desde el líder a otros servidores. Un líder puede fallar o desconectarse de los otros servidores, en cuyo caso se elige un nuevo líder.

Dado el enfoque del líder, Raft descompone el problema de consenso en tres subproblemas relativamente independientes, que se analizan en las subsecciones siguientes:

- Elección de líder:se debe elegir un nuevo líder cuando un líder existente falla (Sección 5.2).
- Registro de replicación:el líder debe aceptar las entradas de registro

Estado

Estado persistente en todos los servidores:
(Actualizado en almacenamiento estable antes de responder a RPC)

término actual	último término que ha visto el servidor (inicializado en 0 en el primer arranque, aumenta monótonamente)
votado por	ID de candidato que recibió el voto en la actualidad término (o nulo si ninguno)
Iniciar sesión[]	entradas de registro; cada entrada contiene comando para la máquina de estado y el término cuando el líder recibió la entrada (el primer índice es 1)

Estado volátil en todos los servidores: **índice**

de confirmación	índice de la entrada de registro más alta que se sabe que está comprometida (inicializado en 0, aumenta monótonamente)
últimoAplicado	índice de la entrada de registro más alta aplicada al estado máquina (inicializado a 0, aumenta monótonamente)

Estado volátil en los líderes:
(Reinicializado después de la elección)

índice siguiente[]	para cada servidor, índice de la siguiente entrada de registro para enviar a ese servidor (inicializado en líder último índice de registro + 1)
índicedecoincidencia[]	para cada servidor, índice de la entrada de registro más alta se sabe que se replica en el servidor (inicializado en 0, aumenta monótonamente)

AppendEntries RPC

Invocado por líder para replicar entradas de registro (§5.3); también se usa como latido del corazón (§5.2).

Argumentos:

término	término del líder
ID de líder	para que el seguidor pueda redirigir el índice de clientes de la
anteriorLogIndex	entrada de registro inmediatamente anterior nuevos
prevLogTerm	término de la entrada prevLogIndex
entradas[]	entradas de registro para almacenar (vacío para el latido del corazón; puede enviar más de uno por eficiencia) índice
líderCommit	de compromiso del líder

Resultados:

término	currentTerm, para que el líder se actualice como verdadero
éxito	si el seguidor contenía coincidencias de entrada prevLogIndex y prevLogTerm

Implementación del receptor:

1. Responda falso si término < término actual (§5.1)
2. Responda falso si el registro no contiene una entrada en prevLogIndex cuyo término coincida con prevLogTerm (§5.3)
3. Si una entrada existente entra en conflicto con una nueva (mismo índice pero términos diferentes), elimine la entrada existente y todo lo que le sigue (§5.3)
4. Agregue cualquier entrada nueva que aún no esté en el registro
5. Si leaderCommit > commitIndex, establezca commitIndex = min(leaderCommit, índice de la última entrada nueva)

Solicitar Voto RPC

Invocado por los candidatos para reunir votos (§5.2).

Argumentos:

término	término del candidato
Identificación del candidato	candidato solicitando voto
lastLogIndex	índice de la última entrada de registro del candidato (§5.4)
lastLogTerm	término de la última entrada de registro del candidato (§5.4)

Resultados:

término	currentTerm, para que el candidato se actualice
votarOtorgado	verdadero significa que el candidato recibió el voto

Implementación del receptor:

1. Responda falso si término < término actual (§5.1)
2. Si votado por es nulo o candidatoId, y el registro del candidato está al menos tan actualizado como el registro del receptor, conceda el voto (§5.2, §5.4)

Reglas para servidores

Todos los servidores:

- Si commitIndex > lastApplied: incrementa lastApplied, aplica log[lastApplied] a la máquina de estado (§5.3)
- Si la solicitud o respuesta de RPC contiene el término T > currentTerm: establecer currentTerm = T, convertir a seguidor (§5.1)

Seguidores (§5.2):

- Responder a RPC de candidatos y líderes
- Si transcurre el tiempo de espera de la elección sin recibir AppendEntries RPC del líder actual o otorgar el voto al candidato: convertirse en candidato

Candidatos (§5.2):

- En la conversión a candidato, comience la elección:
  - Incrementar término actual
  - Votar por uno mismo
  - Restablecer temporizador de elecciones
  - Enviar RPC de RequestVote a todos los demás servidores
- Si se recibieron votos de la mayoría de los servidores: convertirse en líder
- Si se recibieron AppendEntries RPC del nuevo líder: convertir a seguidor
- Si transcurre el tiempo de espera de la elección: iniciar una nueva elección

Líderes:

- Tras la elección: enviar RPC de AppendEntries vacíos iniciales (latido) a cada servidor; repetir durante los períodos de inactividad para evitar tiempos de espera de elección (§5.2)
- Si se recibió un comando del cliente: agregue la entrada al registro local, responder después de la entrada aplicada a la máquina de estado (§5.3)
- Si el último índice de registro ≥ nextIndex para un seguidor: enviar AppendEntries RPC con entradas de registro que comienzan en nextIndex
  - Si tiene éxito: actualice nextIndex y matchIndex para seguidor (§5.3)
  - Si AppendEntries falla debido a la inconsistencia del registro: disminuya nextIndex y vuelva a intentarlo (§5.3)
- Si existe un N tal que N > commitIndex, la mayoría de matchIndex[i] ≥ N, y log[N].term == currentTerm: establecer commitIndex = N (§5.3, §5.4).

Figura 2:Un resumen condensado del algoritmo de consenso de Raft (excluyendo los cambios de membresía y la compactación de registros). El comportamiento del servidor en el cuadro superior izquierdo se describe como un conjunto de reglas que se activan de forma independiente y repetida. Números de sección como§5.2 indique dónde se discuten las características particulares. Una especificación formal [31] describe el algoritmo con mayor precisión.

Seguridad electoral:a lo sumo un líder puede ser elegido en un término dado.§5.2

Solo anexar líder:un líder nunca sobrescribe ni elimina entradas en su registro; solo agrega nuevas entradas.§5.3

Coincidencia de registro:si dos registros contienen una entrada con el mismo índice y término, entonces los registros son idénticos en todas las entradas hasta el índice dado.§5.3

Completitud del líder:si una entrada de registro se confirma en un término dado, entonces esa entrada estará presente en los registros de los líderes para todos los términos con números más altos.§5.4

Estado de seguridad de la máquina:si un servidor ha aplicado una entrada de registro en un índice dado a su máquina de estado, ningún otro servidor aplicará una entrada de registro diferente para el mismo índice. § 5.4.3

Figura 3:Raft garantiza que cada una de estas propiedades se cumple en todo momento. Los números de sección indican dónde se analiza cada propiedad.

de los clientes y replicarlos en todo el clúster, obligando a los otros registros a estar de acuerdo con los suyos (Sección 5.3).

- La seguridad:la propiedad de seguridad clave para Raft es la Propiedad de seguridad de la máquina de estado en la Figura 3: si algún servidor ha aplicado una entrada de registro particular a su máquina de estado, entonces ningún otro servidor puede aplicar un comando diferente para el mismo índice de registro. La Sección 5.4 describe cómo Raft garantiza esta propiedad; la solución implica una restricción adicional al mecanismo de elección descrito en la Sección 5.2.

Después de presentar el algoritmo de consenso, esta sección analiza el tema de la disponibilidad y el papel de la temporización en el sistema.

5.1 Conceptos básicos de la balsa

Un clúster de Raft contiene varios servidores; cinco es un número típico, que permite que el sistema tolere dos fallas. En un momento dado, cada servidor se encuentra en uno de estos tres estados: *líder*, *seguidor*, o*candidato*. En funcionamiento normal, hay exactamente un líder y todos los demás servidores son seguidores. Los seguidores son pasivos: no emiten solicitudes por su cuenta, sino que simplemente responden a las solicitudes de los líderes y candidatos. El líder maneja todas las solicitudes de los clientes (si un cliente contacta a un seguidor, el seguidor lo redirige al líder). El tercer estado, candidato, se utiliza para elegir un nuevo líder como se describe en la Sección 5.2. la Figura 4 muestra los estados y sus transiciones; las transiciones se analizan a continuación.

Raft divide el tiempo en*términos*de longitud arbitraria, como se muestra en la Figura 5. Los términos se numeran con números enteros consecutivos. Cada término comienza con un*elección*, en el que uno o más candidatos intentan convertirse en líderes como se describe en la Sección 5.2. Si un candidato gana la elección, se desempeña como líder por el resto del mandato. En algunas situaciones, una elección resultará en una votación dividida. En este caso el término terminará sin líder; un nuevo mandato (con una nueva elección)

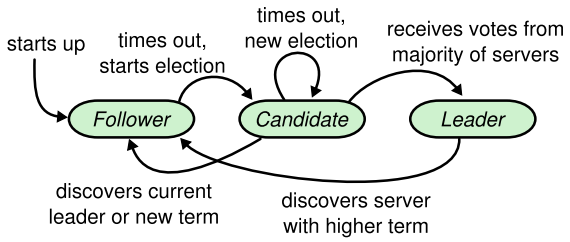

Figura 4:Estados del servidor. Los seguidores solo responden a solicitudes de otros servidores. Si un seguidor no recibe comunicación, se convierte en candidato e inicia una elección. Un candidato que recibe los votos de la mayoría del grupo completo se convierte en el nuevo líder. Los líderes suelen operar hasta que fallan.

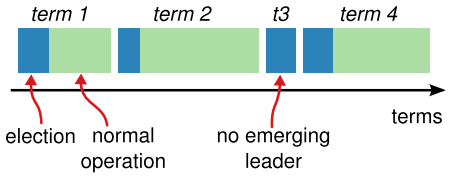

Figura 5:El tiempo se divide en términos, y cada término comienza con una elección. Después de una elección exitosa, un solo líder administra el clúster hasta el final del mandato. Algunas elecciones fracasan, en cuyo caso el mandato termina sin elegir un líder. Las transiciones entre términos pueden observarse en diferentes momentos en diferentes servidores.

comenzará en breve. Raft asegura que haya como máximo un líder en un período determinado.

Diferentes servidores pueden observar las transiciones entre términos en diferentes momentos y, en algunas situaciones, es posible que un servidor no observe una elección o incluso términos completos. Los términos actúan como un reloj lógico [14] en Raft y permiten que los servidores detecten información obsoleta, como líderes obsoletos. Cada servidor almacena un*término actual*número, que aumenta monótonamente con el tiempo. Los términos actuales se intercambian cada vez que los servidores se comunican; si el término actual de un servidor es más pequeño que el del otro, entonces actualiza su término actual al valor más grande. Si un candidato o líder descubre que su mandato está vencido, inmediatamente vuelve al estado de seguidor. Si un servidor recibe una solicitud con un número de término obsoleto, la rechaza.

Los servidores Raft se comunican mediante llamadas a procedimientos remotos (RPC), y el algoritmo de consenso básico requiere solo dos tipos de RPC. Los candidatos inician los RPC de RequestVote durante las elecciones (Sección 5.2), y los líderes inician los RPC de Anexar entradas para replicar las entradas de registro y proporcionar una forma de latido (Sección 5.3). La sección 7 agrega una tercera RPC para transferir instantáneas entre servidores. Los servidores vuelven a intentar las RPC si no reciben una respuesta de manera oportuna y emiten las RPC en paralelo para obtener el mejor rendimiento.

5.2 Elección de líder

Raft utiliza un mecanismo de latidos del corazón para desencadenar la elección del líder. Cuando los servidores se inician, comienzan como seguidores. Un servidor permanece en estado de seguidor mientras recibe información válida.

RPC de un líder o candidato. Los líderes envían latidos de corazón periódicos (RPC de AppendEntries que no llevan entradas de registro) a todos los seguidores para mantener su autoridad. Si un seguidor no recibe ninguna comunicación durante un período de tiempo denominado *tiempo de espera de las elecciones*, luego asume que no hay un líder viable y comienza una elección para elegir un nuevo líder.

Para comenzar una elección, un seguidor incrementa su mandato actual y pasa al estado de candidato. Luego vota por sí mismo y emite RPC RequestVote en paralelo a cada uno de los otros servidores en el clúster. Un candidato continúa en este estado hasta que sucede una de estas tres cosas: (a) gana la elección, (b) otro servidor se establece como líder, o (c) pasa un período de tiempo sin ganador. Estos resultados se analizan por separado en los párrafos siguientes.

Un candidato gana una elección si recibe votos de la mayoría de los servidores en el clúster completo para el mismo período. Cada servidor votará como máximo por un candidato en un período determinado, por orden de llegada (nota: la Sección 5.4 agrega una restricción adicional a los votos). La regla de la mayoría asegura que, como máximo, un candidato puede ganar la elección por un período en particular (la propiedad de seguridad electoral en la Figura 3). Una vez que un candidato gana una elección, se convierte en líder. Luego envía mensajes de latido a todos los demás servidores para establecer su autoridad y evitar nuevas elecciones.

Mientras espera los votos, un candidato puede recibir un RPC de AppendEntries de otro servidor que afirma ser líder. Si el mandato del líder (incluido en su RPC) es al menos tan largo como el mandato actual del candidato, entonces el candidato reconoce al líder como legítimo y vuelve al estado de seguidor. Si el término en el RPC es menor que el término actual del candidato, entonces el candidato rechaza el RPC y continúa en estado de candidato.

El tercer resultado posible es que un candidato no gane ni pierda la elección: si muchos seguidores se convierten en candidatos al mismo tiempo, los votos podrían dividirse para que ningún candidato obtenga la mayoría. Cuando esto suceda, cada candidato expirará y comenzará una nueva elección incrementando su mandato e iniciando otra ronda de RPC de solicitud de voto. Sin embargo, sin medidas adicionales, los votos divididos podrían repetirse indefinidamente.

Raft utiliza tiempos de espera de elecciones aleatorios para garantizar que los votos divididos sean raros y que se resuelvan rápidamente. En primer lugar, para evitar votos divididos, los tiempos de espera de las elecciones se eligen aleatoriamente a partir de un intervalo fijo (p. ej., 150–300 ms). Esto distribuye los servidores para que, en la mayoría de los casos, solo se agote el tiempo de espera de un único servidor; gana la elección y envía latidos antes de que se agote el tiempo de espera de cualquier otro servidor. El mismo mecanismo se utiliza para manejar votos divididos. Cada candidato reinicia su tiempo de espera de elección aleatorio al comienzo de una elección y espera a que transcurra ese tiempo de espera antes de comenzar la próxima elección; esto reduce la probabilidad de otro voto dividido en la nueva elección. La sección 9.3 muestra que este enfoque elige a un líder rápidamente.

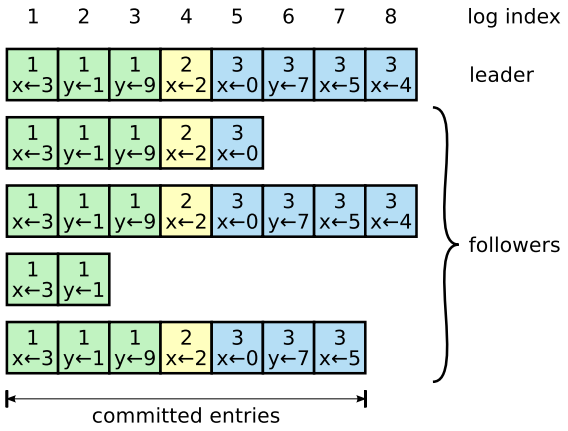

Figura 6:Los registros se componen de entradas, que se numeran secuencialmente. Cada entrada contiene el término en el que se creó (el número en cada casilla) y un comando para la máquina de estados. Se considera una entrada *comprometido* si es seguro que esa entrada se aplique a las máquinas de estado.

Las elecciones son un ejemplo de cómo la comprensibilidad guió nuestra elección entre alternativas de diseño. Inicialmente, planeamos usar un sistema de clasificación: a cada candidato se le asignaba una clasificación única, que se usaba para seleccionar entre candidatos en competencia. Si un candidato descubre a otro candidato con un rango más alto, volverá al estado de seguidor para que el candidato con un rango más alto pueda ganar más fácilmente las próximas elecciones. Descubrimos que este enfoque creaba problemas sutiles en torno a la disponibilidad (es posible que un servidor con una clasificación más baja deba expirar y volver a ser candidato si un servidor con una clasificación más alta falla, pero si lo hace demasiado pronto, puede restablecer el progreso hacia la elección de un líder).). Hicimos ajustes al algoritmo varias veces, pero después de cada ajuste aparecieron nuevos casos de esquina.

5.3 Replicación de registros

Una vez que se ha elegido un líder, comienza a atender las solicitudes de los clientes. Cada solicitud de cliente contiene un comando para ser ejecutado por las máquinas de estado replicadas. El líder agrega el comando a su registro como una nueva entrada, luego emite RPC AppendEntries en paralelo a cada uno de los otros servidores para replicar la entrada. Cuando la entrada se ha replicado de forma segura (como se describe a continuación), el líder aplica la entrada a su máquina de estado y devuelve el resultado de esa ejecución al cliente. Si los seguidores fallan o se ejecutan lentamente, o si se pierden los paquetes de red, el líder vuelve a intentar las RPC Append-Entries indefinidamente (incluso después de haber respondido al cliente) hasta que todos los seguidores finalmente almacenen todas las entradas de registro.

Los registros están organizados como se muestra en la Figura 6. Cada entrada de registro almacena un comando de máquina de estado junto con el número de término cuando el líder recibió la entrada. El término números en las entradas de registro se usa para detectar inconsistencias entre registros y para garantizar algunas de las propiedades de la Figura 3. Cada entrada de registro también tiene un índice de número entero identificador.

tificando su posición en el registro.

El líder decide cuándo es seguro aplicar una entrada de registro a las máquinas de estado; tal entrada se llama *comprometido*. Raft garantiza que las entradas comprometidas son duraderas y eventualmente serán ejecutadas por todas las máquinas de estado disponibles. Una entrada de registro se confirma una vez que el líder que creó la entrada la ha replicado en la mayoría de los servidores (por ejemplo, la entrada 7 en la Figura 6). Esto también confirma todas las entradas anteriores en el registro del líder, incluidas las entradas creadas por líderes anteriores. La Sección 5.4 analiza algunas sutilezas al aplicar esta regla después de cambios de líder, y también muestra que esta definición de compromiso es segura. El líder realiza un seguimiento del índice más alto que sabe que está comprometido, e incluye ese índice en futuros RPC de AppendEntries (incluidos los latidos) para que los otros servidores finalmente lo descubran. Una vez que un seguidor se entera de que se ha confirmado una entrada de registro, aplica la entrada a su máquina de estado local (en orden de registro).

Diseñamos el mecanismo de registro de Raft para mantener un alto nivel de coherencia entre los registros en diferentes servidores. Esto no solo simplifica el comportamiento del sistema y lo hace más predecible, sino que es un componente importante para garantizar la seguridad. Raft mantiene las siguientes propiedades, que juntas constituyen la propiedad de coincidencia de registro en la figura 3:

- Si dos entradas en diferentes registros tienen el mismo índice y término, entonces almacenan el mismo comando.
- Si dos entradas en diferentes registros tienen el mismo índice y término, entonces los registros son idénticos en todas las entradas anteriores.

La primera propiedad se deriva del hecho de que un líder crea como máximo una entrada con un índice de registro dado en un término dado, y las entradas de registro nunca cambian su posición en el registro. La segunda propiedad está garantizada por una simple verificación de coherencia realizada por AppendEntries. Al enviar un RPC de AppendEntries, el líder incluye el índice y el término de la entrada en su registro que precede inmediatamente a las nuevas entradas. Si el seguidor no encuentra una entrada en su registro con el mismo índice y término, rechaza las nuevas entradas. La verificación de coherencia actúa como un paso de inducción: el estado vacío inicial de los registros satisface la propiedad de coincidencia de registros, y la verificación de coherencia conserva la propiedad de coincidencia de registros cada vez que se amplían los registros. Como resultado, cada vez que AppendEntries regresa con éxito,

Durante el funcionamiento normal, los registros del líder y los seguidores se mantienen coherentes, por lo que la comprobación de coherencia de AppendEntries nunca falla. Sin embargo, los bloqueos del líder pueden dejar los registros inconsistentes (es posible que el líder anterior no haya replicado completamente todas las entradas en su registro). Estas inconsistencias pueden agravarse en una serie de fallas de líder y seguidor. La Figura 7 ilustra las formas en que los registros de los seguidores pueden diferir de los de un nuevo líder. Un seguidor puede

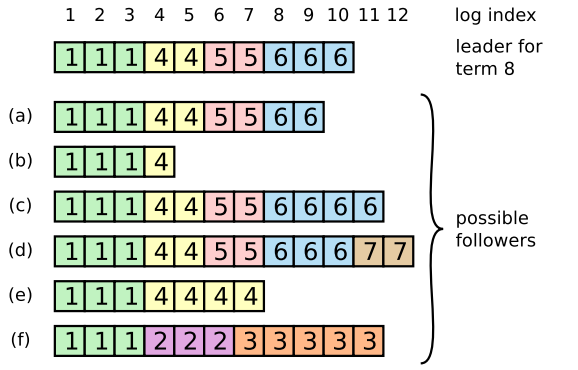

Figura 7:Cuando el líder en la parte superior llega al poder, es posible que cualquiera de los escenarios (a–f) ocurra en los registros de los seguidores. Cada cuadro representa una entrada de registro; el número en el recuadro es su término. A un seguidor le pueden faltar entradas (a–b), puede tener entradas adicionales no comprometidas (c–d) o ambas (e–f). Por ejemplo, el escenario (f) podría ocurrir si ese servidor fuera el líder para el término 2, agregara varias entradas a su registro y luego fallara antes de confirmar cualquiera de ellas; se reinició rápidamente, se convirtió en líder para el término 3 y agregó algunas entradas más a su registro; antes de que se confirmara cualquiera de las entradas en el término 2 o en el término 3, el servidor se bloqueó nuevamente y permaneció inactivo durante varios términos.

pueden faltar entradas que están presentes en el líder, puede tener entradas adicionales que no están presentes en el líder, o ambas cosas. Las entradas faltantes y extrañas en un registro pueden abarcar varios términos.

En Raft, el líder maneja las inconsistencias al obligar a los registros de los seguidores a duplicar los suyos. Esto significa que las entradas en conflicto en los registros de los seguidores se sobrescribirán con las entradas del registro del líder. La Sección 5.4 mostrará que esto es seguro cuando se combina con una restricción más.

Para que el registro de un seguidor sea coherente con el suyo propio, el líder debe encontrar la última entrada de registro donde los dos registros coincidan, eliminar cualquier entrada en el registro del seguidor después de ese punto y enviar al seguidor todas las entradas del líder después de ese punto. Todas estas acciones suceden en respuesta a la verificación de consistencia realizada por los RPC de AppendEntries. El líder mantiene una *índice siguiente* para cada seguidor, que es el índice de la siguiente entrada de registro que el líder enviará a ese seguidor. Cuando un líder llega al poder por primera vez, inicializa todos los valores de nextIndex en el índice justo después del último en su registro (11 en la Figura 7). Si el registro de un seguidor es incoherente con el del líder, la comprobación de coherencia de AppendEntries fallará en el siguiente RPC de AppendEntries. Después de un rechazo, el líder disminuye nextIndex y vuelve a intentar el RPC de AppendEntries. Eventualmente, nextIndex llegará a un punto en el que los registros de líder y seguidor coincidan. Cuando esto sucede, AppendEntries tendrá éxito, lo que elimina cualquier entrada en conflicto en el registro del seguidor y agrega entradas del registro del líder (si corresponde). Una vez que AppendEntries tiene éxito, el registro del seguidor es consistente con el del líder y permanecerá así por el resto del término.

Si lo desea, el protocolo se puede optimizar para reducir el número de RPC de AppendEntries rechazadas. Por ejemplo, al rechazar una solicitud de AppendEntries, el seguidor

puede incluir el término de la entrada en conflicto y el primer índice que almacena para ese término. Con esta información, el líder puede disminuir nextIndex para omitir todas las entradas en conflicto en ese término; se requerirá una RPC de AppendEntries para cada término con entradas en conflicto, en lugar de una RPC por entrada. En la práctica, dudamos que esta optimización sea necesaria, ya que las fallas ocurren con poca frecuencia y es poco probable que haya muchas entradas inconsistentes.

Con este mecanismo, un líder no necesita realizar ninguna acción especial para restaurar la consistencia del registro cuando se trata de poder. Simplemente comienza la operación normal y los registros convergen automáticamente en respuesta a fallas en la verificación de coherencia de Append-Entries. Un líder nunca sobrescribe ni elimina entradas en su propio registro (la propiedad de solo agregar líder en la Figura 3).

Este mecanismo de replicación de registros exhibe las propiedades de consenso deseables descritas en la Sección 2: Raft puede aceptar, replicar y aplicar nuevas entradas de registros siempre que la mayoría de los servidores estén activos; en el caso normal, una nueva entrada se puede replicar con una sola ronda de RPC para la mayoría del grupo; y un solo seguidor lento no afectará el rendimiento.

5.4 Seguridad

Las secciones anteriores describieron cómo Raft elige a los líderes y replica las entradas de registro. Sin embargo, los mecanismos descritos hasta ahora no son suficientes para garantizar que cada máquina de estado ejecute exactamente los mismos comandos en el mismo orden. Por ejemplo, un seguidor podría no estar disponible mientras el líder confirma varias entradas de registro, luego podría ser elegido líder y sobrescribir estas entradas con otras nuevas; como resultado, diferentes máquinas de estado pueden ejecutar diferentes secuencias de comandos.

Esta sección completa el algoritmo Raft agregando una restricción sobre qué servidores pueden ser elegidos líderes. La restricción asegura que el líder para cualquier término dado contenga todas las entradas comprometidas en términos anteriores (la propiedad de completitud del líder de la Figura 3). Dada la restricción de elección, entonces hacemos más precisas las reglas para el compromiso. Finalmente, presentamos un boceto de prueba para la propiedad de integridad del líder y mostramos cómo conduce al comportamiento correcto de la máquina de estado replicada.

5.4.1 Restricción de elección

En cualquier algoritmo de consenso basado en líder, el líder finalmente debe almacenar todas las entradas de registro comprometidas. En algunos algoritmos de consenso, como Viewstamped Replication [22], se puede elegir un líder incluso si inicialmente no contiene todas las entradas comprometidas. Estos algoritmos contienen mecanismos adicionales para identificar las entradas faltantes y transmitirlas al nuevo líder, ya sea durante el proceso de elección o poco tiempo después. Desafortunadamente, esto da como resultado un mecanismo y una complejidad adicionales considerables. Raft utiliza un enfoque más simple en el que garantiza que todas las entradas comprometidas de anteriores

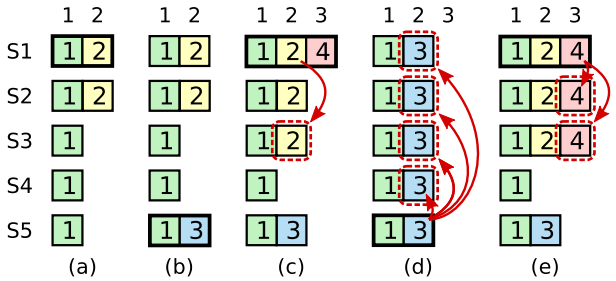

Figura 8:Una secuencia de tiempo que muestra por qué un líder no puede determinar el compromiso utilizando entradas de registro de términos anteriores. En (a) S1 es líder y replica parcialmente la entrada de registro en el índice 2. En (b) S1 falla; S5 es elegido líder para el término 3 con votos de S3, S4 y de sí mismo, y acepta una entrada diferente en el índice de registro 2. En (c) S5 falla; S1 se reinicia, es elegido líder y continúa la replicación. En este punto, la entrada de registro del término 2 se ha replicado en la mayoría de los servidores, pero no se ha confirmado. Si S1 falla como en (d), S5 podría ser elegido líder (con votos de S2, S3 y S4) y sobrescribir la entrada con su propia entrada del término 3. Sin embargo, si S1 replica una entrada de su término actual en un la mayoría de los servidores antes de fallar, como en (e), entonces esta entrada se confirma (S5 no puede ganar una elección). En este punto, todas las entradas anteriores en el registro también se confirman.

los términos están presentes en cada nuevo líder desde el momento de su elección, sin necesidad de transferir esas entradas al líder. Esto significa que las entradas de registro solo fluyen en una dirección, de los líderes a los seguidores, y los líderes nunca sobrescriben las entradas existentes en sus registros.

Raft utiliza el proceso de votación para evitar que un candidato gane una elección a menos que su registro contenga todas las entradas comprometidas. Un candidato debe comunicarse con la mayoría del grupo para ser elegido, lo que significa que cada entrada confirmada debe estar presente en al menos uno de esos servidores. Si el registro del candidato está al menos tan actualizado como cualquier otro registro en esa mayoría (donde "actualizado" se define precisamente a continuación), entonces contendrá todas las entradas confirmadas. El RequestVote RPC implementa esta restricción: el RPC incluye información sobre el registro del candidato y el votante niega su voto si su propio registro está más actualizado que el del candidato.

Raft determina cuál de los dos registros está más actualizado al comparar el índice y el término de las últimas entradas en los registros. Si los registros tienen últimas entradas con diferentes términos, entonces el registro con el último término está más actualizado. Si los registros terminan con el mismo término, el registro que sea más largo estará más actualizado.

5.4.2 Confirmación de entradas de términos anteriores

Como se describe en la Sección 5.3, un líder sabe que una entrada de su término actual se confirma una vez que esa entrada se almacena en la mayoría de los servidores. Si un líder falla antes de confirmar una entrada, los futuros líderes intentarán terminar de replicar la entrada. Sin embargo, un líder no puede concluir de inmediato que una entrada de un período anterior está comprometida una vez que se almacena en la mayoría de los servidores. Higo-

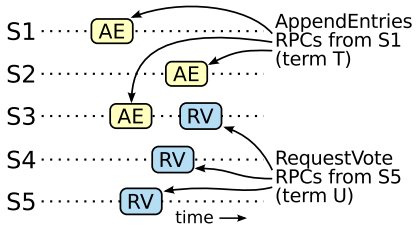

Figura 9:Si S1 (líder para el término T) confirma una nueva entrada de registro de su término y S5 es elegido líder para un término U posterior, entonces debe haber al menos un servidor (S3) que aceptó la entrada de registro y también votó por S5.

La figura 8 ilustra una situación en la que una entrada de registro antigua se almacena en la mayoría de los servidores, pero aún puede ser sobrescrita por un futuro líder.

Para eliminar problemas como el de la Figura 8, Raft nunca confirma entradas de registro de períodos anteriores contando réplicas. Solo las entradas de registro del mandato actual del líder se confirman mediante el conteo de réplicas; una vez que una entrada del término actual se ha confirmado de esta manera, todas las entradas anteriores se confirman indirectamente debido a la propiedad de coincidencia de registros. Hay algunas situaciones en las que un líder podría concluir con seguridad que se ha confirmado una entrada de registro más antigua (por ejemplo, si esa entrada está almacenada en todos los servidores), pero Raft adopta un enfoque más conservador por simplicidad.

Raft incurre en esta complejidad adicional en las reglas de compromiso porque las entradas de registro conservan sus números de término originales cuando un líder replica entradas de términos anteriores. En otros algoritmos de consenso, si un nuevo líder vuelve a replicar entradas de "términos" anteriores, debe hacerlo con su nuevo "número de término". El enfoque de Raft facilita el razonamiento sobre las entradas de registro, ya que mantienen el mismo número de términos a lo largo del tiempo y en todos los registros. Además, los nuevos líderes en Raft envían menos entradas de registro de términos anteriores que en otros algoritmos (otros algoritmos deben enviar entradas de registro redundantes para volver a numerarlas antes de que puedan confirmarse).

5.4.3 Argumento de seguridad

Dado el algoritmo Raft completo, ahora podemos argumentar con mayor precisión que se cumple la propiedad de integridad del líder (este argumento se basa en la prueba de seguridad; consulte la Sección 9.2). Suponemos que la propiedad de integridad del líder no se cumple, luego demostramos una contradicción. Supongamos que el líder para el término T (líder$_T$) confirma una entrada de registro de su término, pero esa entrada de registro no es almacenada por el líder de algún término futuro. Considere el término más pequeño U > T cuyo líder (líder$_U$) no almacena la entrada.

1. La entrada confirmada debe haber estado ausente del líder$_U$de registro en el momento de su elección (los líderes nunca borran ni sobrescriben entradas).
2. líder$_T$replicó la entrada en la mayoría del grupo, y el líder$_U$recibió los votos de la mayoría del grupo. Por lo tanto, al menos un servidor ("el votante") aceptó la entrada del líder$_T$y votado por líder$_U$, como se muestra en la Figura 9. El votante es clave para llegar a una contradicción.
3. El votante debe haber aceptado la entrada comprometida del líder$_T$*antes de*votando por el lider$_U$; de lo contrario, habría rechazado la solicitud AppendEntries del líder$_T$(su término actual habría sido mayor que T).
4. El votante todavía almacenó la entrada cuando votó por el líder$_U$, dado que cada líder que interviene contenía la entrada (por suposición), los líderes nunca eliminan las entradas y los seguidores solo eliminan las entradas si entran en conflicto con el líder.
5. El votante concedió su voto al líder$_U$, así que líder$_U$El registro de debe haber estado tan actualizado como el del votante. Esto conduce a una de dos contradicciones.
6. Primero, si el votante y líder$_U$compartió el mismo último término logarítmico, luego líder$_U$El registro de debe haber sido al menos tan largo como el del votante, por lo que su registro contenía todas las entradas en el registro del votante. Esto es una contradicción, ya que el votante contenía la entrada comprometida y el líder$_U$se suponía que no.
7. De lo contrario, líder$_U$El último término logarítmico de debe haber sido mayor que el del votante. Además, era más grande que T, ya que el último término de registro del votante fue al menos T (contiene la entrada comprometida del término T). El líder anterior que creó líder$_U$La última entrada de registro debe haber contenido la entrada confirmada en su registro (por suposición). Luego, por la propiedad de coincidencia de registros, líder$_U$El registro también debe contener la entrada confirmada, lo cual es una contradicción.
8. Esto completa la contradicción. Por lo tanto, los encabezados de todos los términos mayores que T deben contener todas las entradas del término T que están comprometidas en el término T.
9. La propiedad de coincidencia de registros garantiza que los futuros líderes también contengan entradas comprometidas indirectamente, como el índice 2 en la figura 8(d).

Dada la propiedad de integridad del líder, podemos probar la propiedad de seguridad de la máquina de estado de la figura 3, que establece que si un servidor ha aplicado una entrada de registro en un índice dado a su máquina de estado, ningún otro servidor aplicará una entrada de registro diferente para la máquina de estado. mismo índice. En el momento en que un servidor aplica una entrada de registro a su máquina de estado, su registro debe ser idéntico al registro del líder a través de esa entrada y la entrada debe confirmarse. Ahora considere el término más bajo en el que cualquier servidor aplica un índice de registro dado; la propiedad Log Completeness garantiza que los líderes de todos los términos superiores almacenarán la misma entrada de registro, por lo que los servidores que aplican el índice en términos posteriores aplicarán el mismo valor. Por lo tanto, se mantiene la propiedad de seguridad de la máquina de estado.

Finalmente, Raft requiere que los servidores apliquen las entradas en orden de índice de registro. Combinado con la propiedad de seguridad de la máquina de estado, esto significa que todos los servidores aplicarán exactamente el mismo conjunto de entradas de registro a sus máquinas de estado, en el mismo orden.

5.5 Bloqueos de seguidores y candidatos

Hasta este punto nos hemos centrado en las fallas de los líderes. Los bloqueos de seguidores y candidatos son mucho más simples de manejar que los bloqueos de líderes, y ambos se manejan de la misma manera. Si un seguidor o candidato falla, los futuros RPC de RequestVote y AppendEntries que se le envíen fallarán. Raft maneja estos errores volviendo a intentarlo indefinidamente; si el servidor bloqueado se reinicia, la RPC se completará correctamente. Si un servidor falla después de completar un RPC pero antes de responder, recibirá el mismo RPC nuevamente después de reiniciarse. Los RPC de balsa son idempotentes, por lo que esto no causa daño. Por ejemplo, si un seguidor recibe una solicitud de AppendEntries que incluye entradas de registro que ya están presentes en su registro, ignora esas entradas en la nueva solicitud.

5.6 Calendario y disponibilidad

Uno de nuestros requisitos para Raft es que la seguridad no debe depender del tiempo: el sistema no debe producir resultados incorrectos solo porque algún evento ocurre más rápido o más lento de lo esperado. Sin embargo, la disponibilidad (la capacidad del sistema para responder a los clientes de manera oportuna) debe depender inevitablemente del tiempo. Por ejemplo, si los intercambios de mensajes demoran más que el tiempo típico entre fallas del servidor, los candidatos no permanecerán despiertos el tiempo suficiente para ganar una elección; sin un líder firme, Raft no puede progresar.

La elección del líder es el aspecto de Raft donde el tiempo es más crítico. Raft podrá elegir y mantener un líder estable siempre que el sistema cumpla con los siguientes *requisito de tiempo*:

$$\textit{tiempo de emisión} \ll \textit{tiempo de espera de las elecciones} \ll \textit{MTBF}$$

En esta desigualdad *tiempo de emisión* es el tiempo medio que tarda un servidor en enviar RPC en paralelo a todos los servidores del clúster y recibir sus respuestas; *tiempo de espera de las elecciones* es el tiempo de espera de elección descrito en la Sección 5.2; y *MTBF* es el tiempo promedio entre fallas para un solo servidor. El tiempo de transmisión debe ser un orden de magnitud menor que el tiempo de espera de las elecciones para que los líderes puedan enviar de manera confiable los mensajes de latido necesarios para evitar que los seguidores comiencen las elecciones; dado el enfoque aleatorio utilizado para los tiempos de espera de las elecciones, esta desigualdad también hace que los votos divididos sean poco probables. El tiempo de espera de las elecciones debe ser unos pocos órdenes de magnitud menor que el MTBF para que el sistema progrese de manera constante. Cuando el líder falla, el sistema no estará disponible durante aproximadamente el tiempo de espera de las elecciones; nos gustaría que esto representara solo una pequeña fracción del tiempo total.

El tiempo de transmisión y el MTBF son propiedades del sistema subyacente, mientras que el tiempo de espera de elección es algo que debemos elegir. Los RPC de Raft generalmente requieren que el destinatario conserve la información en un almacenamiento estable, por lo que el tiempo de transmisión puede oscilar entre 0,5 ms y 20 ms, según la tecnología de almacenamiento. Como resultado, es probable que el tiempo de espera de la elección esté entre 10 ms y 500 ms. Típico los MTBF del servidor son de varios meses o más, lo que satisface fácilmente el requisito de tiempo.

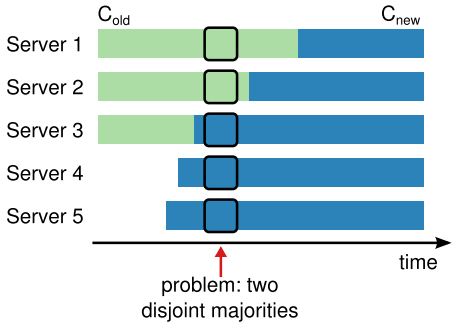

Figura 10:Cambiar directamente de una configuración a otra no es seguro porque diferentes servidores cambiarán en diferentes momentos. En este ejemplo, el clúster crece de tres servidores a cinco. Desafortunadamente, hay un momento en el que se pueden elegir dos líderes diferentes para el mismo período, uno con una mayoría de la configuración anterior ($C_{antiguo}$) y otro con una mayoría de la nueva configuración (C_{nuevo}).

6 Cambios en la membresía del clúster

Hasta ahora hemos asumido que el clúster *configuración* (el conjunto de servidores que participan en el algoritmo de consenso) es fijo. En la práctica, ocasionalmente será necesario cambiar la configuración, por ejemplo, para reemplazar los servidores cuando fallan o para cambiar el grado de replicación. Aunque esto se puede hacer desconectando todo el clúster, actualizando los archivos de configuración y luego reiniciando el clúster, esto dejaría al clúster no disponible durante el cambio. Además, si hay pasos manuales, corren el riesgo de error del operador. Para evitar estos problemas, decidimos automatizar los cambios de configuración e incorporarlos al algoritmo de consenso de Raft.

Para que el mecanismo de cambio de configuración sea seguro, no debe haber ningún punto durante la transición en el que sea posible que se elijan dos líderes para el mismo período. Desafortunadamente, cualquier enfoque en el que los servidores cambien directamente de la configuración anterior a la nueva configuración no es seguro. No es posible cambiar de forma atómica todos los servidores a la vez, por lo que el clúster puede potencialmente dividirse en dos mayorías independientes durante la transición (consulte la Figura 10).

Para garantizar la seguridad, los cambios de configuración deben utilizar un enfoque de dos fases. Hay una variedad de formas de implementar las dos fases. Por ejemplo, algunos sistemas (p. ej., [22]) utilizan la primera fase para deshabilitar la configuración anterior para que no pueda procesar las solicitudes de los clientes; luego la segunda fase habilita la nueva configuración. En Raft, el clúster primero cambia a una configuración de transición que llamamos *consenso conjunto*; una vez que se ha comprometido el consenso conjunto, el sistema pasa a la nueva configuración. El consenso conjunto combina tanto la configuración antigua como la nueva:

- Las entradas de registro se replican en todos los servidores en ambas configuraciones.

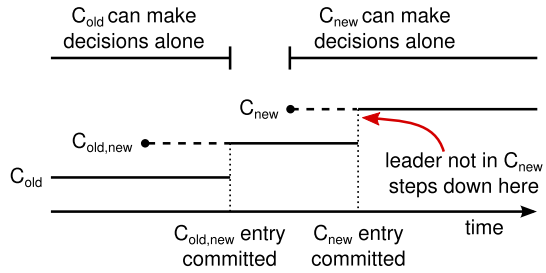

Figura 11:Línea de tiempo para un cambio de configuración. Las líneas discontinuas muestran las entradas de configuración que se crearon pero no se confirmaron y las líneas continuas muestran la última entrada de configuración confirmada. El líder primero crea el $C_{\text{viejo nuevo}}$entrada de configuración en su registro y lo compromete a $C_{\text{viejo nuevo}}$(una mayoría de C_{antiguo} y una mayoria de C_{nuevo}). Entonces crea el C_{nuevo}entrada y lo compromete a una mayoría de C_{nuevo}. No hay un momento en el tiempo en el que C_{antiguo}y C_{nuevo}Ambos pueden tomar decisiones de forma independiente.

- Cualquier servidor de cualquier configuración puede servir como líder.
- Acuerdo (para elecciones y compromiso de entrada) requiere mayorías separadas de *ambas cosas* las configuraciones antiguas y nuevas.

El consenso conjunto permite que los servidores individuales realicen la transición entre configuraciones en diferentes momentos sin comprometer la seguridad. Además, el consenso conjunto permite que el clúster continúe atendiendo las solicitudes de los clientes durante el cambio de configuración.

Las configuraciones de clúster se almacenan y comunican mediante entradas especiales en el registro replicado; La Figura 11 ilustra el proceso de cambio de configuración. Cuando el líder recibe una solicitud para cambiar la configuración de C_{antiguo} a C_{nuevo}, almacena la configuración para consenso conjunto ($C_{\text{viejo nuevo}}$ en la figura) como una entrada de registro y replica esa entrada utilizando los mecanismos descritos anteriormente. Una vez que un servidor determinado agrega la nueva entrada de configuración a su registro, usa esa configuración para todas las decisiones futuras (un servidor siempre usa la última configuración en su registro, independientemente de si la entrada está confirmada). Esto significa que el líder usará las reglas de $C_{\text{viejo nuevo}}$para determinar cuándo la entrada de registro para $C_{\text{viejo nuevo}}$se ha comprometido. Si el líder falla, se puede elegir un nuevo líder en cualquiera de los dos C_{antiguo}o $C_{\text{viejo nuevo}}$, dependiendo de si el candidato ganador ha recibido $C_{\text{viejo nuevo}}$. En todo caso, C_{nuevo}no puede tomar decisiones unilaterales durante este período.

Una vez $C_{\text{viejo nuevo}}$se ha cometido, tampoco C_{antiguo}ni C_{nuevo} puede tomar decisiones sin la aprobación del otro, y la propiedad de completitud del líder asegura que solo los servidores con la $C_{\text{viejo nuevo}}$ la entrada de registro se puede elegir como líder. Ahora es seguro para el líder crear una entrada de registro que describa C_{nuevo}y replicarlo en el clúster. Nuevamente, esta configuración tendrá efecto en cada servidor tan pronto como se vea. Cuando la nueva configuración haya sido comprometida bajo las reglas de C_{nuevo}, la configuración anterior es irrelevante y los servidores que no están en la nueva configuración se pueden apagar. Como se muestra en la Figura 11, no hay tiempo cuando C_{antiguo}y C_{nuevo} ambos pueden tomar decisiones unilaterales; esto garantiza la seguridad.

Hay tres problemas más que abordar para la reconfiguración. El primer problema es que es posible que los nuevos servidores no almacenen inicialmente ninguna entrada de registro. Si se agregan al clúster en este estado, es posible que tarden bastante en ponerse al día, tiempo durante el cual es posible que no sea posible confirmar nuevas entradas de registro. Para evitar brechas de disponibilidad, Raft introduce una fase adicional antes del cambio de configuración, en la que los nuevos servidores se unen al clúster como miembros sin derecho a voto (el líder les replica las entradas de registro, pero no se consideran para la mayoría). Una vez que los nuevos servidores se han puesto al día con el resto del clúster, la reconfiguración puede continuar como se describe anteriormente.

El segundo problema es que el líder del clúster puede no ser parte de la nueva configuración. En este caso, el líder se retira (regresa al estado de seguidor) una vez que ha cometido el C_{nuevo}entrada de registro. Esto significa que habrá un período de tiempo (mientras se está cometiendo C_{nuevo}) cuando el líder está gestionando un clúster que no se incluye a sí mismo; replica las entradas de registro pero no se cuenta a sí mismo en mayorías. La transición de líder ocurre cuando C_{nuevo}se compromete porque este es el primer punto en el que la nueva configuración puede operar de forma independiente (siempre será posible elegir un líder de C_{nuevo}). Antes de este punto, puede darse el caso de que sólo un servidor de C_{antiguo}puede ser elegido líder.

El tercer problema es que los servidores eliminados (aquellos que no están en C_{nuevo}) puede interrumpir el clúster. Estos servidores no recibirán latidos, por lo que se agotarán y comenzarán nuevas elecciones. Luego enviarán RPC de RequestVote con nuevos números de término, y esto hará que el líder actual vuelva al estado de seguidor. Eventualmente, se elegirá un nuevo líder, pero los servidores eliminados volverán a agotarse y el proceso se repetirá, lo que dará como resultado una disponibilidad deficiente.

Para evitar este problema, los servidores ignoran los RPC de RequestVote cuando creen que existe un líder actual. Específicamente, si un servidor recibe un RequestVote RPC dentro del tiempo de espera mínimo de elección para escuchar a un líder actual, no actualiza su término ni otorga su voto. Esto no afecta las elecciones normales, donde cada servidor espera al menos un tiempo de espera de elección mínimo antes de comenzar una elección. Sin embargo, ayuda a evitar las interrupciones de los servidores eliminados: si un líder puede enviar latidos a su clúster, entonces no será depuesto por un mayor número de términos.

7 Compactación de troncos

El registro de Raft crece durante el funcionamiento normal para incorporar más solicitudes de clientes, pero en un sistema práctico, no puede crecer sin límite. A medida que el registro crece, ocupa más espacio y lleva más tiempo reproducirlo. Esto eventualmente causará problemas de disponibilidad sin algún mecanismo para descartar información obsoleta que se ha acumulado en el registro.

La creación de instantáneas es el enfoque más simple para la compactación. En la creación de instantáneas, todo el estado actual del sistema se escribe en un *instantánea* en almacenamiento estable, luego todo el registro hasta

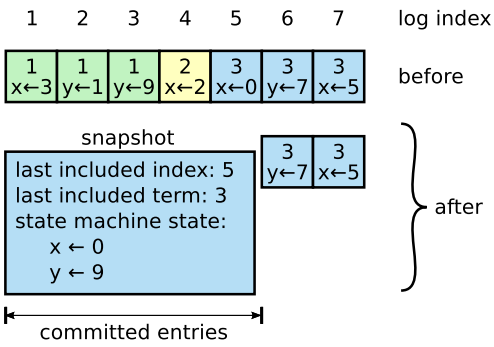

Figura 12:Un servidor reemplaza las entradas confirmadas en su registro (índices 1 a 5) con una nueva instantánea, que almacena solo el estado actual (variables*x*y*y*en este ejemplo). El último índice y término incluidos en la instantánea sirven para posicionar la instantánea en el registro que precede a la entrada 6.

ese punto se descarta. La creación de instantáneas se usa en Chubby y ZooKeeper, y el resto de esta sección describe la creación de instantáneas en Raft.

También son posibles enfoques incrementales para la compactación, como la limpieza de troncos [36] y los árboles de combinación con estructura de troncos [30, 5]. Estos operan en una fracción de los datos a la vez, por lo que distribuyen la carga de compactación de manera más uniforme a lo largo del tiempo. Primero seleccionan una región de datos que ha acumulado muchos objetos eliminados y sobrescritos, luego reescriben los objetos vivos de esa región de manera más compacta y liberan la región. Esto requiere un mecanismo y una complejidad adicionales significativos en comparación con la creación de instantáneas, lo que simplifica el problema al operar siempre en todo el conjunto de datos. Si bien la limpieza de registros requeriría modificaciones en Raft, las máquinas de estado pueden implementar árboles LSM utilizando la misma interfaz que las instantáneas.

La Figura 12 muestra la idea básica de tomar instantáneas en Raft. Cada servidor toma instantáneas de forma independiente, cubriendo solo las entradas confirmadas en su registro. La mayor parte del trabajo consiste en que la máquina de estado escriba su estado actual en la instantánea. Raft también incluye una pequeña cantidad de metadatos en la instantánea: el*último índice incluido*es el índice de la última entrada en el registro que reemplaza la instantánea (la última entrada que aplicó la máquina de estado), y el*último término incluido*es el término de esta entrada. Estos se conservan para admitir la comprobación de coherencia de AppendEntries para la primera entrada de registro que sigue a la instantánea, ya que esa entrada necesita un índice y un término de registro anteriores. Para habilitar los cambios de membresía del clúster (Sección 6), la instantánea también incluye la configuración más reciente en el registro a partir del último índice incluido. Una vez que un servidor completa la escritura de una instantánea, puede eliminar todas las entradas de registro hasta el último índice incluido, así como cualquier instantánea anterior.

Aunque los servidores normalmente toman instantáneas de forma independiente, el líder ocasionalmente debe enviar instantáneas a los seguidores que se quedan atrás. Esto sucede cuando el líder ya ha descartado la siguiente entrada de registro que debe enviar a un seguidor. Afortunadamente, esta situación es poco probable en el funcionamiento normal: un seguidor que se ha mantenido al día con el

Instalar Snapshot RPC

Invocado por el líder para enviar fragmentos de una instantánea a un seguidor. Los líderes siempre envían fragmentos en orden.

Argumentos:

término	término del líder
ID de líder	para que el seguidor pueda redirigir a los clientes
últimoÍndiceIncluido	la instantánea reemplaza todas las entradas hasta e incluyendo este índice
último término incluido	término de lastIncludedIndex
compensar	desplazamiento de bytes donde se coloca el fragmento en el archivo de instantánea
datos[]	bytes sin procesar del fragmento de instantánea, comenzando en compensar
hecho	cierto si este es el último trozo

Resultados:

término	currentTerm, para que el líder se actualice

Implementación del receptor:

1. Responda inmediatamente si el término < término actual
2. Cree un nuevo archivo de instantánea si es el primer fragmento (el desplazamiento es 0)
3. Escribir datos en el archivo de instantánea en el desplazamiento dado
4. Responda y espere más fragmentos de datos si el hecho es falso
5. Guarde el archivo de instantánea, descarte cualquier instantánea existente o parcial con un índice más pequeño
6. Si la entrada de registro existente tiene el mismo índice y término que la última entrada incluida de la instantánea, conserve las entradas de registro que le siguen y responda
7. Deseche todo el registro
8. Restablecer la máquina de estado utilizando el contenido de la instantánea (y cargar la configuración del clúster de la instantánea)

Figura 13:Un resumen de InstallSnapshot RPC. Las instantáneas se dividen en fragmentos para su transmisión; esto le da al seguidor una señal de vida con cada fragmento, por lo que puede restablecer su temporizador de elección.

líder ya tendría esta entrada. Sin embargo, un seguidor excepcionalmente lento o un nuevo servidor que se une al clúster (Sección 6) no lo harían. La forma de actualizar a un seguidor de este tipo es que el líder le envíe una instantánea a través de la red.

El líder usa un nuevo RPC llamado InstallSnapshot para enviar instantáneas a los seguidores que están demasiado atrasados; consulte la Figura 13. Cuando un seguidor recibe una instantánea con este RPC, debe decidir qué hacer con sus entradas de registro existentes. Por lo general, la instantánea contendrá información nueva que aún no está en el registro del destinatario. En este caso, el seguidor descarta todo su registro; todo es reemplazado por la instantánea y es posible que tenga entradas no confirmadas que entren en conflicto con la instantánea. Si, en cambio, el seguidor recibe una instantánea que describe un prefijo de su registro (debido a una retransmisión o por error), las entradas del registro cubiertas por la instantánea se eliminan, pero las entradas que siguen a la instantánea siguen siendo válidas y deben conservarse.

Este enfoque de instantáneas se aparta del principio de líder fuerte de Raft, ya que los seguidores pueden tomar instantáneas sin el conocimiento del líder. Sin embargo, creemos que esta salida está justificada. Si bien tener un líder ayuda a evitar decisiones conflictivas para llegar a un consenso, ya se ha alcanzado el consenso al momento de tomar una instantánea, por lo que no hay conflicto de decisiones. Los datos todavía solo fluyen de los líderes a los seguidores.

reduce, solo los seguidores ahora pueden reorganizar sus datos.

Consideramos un enfoque alternativo basado en el líder en el que solo el líder crearía una instantánea y luego enviaría esta instantánea a cada uno de sus seguidores. Sin embargo, esto tiene dos desventajas. En primer lugar, enviar la instantánea a cada seguidor desperdiciaría el ancho de banda de la red y ralentizaría el proceso de creación de instantáneas. Cada seguidor ya tiene la información necesaria para producir sus propias instantáneas y, por lo general, es mucho más económico para un servidor producir una instantánea desde su estado local que enviar y recibir una a través de la red. En segundo lugar, la implementación del líder sería más compleja. Por ejemplo, el líder necesitaría enviar instantáneas a los seguidores en paralelo con la replicación de nuevas entradas de registro para no bloquear las solicitudes de nuevos clientes.

Hay dos problemas más que afectan el rendimiento de las instantáneas. Primero, los servidores deben decidir cuándo tomar una instantánea. Si un servidor toma instantáneas con demasiada frecuencia, desperdicia ancho de banda y energía del disco; si toma instantáneas con poca frecuencia, corre el riesgo de agotar su capacidad de almacenamiento y aumenta el tiempo necesario para reproducir el registro durante los reinicios. Una estrategia simple es tomar una instantánea cuando el registro alcanza un tamaño fijo en bytes. Si este tamaño se configura para que sea significativamente mayor que el tamaño esperado de una instantánea, la sobrecarga del ancho de banda del disco para la creación de instantáneas será pequeña.

El segundo problema de rendimiento es que escribir una instantánea puede llevar mucho tiempo y no queremos que esto retrase las operaciones normales. La solución es utilizar técnicas de copia en escritura para que se puedan aceptar nuevas actualizaciones sin afectar la instantánea que se está escribiendo. Por ejemplo, las máquinas de estado construidas con estructuras de datos funcionales soportan esto de forma natural. Alternativamente, el soporte de copia en escritura del sistema operativo (por ejemplo, bifurcación en Linux) se puede usar para crear una instantánea en memoria de toda la máquina de estado (nuestra implementación usa este enfoque).

8 Interacción con el cliente

Esta sección describe cómo los clientes interactúan con Raft, incluido cómo los clientes encuentran el líder del clúster y cómo Raft admite la semántica linealizable [10]. Estos problemas se aplican a todos los sistemas basados en el consenso, y las soluciones de Raft son similares a las de otros sistemas.

Los clientes de Raft envían todas sus solicitudes al líder. Cuando un cliente se inicia por primera vez, se conecta a un servidor elegido al azar. Si la primera opción del cliente no es el líder, ese servidor rechazará la solicitud del cliente y proporcionará información sobre el líder más reciente del que ha tenido noticias (las solicitudes de AppendEntries incluyen la dirección de red del líder). Si el líder falla, las solicitudes de los clientes expirarán; los clientes vuelven a intentarlo con servidores elegidos al azar.

Nuestro objetivo para Raft es implementar una semántica linealizable (cada operación parece ejecutarse instantáneamente, exactamente una vez, en algún punto entre su invocación y su respuesta). Sin embargo, como se ha descrito hasta ahora, Raft puede ejecutar un comando varias veces: por ejemplo, si el líder se bloquea después de confirmar la entrada de registro, pero antes de responder al cliente, el cliente volverá a intentar el comando con un nuevo líder, lo que hará que se ejecute por segunda vez. La solución es que los clientes asignen números de serie únicos a cada comando. Luego, la máquina de estado rastrea el último número de serie procesado para cada cliente, junto con la respuesta asociada. Si recibe un comando cuyo número de serie ya ha sido ejecutado, responde inmediatamente sin volver a ejecutar la solicitud.

Las operaciones de solo lectura se pueden manejar sin escribir nada en el registro. Sin embargo, sin medidas adicionales, esto correría el riesgo de devolver datos obsoletos, ya que el líder que responde a la solicitud podría haber sido reemplazado por un líder más nuevo del que no tiene conocimiento. Las lecturas linealizables no deben devolver datos obsoletos y Raft necesita dos precauciones adicionales para garantizar esto sin usar el registro. Primero, un líder debe tener la información más reciente sobre qué entradas están comprometidas. La propiedad de integridad del líder garantiza que un líder tiene todas las entradas comprometidas, pero al comienzo de su período, es posible que no sepa cuáles son. Para averiguarlo, necesita cometer una entrada de su término. Raft maneja esto haciendo que cada líder cometa un espacio en blanco*sin operaciones* inscripción en el registro al inicio de su vigencia. En segundo lugar, un líder debe verificar si ha sido depuesto antes de procesar una solicitud de solo lectura (su información puede estar obsoleta si se eligió a un líder más reciente). Raft maneja esto haciendo que el líder intercambie mensajes de latido con la mayoría del clúster antes de responder a las solicitudes de solo lectura. Alternativamente, el líder podría confiar en el mecanismo de latido del corazón para proporcionar una forma de arrendamiento [9], pero esto dependería del tiempo por seguridad (supone un sesgo de reloj acotado).

9 Implementación y evaluación

Implementamos Raft como parte de una máquina de estado replicada que almacena información de configuración para RAMCloud [33] y ayuda en la conmutación por error del coordinador de RAMCloud. La implementación de Raft contiene aproximadamente 2000 líneas de código C++, sin incluir pruebas, comentarios o líneas en blanco. El código fuente está disponible gratuitamente [23]. También hay alrededor de 25 implementaciones independientes de código abierto de terceros [34] de Raft en varias etapas de desarrollo, según los borradores de este documento. Además, varias empresas están implementando sistemas basados en Raft [34].

El resto de esta sección evalúa Raft utilizando tres criterios: comprensibilidad, corrección y rendimiento.

9.1 Comprensibilidad

Para medir la comprensibilidad de Raft en relación con Paxos, realizamos un estudio experimental con estudiantes universitarios y graduados de nivel superior en un curso de Sistemas operativos avanzados en la Universidad de Stanford y un curso de Computación distribuida en UC Berkeley. Grabamos una conferencia en video de Raft y otra de Paxos, y creamos los cuestionarios correspondientes. La conferencia de Raft cubrió el contenido de este documento excepto la compactación de troncos; los paxos

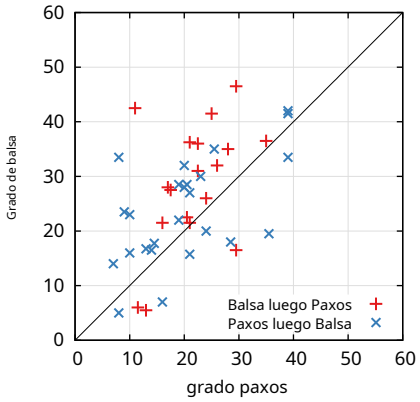

Figura 14:Un diagrama de dispersión que compara el desempeño de 43 participantes en las pruebas Raft y Paxos. Los puntos por encima de la diagonal (33) representan a los participantes que obtuvieron una puntuación más alta en Raft.

La conferencia abarcó suficiente material para crear una máquina de estado replicada equivalente, incluidos Paxos de un solo decreto, Paxos de múltiples decretos, reconfiguración y algunas optimizaciones necesarias en la práctica (como la elección de líder). Los cuestionarios evaluaron la comprensión básica de los algoritmos y también requirieron que los estudiantes razonen sobre casos extremos. Cada estudiante vio un video, tomó la prueba correspondiente, vio el segundo video y tomó la segunda prueba. Alrededor de la mitad de los participantes hizo primero la parte de Paxos y la otra mitad hizo primero la parte de Raft para tener en cuenta las diferencias individuales en el rendimiento y la experiencia adquirida en la primera parte del estudio. Comparamos las puntuaciones de los participantes en cada prueba para determinar si los participantes mostraban una mejor comprensión de Raft.

Intentamos que la comparación entre Paxos y Raft fuera lo más justa posible. El experimento favoreció a Paxos de dos maneras: 15 de los 43 participantes informaron haber tenido alguna experiencia previa con Paxos, y el video de Paxos es un 14 % más largo que el video de Raft. Como se resume en la Tabla 1, hemos tomado medidas para mitigar las posibles fuentes de sesgo. Todos nuestros materiales están disponibles para su revisión [28, 31].

En promedio, los participantes obtuvieron 4,9 puntos más en el cuestionario Raft que en el cuestionario Paxos (de 60 puntos posibles, el puntaje promedio de Raft fue 25,7 y el puntaje promedio de Paxos fue 20,8); La figura 14 muestra sus puntajes individuales. un emparejado t-test establece que, con un 95 % de confianza, la distribución real de las puntuaciones de Raft tiene una media de al menos 2,5 puntos mayor que la distribución real de las puntuaciones de Paxos.

También creamos un modelo de regresión lineal que predice los puntajes de las pruebas de un nuevo estudiante en función de tres factores: qué prueba tomaron, su grado de experiencia previa en Paxos y

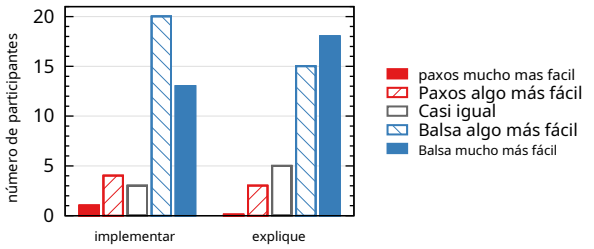

Figura 15:Utilizando una escala de 5 puntos, se preguntó a los participantes (izquierda) qué algoritmo creían que sería más fácil de implementar en un sistema funcional, correcto y eficiente, y (derecha) cuál sería más fácil de explicar a un estudiante de posgrado en informática.

el orden en que aprendieron los algoritmos. El modelo predice que la elección del cuestionario produce una diferencia de 12,5 puntos a favor de Raft. Esto es significativamente más alto que la diferencia observada de 4,9 puntos, porque muchos de los estudiantes reales tenían experiencia previa en Paxos, lo que ayudó considerablemente a Paxos, mientras que ayudó a Raft un poco menos. Curiosamente, el modelo también predice puntuaciones 6,3 puntos más bajas en Raft para las personas que ya han realizado el cuestionario de Paxos; aunque no sabemos por qué, esto parece ser estadísticamente significativo.

También encuestamos a los participantes después de sus cuestionarios para ver qué algoritmo creían que sería más fácil de implementar o explicar; estos resultados se muestran en la Figura 15. Una gran mayoría de los participantes informaron que Raft sería más fácil de implementar y explicar (33 de 41 para cada pregunta). Sin embargo, estos sentimientos autoinformados pueden ser menos confiables que los puntajes de las pruebas de los participantes, y los participantes pueden haber estado sesgados por el conocimiento de nuestra hipótesis de que Raft es más fácil de entender.

Una discusión detallada del estudio de usuarios de Raft está disponible en [31].

9.2 Corrección

Hemos desarrollado una especificación formal y una prueba de seguridad para el mecanismo de consenso descrito en la Sección 5. La especificación formal [31] hace que la información resumida en la Figura 2 sea completamente precisa utilizando el lenguaje de especificación TLA+ [17]. Tiene alrededor de 400 líneas y sirve como tema de la prueba. También es útil por sí solo para cualquiera que implemente Raft. Hemos probado mecánicamente la propiedad de completitud del registro utilizando el sistema de prueba TLA [7]. Sin embargo, esta prueba se basa en invariantes que no se han verificado mecánicamente (por ejemplo, no hemos probado la seguridad de tipo de la especificación). Además, hemos escrito una prueba informal [31] de la propiedad State Machine Safety que es completa (se basa solo en la especificación) y relativa.

Inquietud	Medidas adoptadas para mitigar el sesgo	Materiales para revisión [28, 31]
Igual calidad de lectura	El mismo profesor para ambos. Conferencia de Paxos basada y mejorada a partir de materiales existentes utilizados en varias universidades. La conferencia de Paxos es un 14% más larga.	vídeos
Prueba de igual dificultad	Preguntas agrupadas en dificultad y emparejadas entre exámenes.	cuestionarios
calificación justa	Rúbrica usada. Calificado en orden aleatorio, alternando entre cuestionarios.	rúbrica

Tabla 1:Inquietudes sobre un posible sesgo contra Paxos en el estudio, pasos tomados para contrarrestar cada uno y materiales adicionales disponibles.

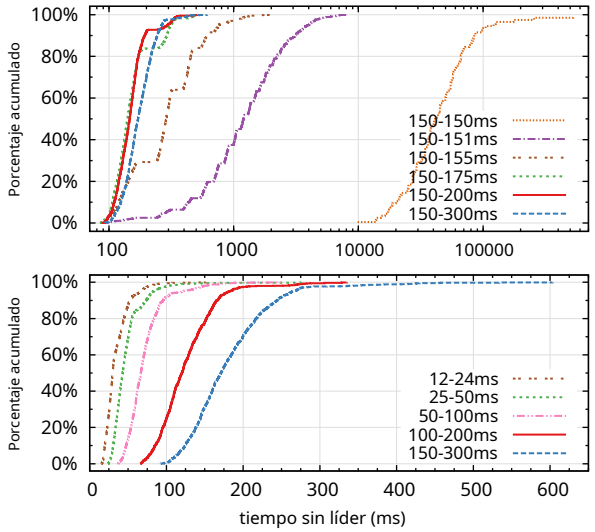

Figura 16:El momento de detectar y reemplazar un líder estrellado. El gráfico superior varía la cantidad de aleatoriedad en los tiempos de espera de las elecciones y el gráfico inferior escala el tiempo de espera mínimo de las elecciones. Cada línea representa 1000 intentos (excepto 100 intentos para "150-150 ms") y corresponde a una elección particular de tiempos de espera de elección; por ejemplo, “150–155ms” significa que los tiempos de espera de las elecciones se eligieron de manera aleatoria y uniforme entre 150ms y 155ms. Las medidas se tomaron en un grupo de cinco servidores con un tiempo de transmisión de aproximadamente 15 ms. Los resultados para un clúster de nueve servidores son similares.

bastante preciso (tiene alrededor de 3500 palabras).

9.3 Rendimiento

El rendimiento de Raft es similar al de otros algoritmos de consenso como Paxos. El caso más importante para el rendimiento es cuando un líder establecido está replicando nuevas entradas de registro. Raft logra esto utilizando la cantidad mínima de mensajes (un solo viaje de ida y vuelta desde el líder hasta la mitad del grupo). También es posible mejorar aún más el rendimiento de Raft. Por ejemplo, admite fácilmente solicitudes de procesamiento por lotes y canalización para un mayor rendimiento y una latencia más baja. Se han propuesto varias optimizaciones en la literatura para otros algoritmos; muchos de estos podrían aplicarse a Raft, pero dejamos esto para trabajos futuros.

Usamos nuestra implementación de Raft para medir el rendimiento del algoritmo de elección de líder de Raft y responder dos preguntas. Primero, ¿converge rápidamente el proceso electoral? En segundo lugar, ¿cuál es el tiempo de inactividad mínimo que se puede lograr después de que el líder falle?

Para medir la elección del líder, colapsamos repetidamente al líder de un grupo de cinco servidores y calculamos el tiempo que tomó detectar el bloqueo y elegir un nuevo líder (consulte la Figura 16). Para generar el peor de los casos, los servidores de cada prueba tenían diferentes longitudes de registro, por lo que algunos candidatos no eran elegibles para convertirse en líderes. Además, para alentar los votos divididos, nuestro script de prueba activó una transmisión sincronizada de RPC de latidos del líder antes de finalizar su proceso (esto se aproxima al comportamiento del líder al replicar una nueva entrada de registro antes del bloqueo).

En g). El líder se estrelló de manera uniforme y aleatoria dentro de su intervalo de latido, que era la mitad del tiempo de espera mínimo de elección para todas las pruebas. Por lo tanto, el tiempo de inactividad más pequeño posible fue aproximadamente la mitad del tiempo de espera mínimo de las elecciones.

El gráfico superior de la Figura 16 muestra que una pequeña cantidad de aleatorización en el tiempo de espera de las elecciones es suficiente para evitar votos divididos en las elecciones. En ausencia de aleatoriedad, la elección del líder tomó constantemente más de 10 segundos en nuestras pruebas debido a muchos votos divididos. Agregar solo 5 ms de aleatoriedad ayuda significativamente, lo que resulta en un tiempo de inactividad promedio de 287 ms. El uso de más aleatoriedad mejora el comportamiento en el peor de los casos: con 50 ms de aleatoriedad, el tiempo de finalización del peor de los casos (más de 1000 intentos) fue de 513 ms.

El gráfico inferior de la Figura 16 muestra que el tiempo de inactividad se puede reducir al reducir el tiempo de espera de la elección. Con un tiempo de espera de elección de 12 a 24 ms, solo se necesitan 35 ms en promedio para elegir un líder (el juicio más largo tomó 152 ms). Sin embargo, reducir los tiempos de espera más allá de este punto viola el requisito de tiempo de Raft: los líderes tienen dificultades para transmitir los latidos del corazón antes de que otros servidores comiencen nuevas elecciones. Esto puede provocar cambios de líder innecesarios y una menor disponibilidad general del sistema. Recomendamos usar un tiempo de espera de elección conservador, como 150–300ms; es poco probable que dichos tiempos de espera provoquen cambios de líder innecesarios y aun así proporcionarán una buena disponibilidad.

10 Trabajo relacionado

Ha habido numerosas publicaciones relacionadas con los algoritmos de consenso, muchas de las cuales pertenecen a una de las siguientes categorías:

- La descripción original de Lamport de Paxos [15] e intenta explicarla más claramente [16, 20, 21].
- Elaboraciones de Paxos, que completan los detalles que faltan y modifican el algoritmo para proporcionar una mejor base para la implementación [26, 39, 13].
- Sistemas que implementan algoritmos de consenso, como Chubby [2, 4], ZooKeeper [11, 12] y Spanner [6]. Los algoritmos de Chubby y Spanner no se han publicado en detalle, aunque ambos afirman estar basados en Paxos. El algoritmo de ZooKeeper se ha publicado con más detalle, pero es bastante diferente al de Paxos.
- Optimizaciones de rendimiento que se pueden aplicar a Paxos [18, 19, 3, 25, 1, 27].
- Viewstamped Replication (VR) de Oki y Liskov, un enfoque alternativo al consenso desarrollado casi al mismo tiempo que Paxos. La descripción original [29] se entrelazó con un protocolo para transacciones distribuidas, pero el protocolo central de consenso se separó en una actualización reciente [22]. VR utiliza un enfoque basado en líderes con muchas similitudes con Raft.

La mayor diferencia entre Raft y Paxos es el fuerte liderazgo de Raft: Raft usa la elección del líder como una parte esencial del protocolo de consenso, y se concentra en

Trate tanta funcionalidad como sea posible en el líder. Este enfoque da como resultado un algoritmo más simple que es más fácil de entender. Por ejemplo, en Paxos, la elección del líder es ortogonal al protocolo de consenso básico: solo sirve como una optimización del rendimiento y no se requiere para lograr el consenso. Sin embargo, esto da como resultado un mecanismo adicional: Paxos incluye un protocolo de dos fases para el consenso básico y un mecanismo separado para la elección del líder. Por el contrario, Raft incorpora la elección del líder directamente en el algoritmo de consenso y lo utiliza como la primera de las dos fases del consenso. Esto da como resultado menos mecanismo que en Paxos.

Al igual que Raft, VR y ZooKeeper se basan en líderes y, por lo tanto, comparten muchas de las ventajas de Raft sobre Paxos. Sin embargo, Raft tiene menos mecanismos que VR o ZooKeeper porque minimiza la funcionalidad en los no líderes. Por ejemplo, las entradas de registro en Raft fluyen en una sola dirección: hacia afuera desde el líder en AppendEntries RPC. En VR, las entradas de registro fluyen en ambas direcciones (los líderes pueden recibir entradas de registro durante el proceso de elección); esto da como resultado un mecanismo y una complejidad adicionales. La descripción publicada de ZooKeeper también transfiere entradas de registro hacia y desde el líder, pero la implementación aparentemente se parece más a Raft [35].

Raft tiene menos tipos de mensajes que cualquier otro algoritmo para la replicación de registros basada en consenso que conozcamos. Por ejemplo, contamos los tipos de mensajes que usan VR y ZooKeeper para el consenso básico y los cambios de membresía (excluyendo la compactación de registros y la interacción del cliente, ya que estos son casi independientes de los algoritmos). VR y ZooKeeper definen cada uno 10 tipos de mensajes diferentes, mientras que Raft tiene solo 4 tipos de mensajes (dos solicitudes RPC y sus respuestas). Los mensajes de Raft son un poco más densos que los de otros algoritmos, pero colectivamente son más simples. Además, VR y ZooKeeper se describen en términos de transmitir registros completos durante los cambios de líder; se requerirán tipos de mensajes adicionales para optimizar estos mecanismos para que sean prácticos.

El fuerte enfoque de liderazgo de Raft simplifica el algoritmo, pero excluye algunas optimizaciones de rendimiento. Por ejemplo, Egalitarian Paxos (EPaxos) puede lograr un mayor rendimiento en algunas condiciones con un enfoque sin líderes [27]. EPaxos explota la conmutatividad en los comandos de la máquina de estado. Cualquier servidor puede enviar un comando con solo una ronda de comunicación, siempre y cuando otros comandos que se propongan simultáneamente con él. Sin embargo, si los comandos que se proponen simultáneamente no se conmutan entre sí, EPaxos requiere una ronda adicional de comunicación. Debido a que cualquier servidor puede enviar comandos, EPaxos equilibra bien la carga entre servidores y puede lograr una latencia más baja que Raft en la configuración de WAN. Sin embargo, agrega una complejidad significativa a Paxos.

En otros trabajos, se propusieron o implementaron varios enfoques diferentes para los cambios en la membresía del clúster, incluida la propuesta original de Lamport [15], VR [22] y SMART [24]. Elegimos el enfoque de consenso conjunto para Raft porque aprovecha el resto del protocolo de consenso, por lo que se requiere muy poco mecanismo adicional para los cambios de membresía. de LamportαEl enfoque basado en el enfoque no era una opción para Raft porque supone que se puede llegar a un consenso sin un líder. En comparación con VR y SMART, el algoritmo de reconfiguración de Raft tiene la ventaja de que pueden ocurrir cambios de membresía sin limitar el procesamiento de solicitudes normales; por el contrario, VR detiene todo el procesamiento normal durante los cambios de configuración, y SMART impone unα-como límite en el número de solicitudes pendientes. El enfoque de Raft también agrega menos mecanismo que VR o SMART.

11 Conclusión

Los algoritmos a menudo se diseñan con la corrección, la eficiencia y/o la concisión como objetivos principales. Aunque todos estos son objetivos valiosos, creemos que la comprensión es igual de importante. Ninguno de los otros objetivos se puede lograr hasta que los desarrolladores conviertan el algoritmo en una implementación práctica, que inevitablemente se desviará y ampliará la forma publicada. A menos que los desarrolladores tengan una comprensión profunda del algoritmo y puedan crear intuiciones al respecto, será difícil para ellos conservar sus propiedades deseables en su implementación.

En este documento, abordamos el tema del consenso distribuido, donde un algoritmo ampliamente aceptado pero impenetrable, Paxos, ha desafiado a los estudiantes y desarrolladores durante muchos años. Desarrollamos un nuevo algoritmo, Raft, que hemos demostrado que es más comprensible que Paxos. También creemos que Raft proporciona una mejor base para la creación de sistemas. El uso de la comprensibilidad como objetivo de diseño principal cambió la forma en que abordamos el diseño de Raft; a medida que avanzaba el diseño, nos encontramos reutilizando algunas técnicas repetidamente, como descomponer el problema y simplificar el espacio de estado. Estas técnicas no solo mejoraron la comprensibilidad de Raft, sino que también hicieron que fuera más fácil convencernos de su corrección.

12 Agradecimientos

El estudio de usuarios no hubiera sido posible sin el apoyo de Ali Ghodsi, David Mazières y los estudiantes de CS 294-91 en Berkeley y CS 240 en Stanford. Scott Klemmer nos ayudó a diseñar el estudio de usuarios y Nelson Ray nos aconsejó sobre el análisis estadístico. Las diapositivas de Paxos para el estudio del usuario se tomaron prestadas en gran medida de una plataforma de diapositivas creada originalmente por Lorenzo Alvisi. Un agradecimiento especial a David Mazières y Ezra Hoch por encontrar errores sutiles en Raft. Muchas personas brindaron comentarios útiles sobre el documento y los materiales de estudio de los usuarios, incluidos Ed Bugnion, Michael Chan, Hugues Evrard,

Daniel Giffin, Arjun Gopalan, Jon Howell, Vimalkumar Jeyakumar, Ankita Kejriwal, Aleksandar Kracun, Amit Levy, Joel Martin, Satoshi Matsushita, Oleg Pesok, David Ramos, Robbert van Renesse, Mendel Rosenblum, Nicolas Schiper, Deian Stefan, Andrew Stone, Ryan Stutsman, David Terei, Stephen Yang, Matei Zaharia, 24 revisores anónimos de la conferencia (con duplicados), y especialmente nuestro pastor Eddie Kohler. Werner Vogels tuiteó un enlace a un borrador anterior, lo que le dio a Raft una exposición significativa. Este trabajo fue apoyado por el Gigascale Systems Research Center y el Multiscale Systems Center, dos de los seis centros de investigación financiados bajo el Focus Center Research Program, un programa de Semiconductor Research Corporation, por STARnet, un programa de Semiconductor Research Corporation patrocinado por MARCO y DARPA, por la Fundación Nacional de Ciencias bajo la Subvención No. 0963859, y por subvenciones de Facebook, Google, Mellanox, NEC, NetApp, SAP y Samsung. Diego Ongaro cuenta con el apoyo de The Junglee Corporation Stanford Graduate Fellowship.

Referencias

[1] segundoOLOSKY, WJ, BRADSHAW, D, HAGENS, RB, Kusuarios, NP, YLyo, P. Paxos replicaron máquinas de estado como base de un almacén de datos de alto rendimiento. En *proc. NSDI'11, Conferencia USENIX sobre Diseño e Implementación de Sistemas en Red*(2011), USENIX, págs. 141–154.

[2] segundoURUJAS, M. El servicio de bloqueo Chubby para sistemas distribuidos débilmente acoplados. En*proc. OSDI'06, Simposio sobre Diseño e Implementación de Sistemas Operativos* (2006), USENIX, págs. 335–350.

[3]CAMARGOS, LJ, SCHMIDT, RM,YPAGSEDÓN, f Paxos multicoordinados. En*proc. PODC'07, Simposio ACM sobre Principios de Computación Distribuida*(2007), ACM, págs. 316–317.

[4]CMANDRA, TD, GRIESEMER, r,YREDSTONE, j Paxos hecho en vivo: una perspectiva de ingeniería. En*proc. PODC'07, Simposio ACM sobre Principios de Computación Distribuida*(2007), ACM, págs. 398–407.

[5]CCOLGAR, F, DEAN, J, GHEMAWAT, S, HSIEH, WC, WALLACH, AD, BURUJAS, M, CMANDRA, t, FIKES, A.,YGRAMOGOMA, RE Bigtable: un sistema de almacenamiento distribuido para datos estructurados. En*proc. OSDI'06, Simposio USENIX sobre Diseño e Implementación de Sistemas Operativos*(2006), USENIX, págs. 205–218.

[6]CORBETT, JC, DEAN, J, E.PSTEÍNA, M., F.IKES, A., FROST, C, F.URMAN, JJ, GHEMAWAT, S, GUBAREV, A, HEISER, C, HOCHSCHILD, P, HSIEH, W, KUN-GRACIAS, S, KOGAN, E, Lyo, H, LLOYD, SOYELNIK, S, MWAURA , D, NAGLE, D, QUINLAN, S, ROA, r, ROLIG, L, SAITO, Y, SZYMANIAK, M, TAYLOR, C., WESP, r,YWOODFORD, D. Spanner: base de datos distribuida globalmente de Google. En*proc. OSDI'12, Conferencia USENIX sobre Diseño e Implementación de Sistemas Operativos*(2012), USENIX, págs. 251–264.

[7]COUSINEAU, D, DOLIGUEZ, D, LAMPORTAR, L, MZER, s, rTICKETTS, D.,YVANZETTO, H.TLA+pruebas En*proc. FM'12, Simposio sobre Métodos Formales*(2012), D. Giannakopoulou y D. Méry, Eds., vol. 7436 de*Apuntes de clase en informática*, Springer, págs. 147–154.

[8] solHEMAWAT, S, GOBIOFF, H,YLEUNG, S T. El sistema de archivos de Google. En*proc. SOSP'03, Simposio ACM sobre Principios de Sistemas Operativos*(2003), ACM, págs. 29–43.

[9] solRAYO, C.,YCHERITON, D. Arrendamientos: un mecanismo tolerante a fallas eficiente para la consistencia del caché de archivos distribuidos. En*Actas del 12º Simposio ACM sobre principios de sistemas operativos*(1989), págs. 202–210.

[10] HERLIHY, diputado,YWEN G, JM Linealizabilidad: una condición de corrección para objetos concurrentes.*Transacciones ACM en lenguajes y sistemas de programación 12*(julio de 1990), 463–492.

[11] HUNT, PAQUETEONAR, M, JUNQUEIRAFP,YRDEE, B. ZooKeeper: coordinación sin esperas para sistemas a escala de Internet. En*Proc ATC'10, Conferencia Técnica Anual USENIX*(2010), USENIX, págs. 145–158.

[12] JUNQUEIRA, FP, RDEE, ANTES DE CRISTO,YSERAFINI, M. Zab: Transmisión de alto rendimiento para sistemas primarios de respaldo. En*proc. DSN'11, Conferencia Internacional IEEE/IFIP. en sistemas y redes confiables*(2011), IEEE Computer Society, págs. 245–256.

[13] KIRSCH, J,YAMIR, Y. Paxos para constructores de sistemas. tecnología Rep. CNDS-2008-2, Universidad Johns Hopkins, 2008.

[14] LAMPORTAR, L. Tiempo, relojes y ordenamiento de eventos en un sistema distribuido.*Comunicaciones de la ACM 21*, 7 (julio de 1978), 558–565.

[15] LAMPORTAR, L. El parlamento a tiempo parcial.*Transacciones de ACM en sistemas informáticos 16*, 2 (mayo de 1998), 133–169.

[16] LAMPORTAR, L. Paxos simplificado.*ACM SIGACT Noticias 32*, 4 (diciembre de 2001), 18–25.

[17] LAMPORTAR, l*Especificación de sistemas, el lenguaje TLA+ y herramientas para ingenieros de hardware y software*. Addison-Wesley, 2002.

[18] LAMPORTAR, L. Consenso generalizado y Paxos. tecnología Rep. MSR-TR-2005-33, Microsoft Research, 2005.

[19] LAMPORTAR, L. Paxos rápidos.*Computación Distribuida 19*, 2 (2006), 79–103.

[20] LAMSON, BW Cómo construir un sistema de alta disponibilidad usando consenso. En*Algoritmos Distribuidos*, O. Baboaglu y K. Marzullo, Eds. Springer-Verlag, 1996, págs. 1–17.

[21] LAMSON, BW El ABCD de Paxos. En*proc. PODC'01, Simposio ACM sobre Principios de Computación Distribuida*(2001), ACM, págs. 13–13.

[22] LISKOV, B.,YCBÚHO, J. Revisión de la replicación con sello de vista. tecnología Rep. MIT-CSAIL-TR-2012-021, MIT, julio de 2012.

[23] Código fuente de LogCabin. cabaña de troncos / cabaña de troncos. http://github.com/

[24] LORQUESTA, JR, ADYA, A, BOLOSKY, WJ, CHAIKEN, R., D.OUCEUR, JR,YHO BIEN, j El inteligente manera de migrar servicios con estado replicados. En *proc. 1st ACM SIGOPS/EuroSys Conferencia Europea de Sistemas Informáticos* (2006), ACM, págs. 103–115.

[25] MOA, Y, JUNQUEIRAFP,YMETROARZULLO, k Mencius: construcción de máquinas de estado replicadas eficientes para WAN. En *proc. OSDI'08, Conferencia USENIX sobre Diseño e Implementación de Sistemas Operativos* (2008), USENIX, págs. 369–384.

[26] MAZIÈRES, D. Paxos hecho práctico. http://www.scs.stanford.edu/ðm/home/papers/paxos.pdf,enero de 2007.

[27] MORARU, I ANDERSEN, director general,YKAMINSKY, m Hay más consenso en los parlamentos igualitarios. En *proc. SOSP'13, Simposio ACM sobre Principios de Sistemas Operativos* (2013), ACM.

[28] Estudio de usuarios de balsas. http://ramcloud.stanford.edu/õngaro/userstudy/.

[29] OKI, BM,YLISKOV, BH Viewstamped replication: un nuevo método de copia principal para admitir sistemas distribuidos de alta disponibilidad. En *proc. PODC'88, Simposio ACM sobre Principios de Computación Distribuida* (1988), ACM, págs. 8–17.

[30] ENNIE, P, CHENG, P.EJAWLICK, D.,YENNIE, E. El árbol de combinación con estructura de registro (árbol LSM). *Acta Informática 33*, 4 (1996), 351–385.

[31] Ohngaro, D. *Consenso: uniendo la teoría y la práctica*. Tesis doctoral, Universidad de Stanford, 2014 (trabajo en progreso). http://ramcloud.stanford.edu/õngaro/thesis.pdf.

[32] Ongaro, D.,YOUSTERHOUT, j en busca de un algoritmo de consenso comprensible. En *proceso ATC'14, Conferencia técnica anual de USENIX* (2014), USENIX.

[33] OHUSTERHOUT, J., A.GRAWAL, EDUCACIÓN FÍSICARICKSON, D, KOZYRAKIS, C, LEVERICO, J, MAZIÈRES, D, MYOTRA, S, NARAYANAN, A, Ongaro, D, PARULKAR, g, rOSENBLUM, SRESUMBLE, SMSTRATMANN, MI.,YSTUTSMAN, R. El caso de RAMCloud. *Comunicaciones del ACM 54* (julio de 2011), 121–130.

[34] Sitio web del algoritmo de consenso Raft. http://raftconsensus.github.io.

[35] RDEE, B. Comunicaciones personales, 17 de mayo de 2013.

[36] ROSENBLUM, m,YOUSTERHOUT, JK Diseño e implementación de un sistema de archivos con estructura de registro. *ACM Trans. computar sist. 10* (febrero de 1992), 26–52.

[37] SCHNEIDER, FB Implementación de servicios tolerantes a fallas utilizando el enfoque de máquina de estado: un tutorial. *Encuestas informáticas de ACM 22*, 4 (diciembre de 1990), 299–319.

[38] SHVACHKO, k, kUANG, H, RADIA, S.,Y CHANSLER, R. El sistema de archivos distribuido Hadoop. En *proc. MSST'10, Simposio sobre Sistemas y Tecnologías de Almacenamiento Masivo* (2010), IEEE Computer Society, págs. 1–10.

[39]CAMIONETARENESSE, R. Paxos hizo medianamente complejo. tecnología representante, Universidad de Cornell, 2012.