Universidad Nacional de Río Cuarto

Facultad de Ciencias Exactas Fisico-Quimicas y Naturales

Departamento de Computación

Telecomunicaciones y Sistemas Distribuidos

Proyecto final

2018

- **Responsable:**
  - Arroyo Marcelo

- **Integrantes:**
  - Bentolila Fernando, DNI: 38019972
  - Zabala Joaquin, DNI: 38022167

Índice

1. Descripción del problema

Se plantea el desarrollo de un sistema de recursos distribuidos tolerante a fallas (hasta N/2 -1 nodos), con N procesos.

El sistema debe mantener un conjunto de pares (id, v) (diccionario) replicado y responder a mensajes de clientes del tipo get(id) y set(id, v).

Un cliente se puede comunicar con cualquier nodo de la red para realizar un requerimiento (set/get).

2. Selección del Algoritmo

Para la resolución e implementación de este problema se tomaron en cuenta varias alternativas, y diferentes tipos de protocolos, pero finalmente se optó por la implementación de un sistema distribuido cliente-servidor (dado que existen dos roles bien definidos, clientes y servidores) con una **replicación total de los datos** en cada uno de los nodos del sistema. Con este objetivo propuesto, recurrimos al uso de un **algoritmo de consenso**, más concretamente el algoritmo de Raft, el cual nos permite gestionar esta replicación de los datos y mantener la consistencia en los mismos. Raft se apoya, en un protocolo de **elección de un líder** único, encargado de la correcta administración de los recursos (Más adelante se profundizará en una explicación detallada de este algoritmo de Raft).

2.1. Características

A continuación explicaremos las características generales que presenta nuestro sistema distribuido, así como sus pros, contras y los motivos por el cual se decidió realizarlo de este modo. Así mismo, daremos otras posibles soluciones a este problema mediante otros enfoques existentes.

2.1.1. Replicación de datos

Este algoritmo está basado en la replicación de datos, el cual consiste en mantener copias de la información en las múltiples máquinas distribuidas por el sistema. En este caso, el tipo de replicación es total y se mantienen copias de todos los objetos/recursos, en lugar de solo mantener ciertos fragmentos de los mismos.

Este mecanismo favorece al sistema en algunos aspectos, tales como:

- Mayor disponibilidad. El recurso puede encontrarse en varios sitios.
- Confiabilidad. Permite seguir respondiendo a pesar del fallo de un sitio.
- En caso de que el acceso a un recurso sea de solo lectura, varios sitios pueden procesar, en paralelo, las lecturas que impliquen al recurso.
- Mejor protección contra la corrupción y pérdida de datos, al tener gran variedad de copias.

Pero por el contrario, este acarrea algunos problemas consigo:

- Problema de consistencia de datos
    - Se debe asegurar la consistencia mutua, en caso contrario, pueden producirse cómputos erróneos. Para ello, se debe propagar la actualización a todos (o la mayoría) de los sitios que contienen réplicas (sobrecarga durante la actualización).
- Mayor tiempo de procesamiento. El intercambio de mensajes hacia las réplicas y los cálculos adicionales suponen una forma de tiempo extra que debe ser considerada.
- Mantener múltiples copias consistentes resulta a su vez un serio problema de escalabilidad y más en un contexto de consistencia estricta.
- Costos adicionales:
    - Costos del desarrollo de software: es necesario un software para la administración de la replicación.
    - Costos de almacenamiento de las réplicas.
    - Costos de procesamiento

Como dato adicional, se puede apreciar una comparativa entre las características de las distintas estrategias de replicación:

	Replicación total	Replicación parcial	No replicada
Paralelismo	Muy alto	Alto	Nulo
Disponibilidad	Muy alta	Alta	Baja
Costo	Elevado	Moderado	Bajo
Riesgo de pérdida de datos	Muy bajo	Medio	Alta
Realidad	Aplicación posible	Realista	Aplicación posible

Figura 1: Características según la estrategia de replicación.

Algunos usos prácticos a la replicación de datos son:

- Servidores web y servidores de base de datos.
- DNS: copias de los mapeos URL-IP.
- Google: Google Data Centers.
- Replicación del reloj del sistema.

2.1.2. Consenso

En general, los sistemas distribuidos son propensos a fallos ya sea por caídas o fallos intencionados. Para conseguir que los sistemas sean más fiables, se intenta que sean lo más tolerantes frente a dichos fallos. Una de las estrategias más habituales para conseguirlo es mediante la réplica de información. El sistema de replicación es fiable, pues la información reside en varios nodos, pero es un reto mantener la consistencia entre los nodos que comparten

información, a su vez que se tiene un entorno en el que pueden haber fallos. De esto se encargan los algoritmos o protocolos de consenso.

Los algoritmos o protocolos de consenso son mecanismos que permiten a los usuarios o máquinas coordinarse en un entorno distribuido, los cuales deben garantizar que todos los agentes del sistema puedan ponerse de acuerdo respecto a una fuente única de verdad, incluso en el caso de que algunos de ellos fallen. Este tipo de algoritmo es fuertemente utilizado para el funcionamiento de las criptomonedas y su tecnología blockchain, entre ellos, los más destacados son Proof of Work y Proof of Stake.

Los algoritmos de consenso (en nuestro caso Raft) suelen surgir en el contexto de máquinas de estado replicadas. En este enfoque, las máquinas de estado en una colección de servidores calculan copias idénticas del mismo estado y pueden seguir funcionando incluso si algunos de los servidores están caídos.

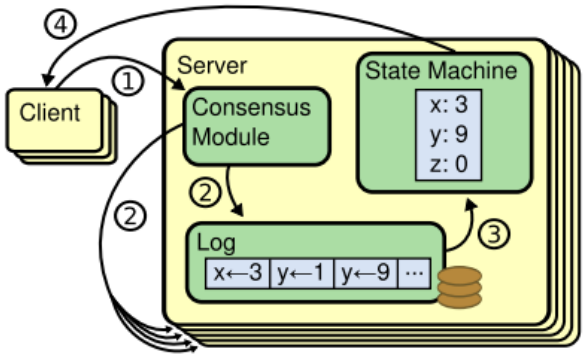

Figura 2: *Arquitectura de máquina de estado replicada. El algoritmo de consenso administra un registro replicado que contiene los comandos de la máquina de estado de los clientes. Las máquinas de estado procesan secuencias idénticas de comandos de los registros, por lo que producen los mismos resultados.*

2.1.3. Eleccion de un lider

Al utilizar la estrategia de seleccionar un líder para la administración de todas las solicitudes del sistema y la replicación de la información, obtenemos algunos beneficios:

- Un líder único hace que sea más fácil para los humanos pensar en los sistemas. Coloca toda la simultaneidad del sistema en un solo lugar, reduce los modos de errores parciales y agrega un solo lugar para buscar los registros y las métricas.
- Los líderes únicos pueden ofrecer a los clientes consistencia con facilidad, ya que pueden ver y controlar todos los cambios realizados en el estado del sistema.
- La escritura de software para un solo líder puede resultar más sencilla.

Desventajas a considerar:

- Se obtiene un punto de error único. Si el sistema no detecta a un mal líder o no lo corrige, es posible que todo el sistema no esté disponible.
- Un líder único es un punto de confianza único. Si un líder está realizando un mal trabajo sin que nadie lo compruebe, puede provocar problemas en todo el sistema con rapidez. Un mal líder tiene un radio de acción alto.
- Implica un punto de escalado único, ya sea en el tamaño de los datos o en la tasa de solicitudes.

Para resumir, la elección de líder que centraliza ciertas tareas es una herramienta muy eficaz que mejora la eficiencia, reduce la coordinación, simplifica las arquitecturas y reduce las operaciones.

Por otra parte, la elección de líder puede introducir nuevos modos de error y cuellos de botella de escalado. Además, esta herramienta puede complicar más la tarea de evaluación de la exactitud de un sistema.

2.2. Otros posibles enfoques

Otra opción para afrontar este problema de desarrollo de un sistema distribuido puede encontrarse en los protocolos y algoritmos para la implementación de tablas de hash distribuidas. A grandes rasgos, estas son un tipo de tablas de hash, en la que los datos se almacenan de forma distribuida en una serie de nodos y proveen un servicio eficiente de búsqueda que permite encontrar el valor asociado a una clave. Para esto último usan un sistema de enrutado que permite encontrar de forma eficiente el nodo en el cual está almacenada la información que se necesita (replicación parcial).

Este tipo de redes son buenas para:

- Muy escalables pues automáticamente distribuyen la carga a los nuevos nodos que se unen a la red.
- Robustez contra fallos de nodos, los datos automáticamente migran fuera de los nodos que fallan.
- Son auto-organizadas, no necesitan de un servidor central. La parte centralizada es únicamente para localizar los nodos que siguen dependiendo de los DNS.

Tienen falencias en:

- Búsquedas; consecuencia del algoritmo hash pues "abc" y "abcd" corresponden a nodos totalmente diferentes (aunque el valor buscado es muy similar).
- Problemas de seguridad; es complicado verificar la integridad de los datos almacenados.

2.2.1. Algoritmo de Chord

Un ejemplo de algoritmo/protocolo para la implementación y manejo de estas tablas de hash distribuidas es el caso del algoritmo de Chord. Este aborda el problema de localizar de manera eficiente el nodo que almacena un elemento de datos en particular.

A modo resumen y sin profundizar demasiado, podemos decir que el protocolo permite tener un conjunto de nodos identificados con un conjunto de bits, donde cada nodo, además, puede tener asociado a una o varias claves con id menor o igual que su identificador, a excepción de el nodo con menor id del conjunto, el cual puede tener claves asociadas con id mayor que él. Esto le brinda a la red una topología de anillo, permitiendo una fácil implementación, y evita hacer cambios bruscos a la red en caso de crecimiento.

El procedimiento de búsqueda consiste en que, cuando un nodo solicita una clave *j*, examina su propia tabla de rutas; si encuentra el nodo responsable de *j*, envía la petición directamente al nodo afectado. En caso contrario, pregunta al nodo cuya id sea más cercana (menor) a *j*, que devolverá la id del nodo más cercano a *j* (menor) que encuentre en su tabla de rutas. De esta manera, el nodo remitente obtiene en cada nueva iteración un nodo más cercano a *k*, hasta llegar a *k* o a su sucesor.

La implementación de este algoritmo, garantiza un sistema sencillo, eficiente, altamente escalable y está diseñado para funcionar en redes descentralizadas Peer-to-peer (sin nodos privilegiados).

3. Algoritmo de Raft

A grandes rasgos Raft es un algoritmo para administrar un registro replicado. Implementa el consenso eligiendo primero a un líder distinguido y luego otorgándole a éste la responsabilidad total de administrar el registro replicado. El líder acepta registrar entradas de clientes, las replica en otros servidores, e indica a los servidores cuando es seguro aplicar estas entradas a sus máquinas de estado.

Raft descompone el problema del consenso en tres sub-problemas:

- **Elección de líder**: se debe elegir un nuevo líder cuando falla un líder existente.
- **Replicación de registros**: el líder debe aceptar las entradas de registros de los clientes y replicarlas en todo el clúster, lo que obliga a los otros registros a coincidir con los suyos.
- **Seguridad:** si algún servidor ha aplicado una entrada de registro en particular a su máquina de estado, ningún otro servidor puede aplicar un comando diferente para el mismo índice de registro.

En un momento dado cada servidor se encuentra en uno de estos tres estados: líder, seguidor o candidato. En un funcionamiento normal hay exactamente un líder y todos los demás servidores son seguidores. Los seguidores son pasivos: no emiten solicitudes, 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 dirige al líder). El tercer estado, candidato, se utiliza para elegir un nuevo líder.

Raft divide el tiempo en términos de longitud arbitraria. Estos se enumeran con números enteros consecutivos. Cada término comienza con una con una elección, en la que uno o más candidatos intentan convertirse en líderes. Si un candidato gana las elecciones, se desempeñará como líder durante el resto de ese término. En algunas situaciones, una elección resultará en una votación dividida. En este caso, el término terminará sin líder; un nuevo término (con una nueva elección) comenzará en breve. Raft asegura que haya como máximo un líder en un periodo determinado.

Los términos actúan como un reloj lógico, y permiten a los servidores detectar información obsoleta, como líderes obsoletos. Cada servidor almacena un número de término actual, que aumenta monótonamente con el tiempo. Los términos actuales se intercambian siempre que los servidores se comunican; si el término actual de un servidor es más pequeño que el del otro, actualiza su término actual al valor más grande. Si un candidato o líder descubre que su mandato está desactualizado, inmediatamente vuelve al estado de seguidor. Si un servidor recibe una solicitud con un número de término obsoleto, rechaza la solicitud.

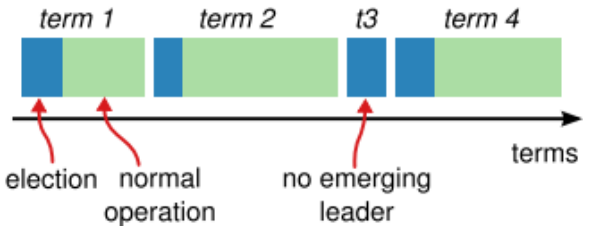

Figura 3: *El tiempo se divide en términos y cada término comienza con una elección. Después de una elección exitosa, un solo líder gestiona el cluster hasta el final del plazo. Algunas elecciones pueden fallar, en cuyo caso el plazo termina sin elegir un líder.*

Los servidores de Raft se comunican mediante llamadas a procedimientos remotos (RPC), y el algoritmo de consenso básico requiere sólo dos tipos de RPC. Los RPC de "***RequestVote"*** que son iniciados por los candidatos durante las elecciones, y los RPC d***e "AppendEntries"*** que son iniciados por los líderes para replicar entradas del registro y proporcionar una forma de heartbeat.

Election Safety: como máximo se puede elegir un líder en un período determinado (puede no haber).
Leader Append-Only: un líder nunca sobrescribe ni elimina entradas en su registro; solo agrega nuevas entradas.
Log Matching: 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.
Leader Completeness: si se confirma una entrada de registro en un período determinado, esta entrada estará presente en los registros de los líderes para todos los términos con números más altos.
State Machine Safety: si un servidor ha aplicado una entrada de registro en un índice dado de su máquina de estado, ningún otro servidor aplicará una entrada de registro diferente para el mismo índice.

Raft garantiza que cada una de estas propiedades sea cierta en todo momento.

3.1. Elección de líder

Raft usa un mecanismo de heartbeat para activar la elección de líder. Cuando los servidores se inician, comienzan como seguidores. Un servidor permanece en estado de seguidor siempre que reciba RPC de un líder o candidato. Los líderes envían periódicamente heartbeats (RPC de AppendEntries sin entradas) a todos los seguidores para mantener su autoridad. Si un seguidor no recibe comunicación durante un periodo de tiempo llamado "***election timeout***", entonces asume que no hay un líder viable y comienza una elección para elegir un nuevo líder.

Para comenzar una elección, un seguidor incrementa su término, y pasa al estado de candidato. Luego vota por sí mismo y emite RPC *RequestVote* en paralelo a cada uno de los otros servidores del cluster. Un candidato, continúa en este estado hasta que suceda una de estas tres cosas: (a) gana la elección, (b) otro servidor se establece como líder, o (c) pasa un periodo de tiempo sin ganador.

Un candidato gana una elección si recibe votos de la mayoría de los servidores del cluster para el mismo término. Cada servidor votará como máximo a un candidato en un plazo determinado (término), por orden de llegada. Solo un candidato puede llegar a ganar una elección para un término en particular. Una vez que un candidato gana una elección, se convierte en líder. Luego envía mensajes de heartbeat a todos los otros servidores para establecer su autoridad y evitar nuevas elecciones.

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

El tercer resultado posible es que un candidato ni gane ni pierda las elecciones: 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 terminará el tiempo de espera y comenzará una nueva elección incrementando su término e iniciando otra ronda de RPC de *RequestVote*.

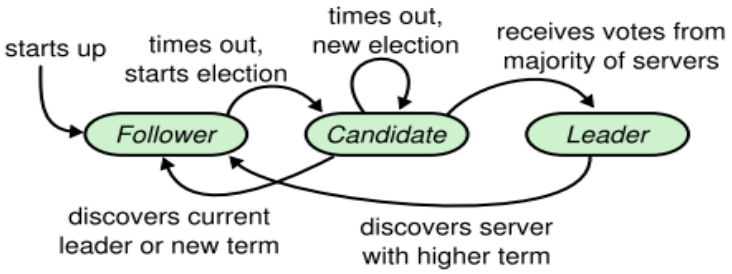

Figura 4: *Estados del servidor. Los seguidores solo responden a las solicitudes de otros servidores. Si un seguidor no recibe comunicación, se convierte en candidato e inicia una elección. Un candidato que recibe votos de la mayoría del grupo, se convierte en el nuevo líder. Los líderes suelen operar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

Raft utiliza tiempos de espera de elección aleatorios (al azar de un intervalo fijo) para garantizar que los votos divididos sean raros y que se resuelvan rápidamente. Esto distribuye los servidores de modo que, en la mayoría de los casos, solo se agote el tiempo de espera de un servidor; gane las elecciones y envié latidos antes de que se agote el tiempo de espera de cualquier otro servidor. Este es el mecanismo que se utiliza para gestionar los votos divididos. Cada candidato reinicia su tiempo de espera de elección aleatoria al comienzo de una elección, y espera antes de comenzar la siguiente elección; esto reduce la probabilidad de otra votación dividida en la nueva elección.

3.2. Replicación de registros

Una vez 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 *AppendEntries* RPC en paralelo a cada uno de los servidores para replicar la entrada. Cuando la entrada se ha replicado de forma segura, el líder aplica la entrada a su máquina de estado y devuelve el resultado de esa ejecución al cliente. Si los seguidores se bloquean, corren lentamente, o si se pierden paquetes de red, el líder vuelve a intentar *AppendEntries* RPC de forma indefinida (incluso después de haber respondido al cliente) hasta que todos los seguidores finalmente almacenen todas las entradas del registro.

Cada entrada de registro almacena un comando de máquina de estado, junto con el número de término de cuando el líder recibió la entrada. Los números de términos en las entradas de registros son utilizados para detectar inconsistencias entre los registros y para asegurar algunas de las propiedades del algoritmo. Cada entrada del registro también tiene un índice entero que identifica su posición en el registro.

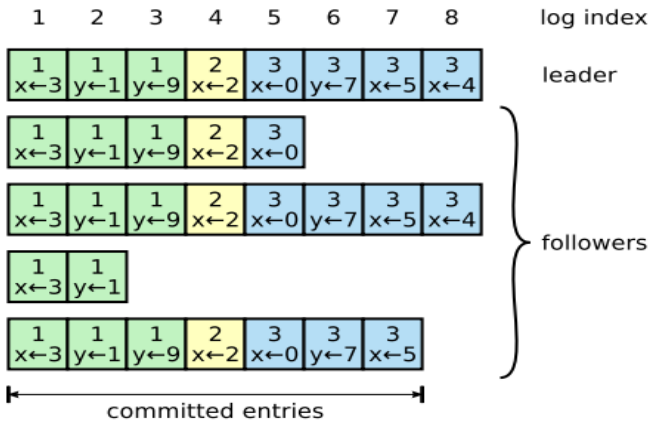

Figura 5: *Los registros se componen de entradas, que están numeradas secuencialmente. Cada entrada contiene el término en el que fue creado (el número en cada cuadro) y un comando para la máquina de estado. Una entrada se considera comprometida si es seguro que esa entrada se aplique a las máquinas de estado.*

El líder decide cuándo es seguro aplicar una entrada de registro a las máquinas de estado; tal entrada se llama comprometida. Raft garantiza que las entradas comprometidas son duraderas y eventualmente serán ejecutadas por todas las máquinas de estado disponibles. Una entrada de registro es comprometida (confirmada), una vez que el líder que creó la entrada la ha replicado en la mayoría de los servidores. Esto también confirma todas las entradas anteriores en el registro del líder, incluidas las entradas creadas por líderes anteriores. El líder realiza un seguimiento del índice más alto que sabe que se ha comprometido e incluye ese índice en futuras RPC de ApendEntries (incluidos los heartbeats) para que otros servidores eventualmente lo averigüen. 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).

Raft tiene mecanismos de registros para mantener un alto nivel de coherencia entre los registros de 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 (Log Matching Property):

- Si dos entradas en registros diferentes tienen el mismo índice y terminó, entonces almacenan el mismo comando.
- Si dos entradas en registros diferentes tienen el mismo índice y terminó, 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 determinado, 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 terminó, entonces rechaza las nuevas entradas. Como resultado, siempre que *AppendEntries* regresa con éxito, el líder sabe que el registro del seguidor es idéntico a su propio registro a través de las nuevas entradas.

En Raft, el líder maneja las inconsistencias obligando a los registros de los seguidores a duplicar los suyos. Esto significa que las entradas conflictivas en los registros de los seguidores se sobrescribirán con las entradas del registro del líder.

Para que el registro de un seguidor sea consistente con el suyo, el líder debe encontrar la última entrada del registro donde coincidan los dos registros, 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 ocurren en respuesta a la comprobación de consistencia realizada por *AppendEntries* RPC. El líder mantiene un "***NextIndex***" 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 *NexIndex* en el índice justo después del último en su registro. Si el registro de un seguidor no es consistente con el del líder. La comprobación de consistencia de *AppendEntries* fallará en el siguiente RPC de *AppendEntries*. Después de un rechazo, el líder disminuye *NextIndex* y vuelve a intentar el RPC *AppendEntries*. Con el tiempo *NextIndex* llegara a un punto en el que coinciden los registros del líder y el seguidor. Cuando eso suceda, *AppendEntries* tendrá éxito, lo que elimina cualquier entrada conflictiva 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 durante el resto del término.

Raft puede aceptar, replicar y aplicar nuevas entradas de registro siempre que la mayoría de los servidores estén activos.

3.3. Seguridad

Los mecanismos descritos hasta ahora no son suficientes para garantizar que cada máquina de estados 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, entonces, luego podría ser elegido líder y sobrescribir estas entradas con otras nuevas; como resultado diferentes máquinas de estados pueden ejecutar diferentes secuencias de comandos.

En esta sección, se agrega una restricción sobre qué servidores pueden ser elegidos líderes. La restricción asegura que el líder para cualquier término determinado contenga todas las entradas comprometidas en términos anteriores.

3.3.1. Restricción de la elección

Raft garantiza que todas las entradas comprometidas de términos anteriores 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 líderes a 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. Si el registro del candidato está al menos tan actualizado como cualquier otro registro en la mayoría del cluster, entonces contendrá todas las entradas comprometidas. El RPC *RequestVote* 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. Para determinar cuál de los dos registros está más actualizado, se compara el índice y el término de las últimas entradas en los registros. Si los registros tienen las últimas entradas con términos diferentes, entonces el registro con el último término está más actualizado. Si los registros terminan con el mismo término, el que sea más largo estará más actualizado.

3.4. Colapso de seguidores y candidatos

Los choques (crash) de seguidores y candidatos son mucho más simples de manejar que los choques de líderes, y ambos se manejan de la misma manera. Si un seguidor o candidato falla, los RPC de *RequestVote* y *AppendEntries* que se le envíen en el futuro fallará. Raft maneja estas fallas volviendo a intentarlo indefinidamente; si el servidor averiado se reinicia, el RPC se completará correctamente. Si un servidor falla después de completar un RPC, pero antes de responder, volverá a recibir el mismo RPC después de reiniciarse. Si un seguidor recibe una solicitud *AppendEntries* que incluye entradas de registro que ya están presentes en su registro, ignorará esas entradas en la nueva solicitud.

3.5. Interacción con el cliente

Los clientes de Raft envían todas sus solicitudes al líder. Cuando un cliente 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. Si el líder falla, las solicitudes del cliente expiran; los clientes luego intentarán de nuevo con servidores elegidos al azar.

Raf puede ejecutar un comando varias veces: por ejemplo, si el líder se bloquea después de la entrada del 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 rastreara el último número de serie procesado para cada cliente, junto con la respuesta asociada. Si recibe un comando cuyo número de serie ya se ha ejecutado, responde inmediatamente sin volver a ejecutar la solicitud.

Las operaciones de sólo lectura se pueden manejar sin escribir nada en el registro.

3.6 . Ventajas y desventajas del algoritmo

Para concluir con el algoritmo de Raft, procederemos a realizar un breve resumen, detallando tanto sus virtudes como debilidades (muchas de ellas ya comentadas y explicadas con anterioridad).

Como puntos fuertes a considerar sobre este algoritmo tenemos:

- Tolerancia a fallos. Permite que el sistema pueda seguir respondiendo y funcionando correctamente a pesar del fallo de un nodo.

- Consistencia fuerte. Garantiza que todos los nodos en el sistema contengan la misma información, de este modo, los datos vistos inmediatamente después de una actualización serán consistentes para todos los observadores de la entidad. Las inconsistencias internas son resueltas y no son expuestas a los clientes.

- Eficiencia. Algunos casos útiles que podemos apreciar son:
    - Al replicar nuevas entradas de registro, Raft logra un gran desempeño utilizando una cantidad mínima de mensajes (un solo viaje de ida y vuelta desde el líder hasta la mitad del grupo).
    - Las operaciones de sólo lectura se pueden manejar sin escribir nada en el registro.

- Sus propiedades de seguridad han sido formalmente especificadas y probadas, garantizando que cada máquina de estados de cada uno de los nodos en el sistema ejecute exactamente los mismos comandos en el mismo orden y nunca se retornen resultados incorrectos.

- Comprensibilidad y con bases claras, por ende, más fácil de aplicar y extender.
  Para esto, se utilizaron técnicas como:
    - Descomposición. Se separa la elección del líder, la replicación del registro y la seguridad.
    - Reducción de espacios de estados. Reduce el grado de no determinismo y la forma en que los servidores pueden estar inconsistentes entre sí.

Algunas falencias a destacar son las siguientes:

- Existe un gran cantidad de intercambios de mensajes entre todos los nodos del sistema debido a su mecanismo de replicación, el cual supone un mayor tiempo de procesamiento el cual debe ser considerado. Adicionalmente, estos mensajes son más densos que la mayoría, acarreando consigo una gran cantidad de información.

- El sistema posee serios problemas de escalabilidad, a causa de mantener la consistencia entre los múltiples nodos del sistema.

- Al poseer un único nodo privilegiado (líder) que gestione las solicitudes del sistema y se encargue de mantener consistente el resto de nodos, implica que se produzca un punto de escala para la mayoría de los mensajes, sobrecargando al nodo y provocando cuellos de botella.

- Raft no es un algoritmo tolerante a fallas bizantinas, los nodos confían en el líder elegido, y si este está realizando un mal trabajo sin que nadie lo compruebe, puede provocar problemas en todo el sistema con rapidez. Un mal líder tiene un radio de acción alto.

4. Consideraciones

A continuación detallaremos y explicaremos algunas consideraciones de implementación que se tuvieron en cuenta a la hora de abordar el proyecto, tanto generales, como módulos y clases utilizadas.

- El sistema fue diseñado enteramente en el lenguaje python, esta elección se debió principalmente por ser un lenguaje muy potente, fácil y sencillo de utilizar y por contar con una vasta comunidad, lo que permite el acceso a una gran cantidad de información disponible.

- Al momento de comunicarnos por las redes entre los distintos servidores/clientes del sistema, python nos brinda un módulo "*socket*" el cual nos permite establecer un protocolo UDP, para lograr dicha comunicación.

- Se escogió el protocolo UDP, debido a que como Raft maneja una gran cantidad de mensajes siendo enviados constantemente a los nodos seguidores, esto nos aseguraría que indudablemente, estos terminarían recibiendo los mensajes en algún momento, de este modo no serían necesarios los mecanismos de reenvío de datos de TCP, además este implicaría una mayor sobrecarga, y mayor complejidad en el código.

- Se optó almacenar la información requerida por el sistema distribuido en archivos, más concretamente en archivos json, ya que la información a almacenar es sencilla y la cantidad es demasiado pequeña como para implicar el uso de una base de datos y complejizar aún más el desarrollo del sistema.

- Para brindarle concurrencia al sistema, se utilizaron *threads* al momento de gestionar las peticiones de los clientes, y ciertos RPCs enviados por los nodos del clúster. Pero esto acarrea consigo problemas de sincronización como lo son las *Condiciones de carrera,* pudiendo generar resultados erróneos, para solucionar este problema, se recurrió al uso de *locks* (*mutex*) para lograr zonas de exclusión mutua en puntos claves del sistema.

- El proyecto fue ejecutado y testeado en OS Windows 10 y Ubuntu 20.04.

- Problema de conectividad Linux-Windows: Para generar la configuración de los host *(raft_setup.py*) se utiliza el comando "*socket.gethostbyname(socket.gethostname())*" el cual retorna la IP privada de la máquina (ejemplo: 192.168.0.1), pero en Linux este la retorna como "127.0.0.1" lo cual imposibilita la conexión desde un dispositivo Linux a otro con un diferente O.S. en la misma red local. Por ello, se deben comentar (#) en el archivo host (/etc/hosts) las líneas que contienen las IP 127.0.0.1 y 127.0.1.1.

4.1. Nodos

Un nodo (*node.py*) representa un único servidor en todo el sistema, son quienes contienen todas las funcionalidades y atributos necesarios para la correcta implementación de Raft, tales como mecanismos para la elección de un líder y replicación de logs, actualización y guardado de datos, etc.

Algunos atributos básicos de un Nodo a destacar son:

- **Node_id**: Un identificador único para el nodo en toda la red.
- **Address**: La dirección IP del nodo en la red.
- **Leader_address**: La dirección IP del último líder conocido hasta ese momento.
- **State**: El estado actual del nodo (LEADER / FOLLOWER / CANDIDATE).
- **Node_list**: Lista de todos los nodos en el sistema (sin considerarse a sí mismo). Cada nodo en esta lista está representado por una clase Host, la cual es una tupla con el id del nodo y su dirección.
- **Dictionary_data**: El recurso compartido en el sistema, en este caso, es un diccionario con sus claves y valores.
- **Logs**: Registro de entradas; Cada entrada contiene un comando para la máquina de estados y un término.
- **Commit_index**: Índice de la entrada de registro más grande conocida para ser comiteada.
- **Last_applied**: Índice de la entrada de registro más grande aplicada a la máquina de estados.
- **Quorum_size**: Cantidad de nodos requeridos para llegar a un consenso.
- **Current_term**: Último término del cual el nodo tiene conocimiento.

En caso de que el nodo sea el líder, este almacena información sobre el resto de los nodos en el sistema, para ayudarlo en la tarea de replicar los logs correctamente y lograr consistencia en los datos. Alguno de estos atributos son:

- **Next_index:** Para cada nodo, el índice de la próxima entrada de registro para ser enviada a ese nodo (inicializar en el último índice de registro +1 del líder).
- **Match_index**: Para cada nodo, índice de la más grande entrada de registro conocida para ser replicada en el servidor (inicializar en 0).

Varios atributos de un nodo (entre ellos, muchos de los nombrados anteriormente) son almacenados en un medio de almacenamiento estable, en nuestro caso, utilizamos archivos json, los cuales son actualizados en el momento anterior al enviarse un RPC. Esto se realiza con el objetivo de poder recuperar la información del nodo al momento de ejecutarlo por primera vez, o en caso de que este falle en algún punto.

Los archivos json se encuentran ubicados en el path: "*configs/server-n.json*" (donde n es el número de un servidor en particular).

4.2. Servidores

Un servidor (*server.py*) es simplemente un módulo, donde este se ve representado a través de un nodo, y se gestionan las peticiones de los clientes junto a los posibles RPCs del resto de servidores. Es aquí también, donde se controlan los tiempos de espera para las elecciones de líderes, y mensajes de heartbeat. Los servidores son inicializados utilizando como argumento un archivos de configuración "*configs/server-n.json*".

4.3. Cliente

Los clientes (*client.py*) son una interfaz gráfica, que le permiten al usuario interactuar con el sistema distribuido, enviando mensajes “ClientRequest” y gestionando las respuestas a los mismos por parte de los servidores.

Estos clientes son inicializados por medio de un archivo json, el cual contiene la configuración inicial para este cliente, este archivo le proporciona el puerto por el cual va a responder, y una lista de todos los servidores del sistema (identificador y dirección que cada servidor) para comunicarse con ellos.

Estos archivos json se encuentran ubicados en el path: “*configs/client-n.json*” (donde n es el número de un cliente en particular).

4.4. Mensajes

Un mensaje (*message.py*), posee el contenido que se enviara via UTP en la comunicación entre los servidores del sistema y los distintos clientes. Estos mensajes contendrán todos los campos requeridos para cada uno de los diferentes tipos de RPCs que utiliza el protocolo de Raft, independientemente si se utilizan o no (en caso de no utilizarse, estos campos estarán vacíos).

Podemos nombrar algunos campos comunes a todos los tipos de mensajes, como lo son el caso de:

- **From_id**: Identificador único del cliente o servidor que envía el mensaje.
- **From_address**: Dirección de quien envió el mensaje.
- **To_address**: Dirección del destinatario.
- **Msg_type**: Tipo del mensaje enviado (*RequestVote / AppendEntries / ClientRequest*).
- **Direction**: Sentido del mensaje (*Request / Reply*).

4.5. Utils

Este módulo (*utils.py*), contiene varias clases, métodos, y parámetros utilizados en el resto de módulos para facilitar la implementación y legibilidad del código. A continuación procederemos a nombrar y explicar cada uno de ellos.

4.5.1. Host

Un Host es la mínima representación de un nodo y son utilizados para facilitar la comunicación entre los servidores del sistema y los clientes. Éstos sólo contienen el identificador único del nodo y la dirección del mismo.

4.5.2. Log

Un log es un campo perteneciente al registro de logs de un servidor, y son utilizados en el proceso de replicación de registros entre todos los servidores del sistema. Estos contienen un comando enviado por un cliente y un término que identifica el momento en el que el log fue agregado.

4.5.3. Command

Un Command representa una instrucción enviada por un cliente a un servidor para ser ejecutada en su máquina de estados.

Estos comandos contienen:

- La dirección del cliente quien lo creó.
- Un número serial único para identificarlo.
- La acción a ser ejecutada (GET / SET).
- La posición en la cual la acción tomará efecto.
- El nuevo valor a ser aplicado (en caso que la acción sea un “SET”).
- El valor anterior que tenía la máquina de estados en esa posición, antes de ejecutar el comando.
- Un campo que señale si el comando ha sido ya ejecutado.

4.5.4. Parametros de Configuracion

Se definieron cuatro parámetros en un archivo json de configuración (params.json), los cuales son utilizados con el objetivo de administrar los tiempos de espera y envió de mensajes en el protocolo de comunicación de Raft, estos parámetros son:

Utilizados por el cliente

- ***TIME_TO_RETRY*:** Tiempo de espera máximo, para aguardar por un mensaje de respuesta por parte del sistema, a una petición enviada anteriormente.

- ***SERVER_TIMEOUT*:** Tiempo de espera máximo, para aguardar por un mensaje de respuesta por parte de un único servidor al que se le envio la peticion (Esto se debe a que, por ejemplo, si un servidor falla antes de enviar una respuesta, el cliente pueda proseguir enviando la misma petición a un servidor diferente).

Utilizados por el servidor

- ***HEARTHBEAT_TIMEOUT*:** Tiempo de espera por parte de un servidor líder para enviar un mensaje AppendEntries a el resto de servidores del sistema.

- ***ELECTION_INTERVAL***: Intervalo de espera, para aguardar por un mensaje AppendEntries enviado por parte del líder (Se toma un número aleatorio contenido en el intervalo dado).

4.6. Inicializando el sistema

Para que el sistema funcione y tanto servidores como clientes puedan ser ejecutados, ambos requieren disponer de sus archivos de configuración (ya nombrados anteriormente). Para esto, el módulo “**raft_setup.py”** es el encargado de su creación, al momento de ejecutarlo nos dispondremos a pasarle como argumentos la cantidad de servidores y clientes, esto creará sus respectivos archivos de configuración, por defecto, en caso de no brindar argumentos, se crearán un total de cinco servidores y tres clientes.

El sistema asumirá que está compuesto por la cantidad de servidores ingresados anteriormente, independientemente si estos están activos o no.

5. Referencias

[1] Ajay D. Kshemkalyani, Mukesh Shingal, *Distributed Computing. Principles, Algorithms and Systems.* Cambridge University Press. ISBN-13: 978-0-511-39341-9. 2008.

[2] Marcelo Arroyo. Notas del curso (slides) *Sistemas Distribuidos*. 2018.

[3] *In Search of an Understandable Consensus Algorithm.*
https://raft.github.io/raft.pdf

[4] *Consistency and Replication.*
http://docentes.uaa.mx/guido/wp-content/uploads/sites/2/2015/12/Consistencia.pdf

[5] *Arquitectura de los Sistemas Distribuidos.*
http://laurel.datsi.fi.upm.es/_media/docencia/asignaturas/sd/arquitecturas_parte2-4pp.pdf

[6] *Replicación.*
http://aisii.azc.uam.mx/areyes/archivos/licenciatura/sd/U3/ConceptoReplicacion.pdf

[7] Wikipedia, *Raft (algorithm).*
https://en.wikipedia.org/wiki/Raft_(algorithm)#cite_note-paper-1

[8] *Raft: Understandable Distributed Consensus*
http://thesecretlivesofdata.com/raft/